KAPITEL 3

Mit Daten arbeiten

Nachdem Sie R installiert haben, können wir uns jetzt dem eigentlichen »Werkstoff« von Statistik zuwenden: den Daten.

Dazu werden wir uns ansehen, wie R Daten speichert, sodass Sie mit ihnen arbeiten können. Zunächst betrachten wir *einfache Variablen*, die zum Beispiel eine einzelne Zahl aufnehmen. Danach werden Sie *Vektoren* kennenlernen, spezielle Variablen, die mehrere gleichartige Datenpunkte speichern, zum Beispiel alle Ergebniswerte einer Reihe von Messungen. *Dataframes* schließlich bestehen wiederum aus mehreren Vektoren und sind das Vehikel, mit dem Sie in R ganze Datensätze bearbeiten. Wenn Sie sich mit diesen grundlegenden Konzepten vertraut gemacht haben, können wir im letzten Abschnitt dieses Kapitels zur Tat schreiten und erstmals unseren Beispieldatensatz nach R einlesen.

Einfache Variablen und Zuweisungen

In diesem Abschnitt erfahren Sie,

- was Variablen in R sind und wozu sie verwendet werden,
- wie Sie Variablen Werte zuweisen,
- wie Sie sich den Wert von Variablen anzeigen lassen,
- welche Typen von Variablen es gibt und welche Art von Daten sie aufnehmen können,
- wie Sie Variablen in eine Variable eines anderen Typs konvertieren,
- wie Sie Variablen köschen können.

Folgende R-Funktionen werden in diesem Abschnitt behandelt:

- **`<-`**: Weist einer Variablen einen Wert zu.
- **`class`**`()`: Zeigt den Typ einer Variablen an oder weist einer Variablen einen bestimmten Typ zu.
- **`levels`**`()`: Zeigt die möglichen Ausprägungen einer Faktorvariablen, das heißt einer kategorialen Variablen, an.

- **`ls`**(): Listet alle aktuell existierenden Variablen auf.
- **`rm`**(): Löscht eine oder mehrere Variablen.

Variablen als Datenspeicher

In diesem Kapitel beginnen wir die eigentliche Arbeit mit R. Dazu müssen wir im ersten Schritt die Daten, die wir analysieren wollen, nach R importieren. R speichert Daten in *Variablen*. Deshalb ist es wichtig, dass Sie, bevor wir das erste Mal unseren Datensatz nach R laden, zunächst das Konzept der Variablen und die Art, wie R Daten speichert und manipuliert, verstehen.

Variablen in R haben die gleiche Funktion wie jene Variablen, die Sie aus der Mathematik kennen. Es sind Platzhalter für Werte. So, wie Sie im realen Leben Gegenstände in einen Karton legen und diesen beschriften können, so können Sie in R Ihre Daten in Variablen ablegen und diesen einen Namen geben. Fortan können Sie diesen Namen verwenden, um auf den Inhalt, das heißt die Daten, zuzugreifen.

Diejenigen, die sich vielleicht schon etwas genauer mit R befasst haben, wissen, dass es in R eigentlich gar nicht die Variable selbst ist, die den Wert beinhaltet. Die Variable ist tatsächlich nur ein *Symbol*, das auf ein anderes Objekt zeigt. Dieses Objekt trägt den eigentlichen Wert. Sie können sich das Symbol wie einen Griff vorstellen, mit dem Sie Ihren Karton hochheben. Der Wert ist im Karton, aber Sie nutzen den Griff, um an ihn heranzukommen. Wir werden diese feine Unterscheidung im Sprachkonzept von R in diesem Buch nicht weiter berücksichtigen und stattdessen nur noch von »Variablen« sprechen. Den Begriff »Objekt« verwenden wir synonym dazu. Eine genaue Kenntnis vom Konzept der Symbole wäre für die praktische Arbeit in R überhaupt nicht wichtig, wenn nicht manche der hin und wieder auftretenden Fehlermeldungen, die in R (ebenso wie in den meisten Programmiersprachen) nicht eben vor Verständlichkeit und Klarheit strotzen, von »Symbolen« sprechen würden. Wir werden uns einige dieser Fehlermeldungen noch genauer anschauen.

Die Inhalte, die in Variablen gespeichert werden, können sehr unterschiedlich sein. Sie denken wahrscheinlich am ehesten an Zahlen. Statistik, das ist doch schließlich eine Arbeit mit Zahlen, oder? Und in der Tat nehmen Variablen natürlich oftmals Zahlen auf. Gleichwohl können auch ganz andere Arten von Daten den Inhalt von Variablen bilden, zum Beispiel Zeichen (etwa Buchstaben) oder logische Werte, also die Wahrheitswerte *wahr* und *falsch* (bzw., da R wie die meisten Programmiersprachen seine Schlüsselwörter der englischen Sprache entnimmt, `TRUE` und `FALSE`). In den folgenden Abschnitten werden Sie die wichtigsten dieser Variablentypen eingehender kennenlernen.

Manche Programmiersprachen verlangen, dem Programm vor der erstmaligen Verwendung einer Variablen mitzuteilen, dass man gern mit einer Variablen dieses

Namens arbeiten möchte. Man muss, wie man auch sagt, die Variable *deklarieren*, das heißt beim Programm anmelden. Dadurch werden die Programme robuster gegenüber Tippfehlern: Wenn Sie sich nämlich beim Schreiben eines Variablennamens vertippen, merkt das Programm, dass hier eine Variable verwendet werden soll, die es gar nicht gibt, die also noch nicht deklariert ist. Das Programm kann Sie dann mit einer entsprechenden Meldung auf den Fehler hinweisen. Das geht nicht in Programmiersprachen, in denen Variablen überhaupt nicht deklariert werden müssen. Zu diesen Sprachen zählt auch R. Wenn Sie sich in R beim Schreiben eines Variablennamens vertippen, denkt R, Sie wollten einfach eine neue Variable erzeugen, und legt eine für Sie an. Der Wert, der gegebenenfalls bereits in der Variablen mit dem richtig geschriebenen Namen gespeichert war, ist in der irrtümlich neu erzeugten Variablen natürlich nicht vorhanden. Dementsprechend tut Ihr Programm, wenn es mit der neu erzeugten Variablen arbeitet, gegebenenfalls nicht das, was es eigentlich sollte. Der Vorteil, Variablen nicht deklarieren zu müssen, ist andererseits natürlich, dass Sie flexibel sind und jederzeit einfach damit beginnen können, mit einer Variablen zu arbeiten. Und genau das machen wir jetzt!

Variablen erzeugen und mit Werten versehen

Betrachten Sie die folgende Codezeile:

```
> x <- 1
```

Hier weisen wir einer Variablen `x` (die wir in diesem Moment praktisch aus dem Nichts erzeugen) den Wert `1` zu. Beachten Sie den Pfeil! Das ist der Zuweisungsoperator. Er deutet an, welche Seite der Zuweisung diejenige ist, deren Wert ersetzt wird, in diesem Beispiel `x`. Man könnte also diese Codezeile auch lesen als »schreibe den Wert 1 in die Variable `x`«. Anders als in der Mathematik benutzt R nicht das Gleichheitszeichen, um Zuweisungen vornehmen, sondern den Pfeil. Zwar wird auch das Gleichzeitszeichen von R als Zuweisungsoperator erkannt (was gut ist, denn Sie werden besonders am Anfang oft aus Versehen das Gleichheitszeichen statt des Pfeils einsetzen), aber der empfohlene Standard ist in R der Pfeil. Dieser hat den Vorteil, dass er die Richtung der Zuweisungsbeziehung mit angibt. Und wenn die Richtung der Beziehung durch den Pfeil bestimmt wird, sind wir eigentlich nicht mehr darauf angewiesen, dass diejenige Seite der Zuweisung, deren Wert verändert wird, stets die linke ist. Versuchen wir es doch einmal umgekehrt:

```
> 1 -> y
```

Auch das ist eine gültige Zuweisung. Um das zu sehen, müssen wir uns den Inhalt der Variablen `x` und `y` anschauen. Das ist in R denkbar einfach:

```
> x
[1] 1
> y
[1] 1
```

Geben Sie einfach den Namen der Variablen in die R-Konsole ein, drücken Sie *Enter*, und schon zeigt Ihnen R, welchen Wert Ihre Variable hat. Das Angabe `[1]` sagt Ihnen, dass diese Zeile der R-Ausgabe mit dem ersten Wert beginnt. Das ist in unserem Beispiel nicht überraschend, schließlich beinhaltet unsere Variable ja nur einen Wert. Wenn wir uns später aber Vektoren anschauen, werden Sie sehen, dass die Information, mit welchem Element die Ausgabezeile beginnt, durchaus sehr hilfreich sein kann.

Lassen Sie uns nun auf die gleiche Art den Wert der Variablen `z` anzeigen:

```
> z
Error: object 'z' not found
```

R weist Sie mit dieser Fehlermeldung darauf hin, dass die Variable `z` gar nicht existiert.

Grundsätzlich wäre `z` aber ein *zulässiger* Variablenname. *Unzulässig* sind dagegen Variablennamen, die mit einer Ziffer beginnen. Versuchen Sie, eine Variable solchen Namens zu benutzen, verweigert R Ihnen die Gefolgschaft:

```
> 2maleins <- 2*1
Error: unexpected symbol in "2maleins"
```

Wenn schon Ziffern als erste Zeichen eines Variablennamens unzulässig sind, wie steht es dann mit Sonderzeichen? Die Antwort auf diese Frage hängt vom verwendeten Sonderzeichen ab. Während die Zuweisung

```
> .nullkommnull <- 0.0
```

tatsächlich eine gültige Zuweisung ist, führt zum Beispiel

```
> _pi <- 3.1415926535
Error: unexpected input in "_"
```

zu einer in diesem Zusammenhang eher kryptischen Fehlermeldung. Wählen Sie deshalb am besten stets Variablennamen, die mit einem Buchstaben beginnen, um allen Problemen aus dem Weg zu gehen.

Variablennamen in R beinhalten oft einen Punkt, zum Beispiel `subsample.a`, `subsample.b`. Der Punkt wird hier als Trennzeichen verwendet, um den zusammengesetzten Namen besser lesbar zu machen. In manchen Programmiersprachen hat der Punkt eine besondere Bedeutung und kann deshalb nicht in Variablenamen verwendet werden. Das ist in R anders. Deshalb werden Sie in R (und so auch in diesem Buch) häufiger Variablennamen sehen, die einen Punkt enthalten, als in anderen Programmiersprachen, in denen stattdessen andere Zeichen eingesetzt werden, zum Beispiel der Unterstrich (`_`) – etwa `subsample_a`, `subsample_b`. Natürlich sind Sie in der Wahl Ihrer Variablennamen in R frei und können, wenn Ihnen der Unterstrich mehr zusagt, auch diesen ausgiebig nutzen.

Anders als der Punkt und der Unterstrich sind bestimmte andere Zeichen in Variablennamen nicht erlaubt, insbesondere Operatoren wie beispielsweise `+`, `*` oder auch logische Operatoren wie `&`, die Sie später kennenlernen werden.

Vorsichtig sein müssen Sie ebenfalls mit der Groß- und Kleinschreibung. R unterscheidet nämlich zwischen beiden, wie das folgende Beispiel illustriert:

```
> einevariable <- 2
> einevariable
[1] 2
> EineVariable
Error: object 'EineVariable' not found
```

Die Programmierer unter Ihnen werden vielleicht den englischen Fachterminus »case sensitive« für diese Eigenschaft von R kennen.

Eine gute Strategie, Probleme mit der Groß- und Kleinschreibung zu umgehen, ist deshalb, grundsätzlich alle Variablennamen kleinzuschreiben und, wo nötig, den Punkt oder den Unterstrich dazu zu verwenden, verschiedene Namensbestandteile voneinander abzugrenzen.

Numerische Variablen

Numerische Variablen sind, wie der Name sagt, Variablen, die Zahlen aufnehmen. Solche Variablen haben Sie bereits im vorangegangenen Abschnitt kennengelernt. Neben ganzen Zahlen (engl. *Integer*) können numerische Variablen auch Fließkommazahlen aufnehmen. Dabei ist zu berücksichtigen, dass in R der Punkt, nicht das Komma, das Dezimaltrennzeichen ist.

Die Zuweisung

```
> x <- 3,5
Error: unexpected ',' in "x <- 3,"
```

führt daher zu einer Fehlermeldung.

Richtig dagegen ist eine Zuweisung dieser Form:

```
> x <- 3.5
```

Zeichenketten

Bisher haben wir Variablen lediglich mit Zahlen »beladen«. Variablen können aber genauso gut Zeichenketten aufnehmen:

```
> name.hund <- "Wuffi"
> name.hund
[1] "Wuffi"
```

Dabei ist es übrigens gleichgültig, ob Sie den Text in doppelte Anführungszeichen oder einfache Anführungszeichen einschließen.

Wenn Sie die Anführungszeichen vergessen, denkt R, Sie wollten der Variablen `name.hund` den Wert einer anderen Variablen mit dem Namen `Wuffi` zuweisen. Da eine solche Variable aber nicht existiert, erhalten Sie konsequenterweise eine Fehlermeldung:

```
> name.hund <- Wuffi
Error: object 'Wuffi' not found
```

Zeichenkettenvariablen werden in R auch *Character-Variablen* genannt (vom englischen Begriff *Character* – Zeichen).

Logische Werte

Ein wichtiger Datentyp neben Zahlen und Zeichenketten sind logische Variablen oder Wahrheitswerte, die angeben, ob ein logischer Ausdruck wahr oder falsch ist. Wenn Sie mit anderen künstlichen Computersprachen vertraut sind, kennen Sie diese Art von Variablen möglicherweise unter dem Namen *Bool'sche* (oder englisch *Boolean*) *Variablen*, benannt nach dem englischen Mathematiker *George Boole*, der im 19. Jahrhundert bemerkenswerte Fortschritte auf dem Gebiet der formalen Logik erzielte. Die formale Logik beschäftigt sich mit dem Wahrheitswert logischer Aussagen.

Eine häufig auftretende Art solcher logischen Aussagen ist der Vergleich zweier Werte. Das tun Sie in R mit dem Vergleichsoperator `==`. Anders als in der Mathematik wird in R ein doppeltes Gleichheitszeichen für Vergleiche verwendet. Sie erinnern sich, dass R Zuweisungen nicht nur mit dem normalen Zuweisungsoperator `<-` erlaubt, sondern auch mit dem Gleichheitszeichen. Damit Zuweisungen von Vergleichen leicht unterschieden werden können, ist der Vergleichsoperator ein doppeltes Gleichheitszeichen. Um nun zu prüfen, ob die Variable `name.hund` den Wert `Wauzi` hat, können Sie einfach den Vergleich in die R-Konsole eingeben:

```
> name.hund == "Wauzi"
[1] FALSE
```

R antwortet mit einem unmissverständlichen `FALSE`, die Werte auf beiden Seiten des Gleichheitsoperators entsprechen sich also nicht. Das Ergebnis dieses Vergleichs können Sie auch in einer neuen Variablen speichern:

```
> z <- name.hund == "Wauzi"
```

Diese vielleicht auf den ersten Blick etwas merkwürdig anmutende Anweisung bedeutet nichts anderes als: Vergleiche, ob die Variable `name.hund` den Wert `Wauzi` hat, und speichere das Ergebnis in der Variablen `z`. Um zu überprüfen, ob R diese Anweisung genau so verstanden hat, lassen Sie sich den Wert von `z` einfach mal anzeigen:

```
> z
[1] FALSE
```

Wir haben also den Wert von `z` durch den Vergleich `name.hund == "Wauzi"` ermittelt. Sie können aber ebenso gut einer Variablen direkt einen logischen Wert zuweisen:

```
> z <- TRUE
```

Beachten Sie hier, dass TRUE nicht von Anführungszeichen umschlossen ist. Die Wahrheitswerte TRUE und FALSE sind Konstanten, die R direkt als Wahrheitswerte erkennt. Nur indem Sie auf die Anführungszeichen verzichten, weiß der R-Interpreter, dass die Variable z einen logischen Wert beinhalten soll. Die Zuweisung

```
> z <- "TRUE"
```

dagegen lässt R denken, dass z eine Zeichenkette beinhaltet.

Die Konstanten TRUE und FALSE können Sie in R übrigens auch kürzer als T und F schreiben:

```
> z <- T
```

Allerdings empfiehlt es sich, TRUE und FALSE stets auszuschreiben, da nur so ausgeschlossen werden kann, dass Sie irrtümlich mit einer gleichnamigen Variablen arbeiten. Hätten Sie eine Variable T, würde im Beispiel durch z <- T der Variablen z nicht etwa der Wert TRUE, sondern der Wert der Variablen T zugewiesen werden. Dieses Missverständnis können Sie verhindern, indem Sie mit den Konstanten TRUE und FALSE arbeiten.

Wahrheitswerte sind, wie Sie später sehen werden, in der Welt der computergestützten Statistik viel nützlicher, als Sie nach dem Genuss von (theoretischen) Statistikvorlesungen und -büchern vermuten werden. Deshalb ist es durchaus entscheidend, ob ein Wert ein echter logischer Wert oder aber eine Zeichenkette ist.

Ist Ihnen aufgefallen, dass wir TRUE immer in Großbuchstaben geschrieben haben? Sie erinnern sich sicher, dass R strikt zwischen Groß- und Kleinschreibung unterscheidet und TRUE daher für R etwas gänzlich anderes ist als true. Achten Sie also auch bei den logischen Konstanten stets auf die korrekte Groß- und Kleinschreibung!.

Übrigens: Auch wenn R Ihnen logische Werte immer mit den »hübschen« Konstanten TRUE und FALSE anzeigt und es Ihnen erlaubt, logische Werte über diese Konstanten zu bearbeiten, so speichert R doch »unter der Haube« statt TRUE den numerischen Wert 1 und statt FALSE den Wert 0. Das scheint auf den ersten Blick nicht weiter wichtig (schließlich können Sie ja stets mit den Konstanten TRUE und FALSE arbeiten), aber es hat eine interessante Konsequenz: Sie können mit logischen Werten auch rechnen:

```
> z <- TRUE
> z*5
[1] 5
```

Da R intern den logischen Wert TRUE durch 1 repräsentiert (und FALSE durch 0), führt die Multiplikation dieses Werts mit einer Zahl wiederum zu einer Zahl. Was an dieser Stelle noch wie Spielerei aussehen mag, wird sich an späterer Stelle als sehr nützlich erweisen.

Faktoren

Ein weiterer wichtiger Datentyp sind *Faktoren*. Variablen dieses Typs sind Ihnen in der Statistik möglicherweise schon unter dem Namen *kategoriale Variablen* begegnet. Dies sind Variablen, die nur eine bestimmte endliche Menge an Ausprägungen annehmen können.

Stellen Sie sich vor, Sie legen einen Datensatz über die Hunde an, die in einer Hundeschule angemeldet sind. Jeder Hund gehört einer bestimmten Hunderasse an, die Sie im Datensatz mit der besonderen Variablen `hunde.rasse` erfassen. »Dalmatiner«, »Golden Retriever«, »Boxer« und »Dackel« sind einige Ausprägungen von Hunderassen. Vermutlich werden Sie in der Hundeschule mehr als einen Hund haben, der einer bestimmten Rasse angehört. Welche Ausprägungen Ihre Faktorvariable annehmen kann, können Sie sich mit dem Befehl `levels` anzeigen lassen:

```
> levels(hunde.rasse)
[1] "Boxer" "Dackel" "Dalmatiner" "Golden Retriever"
```

Wenn Sie sich wie üblich durch Eingabe des Namens der Variablen deren Inhalt anschauen möchten, zeigt Ihnen R bei einer Faktorvariablen als zusätzliche Information die *Levels*, also die möglichen Ausprägungen, der Variablen an:

```
> hunde.rasse
[1] Golden Retriever
Levels: Boxer Dackel Dalmatiner Golden Retriever
```

Faktoren eignen sich also hervorragend dazu, kategoriale Variablen abzubilden. In unserem Datensatz ist übrigens die Variable `INCOMEGROUP` eine solche kategoriale Variable. `INCOMEGROUP` reflektiert das Einkommensniveau der Länder und hat die Levels `High income: nonOECD`, `High income: OECD`, `Upper middle income`, `Lower middle income` und `Low income`. Sie ist damit im Wesentlichen eine *ordinale Variable*, das heißt eine Variable, deren Ausprägungen sich zwar in eine natürliche Reihenfolge bringen lassen, bei der aber nicht notwendigerweise zwei benachbarte Ausprägungen gleich weit voneinander entfernt liegen: Der Einkommensunterschied zwischen `Low income` und `Lower middle income` einerseits und `Lower middle income` und `Upper middle income` andererseits mag durchaus unterschiedlich sein, abhängig von den zugrunde liegenden Definitionen. Nun wäre es bei den Einkommenskategorien noch prinzipiell möglich, dass die Kategorien gleich weit voneinander entfernt liegen. Das liegt daran, dass den Einkommenskategorien eine *metrische Variable*, nämlich das Einkommen, zugrunde liegt. Diese metrische Variable zeichnet sich dadurch aus, dass sie auf einer fortlaufenden quantitativen Skala gemessen wird, bei der die Abstände zwischen zwei Ausprägungen immer gleich sind: Der Einkommensunterschied zwischen 2.000 und 3.000 Euro ist derselbe wie der zwischen 11.000 und 12.000 Euro. Unsere Einkommenskategorien sind durch Aufteilung der metrischen Skala in einzelne Klassen oder Abschnitte entstanden, die (außer im Fall der `High income`-Kategorien, die natürlich nach oben offen sind) eine bestimmte Breite und damit auch einen Mittelpunkt besitzen. Aber es gibt auch ordinale Variablen, bei denen sofort klar wird, dass es grundsätzlich unmög-

lich ist, die Abstände der Kategorien auch nur zu ermitteln. Denken Sie an die Frage, wie Ihr Gesamteindruck von der Qualität der letzten Statistikvorlesung ist: `Sehr gut`, `gut`, `mittel`, `weniger gut`, `schlecht`. Es ist logisch, dass hier der Unterschied zwischen `gut` und `sehr gut` und zwischen `mittel` und `gut` nicht zwingend de gleiche sein muss (und wie würden wir diesen Unterschied überhaupt bestimmen wollen?). Trotzdem lassen sich die Ausprägungen der Variablen in eine Reihenfolge bringen. Genau das zeichnet ordinale Variablen aus. Und genau diese Eigenschaft fehlt der zweiten Art kategorialer Variablen, den *nominalen Variablen*. Ein Beispiel für nominale Variablen sind die Hunderassen. Zwar gibt es eine endliche Zahl von Hunderassen und damit von Ausprägungen unserer Variablen, doch diesen Kategorien liegt keine natürliche Reihenfolge zugrunde. Die Ausprägungen stehen einfach gleichberechtigt nebeneinander.

Ganz gleich, ob den Ausprägungen Ihrer Variablen eine Ordnung (lateinisch *ordo*) zugrunde liegt, es sich also um eine ordinal skalierte Variable handelt, oder ob die Ausprägungen Ihrer Variablen lediglich Namen (lateinisch *nomen*) darstellen und Sie es deshalb mit einer nominal skalierten Variablen zu tun haben: Faktoren eignen sich für beide Arten kategorialer Variablen gleichermaßen.

Bleiben wir nach diesem kurzen Ausflug in die Unterscheidung von Variablenarten noch kurz einen Moment bei unserem Beispiel mit den Hunderassen.

Angenommen, Sie stellen fest, dass der Hund, den Sie sich gerade anschauen, irrtümlich als Golden Retriever klassifiziert wurde, obwohl es sich um einen Boxer handelt. Um das zu korrigieren, müssen Sie der Variablen `hunde.rasse` das Level `Boxer` zuweisen:

```
> hunde.rasse
[1] Golden Retriever
Levels: Dalmatiner Golden Retriever Boxer Dackel
> hunde.rasse <- "Boxer"
> hunde.rasse
[1] Boxer
Levels: Dalmatiner Golden Retriever Boxer Dackel
```

Wie Sie sehen, weisen Sie also eine Faktorvariable genau so zu wie eine Zeichenkette, nämlich indem Sie den zugewiesenen Wert in Anführungszeichen setzen. Anders als bei Zeichenketten sind aber auf der rechten Seite der Zuweisung nicht alle möglichen Werte erlaubt, sondern nur die Levels, das heißt die zulässigen Ausprägungen des Faktors.

An diesem Punkt ist große Vorsicht geboten! Wenn Sie einer einzelnen Variablen vom Typ Faktor eine Zeichenkette zuweisen, die nicht zu den Levels des Faktors gehört – und sei es auch nur, weil Sie die Groß- und Kleinschreibung anders handhaben als in den eigentlichen Levels des Faktors –, wandelt R Ihre Variable in eine Zeichenkette um:

```
> hunde.rasse <- "boxer"
> hunde.rasse
[1] boxer
```

Sie sehen, dass unter der Ausgabe von R die für Faktoren typische Angabe der Levels fehlt. Das liegt daran, dass Ihre Variable gar kein Faktor mehr ist. R hat sie klammheimlich in eine Zeichenkette, also eine Character-Variable, umgewandelt, damit die Variable den Wert, den Sie ihr zuweisen wollen, auch aufnehmen kann.

Datentypen von Variablen ermitteln und konvertieren

Wie Sie am Ende des vergangenen Abschnitts gesehen haben, kann es interessant sein, festzustellen, von welchem Typ eine bestimmte Variable eigentlich ist. Das ist in R mit der Funktion `class` leicht möglich:

```
> class(hunde.rasse)
[1] "factor"
```

Die Variable `hunde.rasse` ist also vom Typ Faktor. Weitere wesentliche Typen neben `factor` sind: `character` für Zeichenketten, `logical` für Wahrheitswerte sowie `numeric` und `integer` für Zahlen.

Zu beachten ist hier, dass R für Zahlen zwei verschiedene Typen verwendet. Der Typ `numeric` ist dabei der allgemeinere, während `integer` von R nur für solche Variablen verwendet wird, die eindeutig Ganzzahlen sind. Sie brauchen sich um diesen Unterschied keine weiteren Gedanken zu machen, wichtig zu erinnern ist lediglich, dass beides Typen für Variablen sind, die Zahlen aufnehmen können.

Für diejenigen, die schon etwas Programmiererfahrung mitbringen und später vielleicht tiefer in die R-Programmierung einsteigen wollen, sei an dieser Stelle kurz darauf hingewiesen, dass der Name der Funktion `class` daran erinnert, dass R, obwohl es auf den ersten Blick vielleicht nicht den Anschein hat, eine objektorientierte Sprache ist und damit auf einem Klassenkonzept aufsetzt. Objekte haben eine `class`-Eigenschaft, die mit dieser Funktion abgefragt wird.

Die Funktion `class` ist aber keine Einbahnstraße: Sie können sie auch dazu verwenden, den Typ von Variablen zu *ändern*. Betrachten Sie den folgenden Codeabschnitt:

```
> x <- "10"
> x*5
Error in x * 5 : non-numeric argument to binary operator
```

Hier wird eine Variable `x` erzeugt, und ihr wird der Wert `"10"` zugewiesen. Wie Sie mittlerweile wissen, führt das Umschließen des zugewiesenen Werts in Anführungszeichen dazu, dass die Variable eine Zeichenkette, eine Variable vom Typ `character`, wird. Mit Zeichenketten kann natürlich nicht gerechnet werden. Und deshalb gibt R Ihnen beim Versuch, die `character`-Variable `x` mit 5 zu multiplizieren, eine (zugegeben) etwas kryptische Fehlermeldung aus. Wie können Sie nun diese Variable in eine numerische Variable umwandeln? Hier machen wir uns die Möglichkeit zunutze, die Funktion `class` auch umgekehrt, das heißt zum Festlegen des Typs, zu verwenden:

```
> class(x) <- "numeric"
> x*5
[1] 50
```

Nach der Umwandlung der `character`-Variablen in eine numerische Variable kann damit auch gerechnet werden.

Diese Umwandlung oder *Konvertierung* von Variablen geht natürlich auch mit anderen Typen ...

```
> x <- "TRUE"
> x
[1] "TRUE"
> class(x)
[1] "character"
> class(x) <- "logical"
> x
[1] TRUE
> class(x)
[1] "logical"
```

... und in andere Richtungen. Zum Beispiel können Sie so auch eine Zahl in eine Zeichenkette umwandeln.

Was immer Sie wohinein umwandeln: Vergewissern Sie sich zuerst, dass die Variable, die Sie umwandeln wollen, auch wirklich einen Inhalt hat, der für den Zielvariablentyp Sinn ergibt. Ein Beispiel, in dem das nicht der Fall ist:

```
> x <- "20 Hunde"
> class(x) <- "numeric"
> class(x)
[1] "numeric"
> x
[1] NA
```

Hier wird versucht, die Zeichenkette `"20 Hunde"` in eine numerische Variable zu konvertieren, was natürlich nicht gelingt. Anstatt aber den Versuch mit einer Fehlermeldung zurückzuweisen, konvertiert R die Variable wie gewünscht, ändert aber ihren Inhalt zu `NA`. Wir werden uns im nächsten Abschnitt mit der Bedeutung dieses `NA` näher auseinandersetzen, weil es für die statistische Arbeit mit echten Daten von großer Bedeutung ist. An dieser Stelle sei aber schon einmal gesagt, dass `NA` (die Abkürzung des englischen Begriffs *Not Available* – nicht verfügbar) anzeigt, dass Ihre Variable keinen Inhalt hat. R hat also, ohne uns darüber zu informieren, zwar den Typ der Variablen wie gefordert geändert, aber den Inhalt der Variablen gelöscht, weil er nicht zu dem neuen Typ passt. Achten Sie daher genau darauf, welche Inhalte Ihre Variablen haben, bevor Sie sie in andere Typen konvertieren.

Sie können übrigens Variablen nicht nur dadurch konvertieren, dass Sie ihnen mit `class` einen neuen Typ zuweisen. R stellt auch eine Reihe von Funktionen zur Verfügung, die einen Typkonvertierung erlauben, darunter die Funktionen `as.numeric`, `as.character`, `as.logical` und `as.factor`. Diese Funktionen konvertieren die ihnen

übergebene Variable bzw. den ihnen übergebenen Wert in den jeweiligen Typ, der im Funktionsnamen genannt wird. Ein Beispiel:

```
> wahr <- "TRUE"
> class(wahr)
[1] "character"
> wahr <- as.logical(wahr)
> wahr
[1] TRUE
> class(wahr)
[1] "logical"
```

Die Variable `wahr` ist zunächst eine Zeichenkettenvariable. Durch Anwendung von `as.logical` machen wir sie zu einer logischen Variablen.

Variablen löschen

Nun haben Sie schon einiges über den Umgang mit einzelnen Variablen gelernt, eines aber noch nicht: Wie werden Sie eine Variable, die Sie einmal angelegt haben, wieder los? In den vorherigen Abschnitten haben wir eine ganze Reihe von Variablen angelegt. Diese Variablen können Sie sich mit der Funktion `ls` anzeigen lassen:

```
> ls()
[1] "hunde.rasse" "name.hund"   "x"           "y"           "z"
```

R listet die bisher angelegten Variablen alphabetisch auf (`ls` ist eine Kurzschreibweise für *list*). Beachten Sie, dass die Funktion `ls` mit den Klammern, also `ls()`, aufgerufen werden muss. Was zunächst vielleicht merkwürdig anmutet, hat eine einfache Erklärung: Funktionen in R können, wie den Funktionen in der Mathematik auch (denken Sie zum Beispiel an Sinus und Kosinus), Argumente übergeben werden. Das haben wir eben bei der Funktion `class` gesehen, der Sie als Argument eine Variable übergeben und die Ihnen dann den Typ dieser Variablen anzeigt. Anders als in der Mathematik *muss* eine Funktion in R aber keine Argumente haben. `ls` ist ein gutes Beispiel für eine Funktion, die ohne Argumente auskommt. Da Funktionen in R aber grundsätzlich so angelegt sind, dass sie Argumente übernehmen können, müssen eben dann, wenn eine Funktion keine Argumente besitzt, leere Klammern angegeben werden. Sollten Sie die Klammern vergessen, gibt Ihnen R den Quellcode der Funktion aus. Funktionen – das werden wir später lernen – sind im Kern eine Abfolge von R-Befehlen. Und diese Befehle zeigt Ihnen R an, wenn Sie die Funktion ohne Klammern eingeben. Wenn es Sie also schon immer mal interessiert hat, wie eine bestimmte Funktion programmiert ist: Lassen Sie einfach mal die Klammern weg. Probieren Sie es aus!

Nun nehmen wir an, Sie wollten die Variable `hunde.rasse` löschen. Das geschieht in R mit der Funktion `rm`:

```
> rm(hunde.rasse)
```

`rm`, die Kurzschreibform für *remove*, also *löschen*, erwartet als Argument die zu löschende Variable. Mit `ls()` können Sie sich rasch überzeugen, dass R die Variable tatsächlich entfernt hat:

```
> ls()
[1] "namehund"    "x"           "y"           "z"
```

Warum nun aber sollten Sie Variablen nach ihrer Verwendung löschen? Können die Variablen nicht einfach (ungenutzt) weiterexistieren? Das können sie natürlich. Die Zeiten, in denen Arbeitsspeicher ein stark limitierender Faktor war und Bill Gates angeblich seinen berühmten Ausspruch: »640K ought to be enough for anybody« (640 Kilobytes sollten genug für jeden sein) von sich gegeben haben soll (was er im Übrigen dementiert), sind lange vorbei. Trotzdem kann es vorkommen – gerade wenn Sie mit sehr großen Datensätzen oder mit Funktionen arbeiten, die als Ergebnis große Objekte zurückgeben –, dass der Speicher stark ausgelastet ist und R Ihnen im Extremfall gar den Dienst verweigert. In solchen Fällen ist eine kleine Variablenbereinigung oft der einzig gangbare Weg. Zudem schaffen Sie natürlich auch etwas mehr Übersicht, wenn Sie nicht mehr Benötigtes entfernen. Das gilt insbesondere in *Dataframes*, die wir uns später anschauen.

Vektoren

In diesem Abschnitt erfahren Sie,

- wie ein Vektor mehrere gleichartige Datenelemente zu einem R-Objekt zusammenfasst,
- wie Sie Vektoren erzeugen,
- wie Sie auf einzelne Elemente eines Vektors zugreifen und diese verändern,
- wie Sie die Länge von Vektoren ermitteln und verändern können,
- wie Sie mit fehlenden Daten, sogenannten Missings (`NA`), in einem Vektor umgehen.

Folgende R-Funktionen werden in diesem Abschnitt behandelt:

- **`c`**`()`: Erzeugt einen Vektor aus mehreren einzelnen Variablen oder Werten.
- **`length`**`()`: Gibt die Länge, das heißt die Anzahl der Elemente, eines Vektors zurück oder ändert die Länge eines Vektors.
- **`NROW`**`()`: Gibt die Länge eines Vektors zurück.
- **`is.na`**`()`: Prüft, ob eine einzelne Variable oder einzelne Elemente eines Vektors Missings, also ohne Daten, sind.

Vektoren anlegen

Bis hierher haben Sie einfache Variablen kennengelernt, Variablen, die genau einen Wert aufnehmen, eine Zahl, eine Zeichenkette, einen Wahrheitswert, einen

Faktor. Aber in der statistischen Arbeit haben wir es typischerweise mit mehr als nur einem Wert zu tun. So haben wir beispielsweise in unserem Datensatz die Staatsausgaben in Prozent des Bruttoinlandsprodukts *für alle Länder*.

R kennt ein spezielles Konstrukt, um mehrere gleichartige Werte aufzunehmen, den *Vektor*. Viele Funktionen in R erwarten einen Vektor als Argument, so zum Beispiel die Funktion `mean`, die wir uns später eingehender ansehen werden und die den arithmetischen Mittelwert einer Menge von Einzelwerten berechnet.

Vektoren werden in R mit einer speziellen Funktion erzeugt, der Funktion `c` (vom englischen Begriff *create* – erschaffen):

```
> x <- c(5,3,7)
> x
[1] 5 3 7
```

Die Funktion `c` nimmt als Argumente die Elemente des Vektors. Da Vektoren natürlich unterschiedlich groß sein können, ist die Zahl der Argumente von `c` variabel. Sie können auf diese Weise auch Vektoren erzeugen, die nur ein einziges Element enthalten. Damit stellen sich die einfachen Variablen, die Sie in den vorangegangenen Abschnitten kennengelernt haben, als Spezialfälle von Vektoren dar. Das heißt

```
> x <- 3
```

bewirkt dasselbe wie

```
> x <- c(3)
```

nämlich dass ein Vektor der Länge 1 mit dem Wert 3 angelegt wird.

Die Elemente müssen natürlich nicht unbedingt numerische Werte sein. Auch dies ist ein gültiger Vektor:

```
> sieliebtmich <- c(TRUE, TRUE, TRUE, FALSE, TRUE)
> sieliebtmich
[1]  TRUE  TRUE  TRUE FALSE  TRUE
```

Sie können der Funktion `c` nicht nur direkt »echte« Werte übergeben, sondern Vektoren auch aus anderen Variablen erzeugen:

```
> x <- 5
> y <- c(3)
> z <- 7
> koordinate <- c(x,y,z)
koordinate
[1]  5 3 7
```

Die Variablen, aus denen Sie einen Vektor erzeugen, können selbst wiederum Vektoren sein:

```
> x <- c(1,2)
> y <- c(3,4,5)
> z <- c(x,y)
> z
[1] 1 2 3 4 5
```

Mit der Anweisung z<-c(x,y) legen wir einen Vektor an, der die Elemente der beiden Input-Vektoren x und y kombiniert und sie in der angegebenen Reihenfolge aneinanderhängt. Die Input-Vektoren werden dabei also aufgelöst.

Was passiert nun, wenn die Elemente, aus denen ein Vektor bestehen soll, nicht vom selben Typ sind? Testen wir das mit einem Beispiel:

```
> z <- c(5, "Eine Zeichenkette", TRUE)
> z
[1] "5"                "Eine Zeichenkette" "TRUE"
```

Auf den ersten Blick scheint die ungewöhnliche Vektorerzeugung gut gegangen zu sein. Keine Fehlermeldung, die Elemente werden angezeigt. Auf den zweiten Blick sehen Sie jedoch, dass alle drei Elemente des Vektors, auch die Zahl 5 und der Wahrheitswert TRUE, von Anführungszeichen umschlossen sind. R behandelt die drei Elemente also alle als Zeichenketten. Das können Sie ja mittlerweile leicht überprüfen:

```
> class(z)
[1] "character"
```

R hat also automatisch einen Datentyp gesucht, der alle drei Werte aufnehmen kann. Der einzige Datentyp, der das in unserem Beispiel vermag, ist character, also eine Zeichenkette. Noch ein weiteres Beispiel dieser Art:

```
> z <- c(5, FALSE, TRUE)
> z
[1] 5 0 1
```

Hier sehen Sie erneut, dass R die Vektorelemente auf den »kleinsten gemeinsamen Nenner« konvertiert hat. Das ist in diesem Fall ein numerischer Typ. Die beiden Wahrheitswerte werden dabei als 0 und 1 abgebildet – wie R sie auch intern ablegt.

Merken Sie sich also, dass die Elemente von Vektoren immer vom selben Typ sein müssen.

In manchen Situationen ist es hilfreich, zu wissen, wie lang ein Vektor eigentlich ist, das heißt, wie viele Elemente er besitzt. Das können Sie in R mit den Funktionen NROW und length leicht feststellen:

```
> NROW(z)
[1] 3
> length(z)
[1] 3
```

Weiter oben hatten Sie am Beispiel der Funktion class gesehen, dass man in R manche Funktionen gewissermaßen »umdrehen« kann, um sich nicht eine bestimmte Eigenschaft von Variablen *anzeigen* zu lassen, sondern diese Eigenschaft zu *setzen*. Das funktioniert auch mit der Länge von Vektoren:

```
> length(z) <- 2
> z
[1] 5 0
```

Hier verkürzen wir der Vektor z auf zwei Elemente. Wenn Sie dasselbe mit der Funktion NROW versuchen, erhalten Sie eine Fehlermeldung. Die Länge eines Vektors ändern können Sie also nur mit length.

Auf die gleiche Art können Sie einen Vektor verlängern, das heißt zusätzliche Elemente anhängen:

```
> length(z) <- 5
> z
[1]  5  0 NA NA NA
```

Hier sehen Sie, dass R die neu hinzugefügten Elemente mit NA aufgefüllt hat. R kann nicht wissen, welche Werte diese neuen Vektorelemente besitzen sollen. Deshalb markiert R die neuen Elemente als *missing*, also fehlend. Was Missings genau sind und wie mit ihnen umzugehen ist, schauen wir uns im nächsten Abschnitt etwas eingehender an.

Bevor wir das tun, sei noch kurz auf eine praktische Operation in R hingewiesen: Sie werden manchmal Vektoren für ganze Zahlen benötigen, zum Beispiel die natürlichen Zahlen von 3 bis 10. Einen solchen Vektor können Sie in R ganz einfach erzeugen:

```
> n <-3:10
> n
[1] 3 4 5 6 7 8 9 10
```

R generiert dann einfach alle ganzen Zahlen zwischen den durch den Doppelpunkt getrennten Werten.

Mit Missings umgehen

Der »Wert« NA, den jede Variable in R annehmen kann, bedeutet *Not Available* und steht in R für ein *Missing*, also einen fehlenden Wert.

Missings sind ein wichtiges Konzept in der praktischen Statistik, denn oft werden Ihre Daten unvollständig sein, zum Beispiel weil Sie manche Daten einfach nicht haben finden können. Auch in unserem Beispieldatensatz gibt es Missings. Die Arbeitslosenquote für Angola beispielsweise war im World Economic Outlook des Internationalen Währungsfonds nicht enthalten. An dieser Stelle steht deshalb in unserem Datensatz ein Missing. Im Zusammenhang mit Missings ist es wichtig, zu verstehen, dass ein Unterschied besteht zwischen einem Datenpunkt, der den Wert 0 hat, und einem Datenpunkt, der gar keinen Wert hat, also ein Missing ist. Nehmen wir als Beispiel die Variable WOMENPARL, die in unserem Datensatz den Prozentsatz von Parlamentssitzen angibt, die von Frauen gehalten werden. Hat WOMENPARL den Wert 0, bedeutet dies, dass keine Frauen im Parlament sitzen, was in unserem Datensatz in Mikronesien, Palau und Katar der Fall ist. Wollten wir nun beispielsweise eine Aussage darüber treffen, wie hoch der *durchschnittliche* Frauenanteil in den Parlamenten der Länder in unserem Datenbestand ist, würden die Datensätze von Mikronesien, Palau und Katar in diesen Durchschnitt eingehen. Ein Datensatz, dessen »Wert« für WOMENPARL dagegen NA ist, würde im Durch-

schnitt nicht berücksichtigt werden. Merken Sie sich also, dass 0 ein echter Wert ist und dass ein Missing dagegen anzeigt, dass kein echter Wert vorliegt und der betroffene Datenpunkt deshalb nicht sinnvoll in statistische Analysen einbezogen werden kann.

Dass ein Missing nicht sinnvoll verarbeitet werden kann, sehen Sie am folgenden einfachen Beispiel:

```
> x <- NA
> x*5
[1]  NA
```

Hier wird zunächst eine Variable x mit NA belegt, also zu einem Missing gemacht. Dann wird versucht, die Variable mit 5 zu multiplizieren. Das führt aber wiederum zu einem Missing. Mit einem fehlenden Wert kann einfach nicht gerechnet werden. Auch hier sehen Sie sehr plastisch den Unterschied zum Wert 0.

In vielen Situation ist es hilfreich, zu ermitteln, ob eine Variable ein Missing ist. Das können Sie natürlich tun, indem Sie sich einfach den Wert der Variablen anzeigen lassen. Aber wenn Sie zum Beispiel die Missings aus einem ganzen Vektor herausfiltern wollen, wäre ein solches Vorgehen unglaublich mühselig. Hier hilft R Ihnen mit der Funktion `is.na`:

```
> is.na(x)
[1]  TRUE
```

Die Funktion gibt einen logischen Wert zurück, der anzeigt, ob die Variable, die Sie der Funktion als Argument übergeben haben, ein Missing ist. Sie können `is.na` natürlich auch auf einen Vektor anwenden. Das Ergebnis der Funktion ist in diesem Fall wiederum ein Vektor, nämlich ein Vektor von Wahrheitswerten, der anzeigt, an welchen Stellen des der Funktion `is.na` als Argument übergebenden Vektors Missings stehen. Angewendet auf unseren Vektor z mit den Werten 5, 0, NA, NA, NA, bedeutet das:

```
> is.na(z)
[1] FALSE FALSE  TRUE  TRUE  TRUE
```

Auf einzelne Elemente eines Vektors zugreifen

Häufig werden Sie in der praktischen statistischen Arbeit Vektoren als Ganzes betrachten und bearbeiten. Manchmal ist es aber auch nötig, auf einzelne Elemente eines Vektors zuzugreifen. Das geschieht in R durch die Angabe des Index des Vektorelements, das heißt seiner fortlaufenden Nummer, in eckigen Klammern. Betrachten wir wieder unseren Vektor z mit den fünf Elementen 5, 0, NA, NA, NA:

```
> z
[1]  5  0 NA NA NA
> z[1]
[1] 5
> z[3]
[1] NA
```

Genauso können Sie einzelne Elemente eines Vektors ändern:

```
> z[5] <- 7
> z
[1]  5  0 NA NA  7
```

Durch die Zuweisung `z[5]<-7` wird das fünfte Element des Vektors, das zuvor `NA` gewesen war, durch den Wert `7` ersetzt.

Vermutlich ohne weiter davon Kenntnis zu nehmen, haben Sie gerade en passant gelernt, dass die Indizierung von Vektorelementen bei `1` beginnt. Das mag Ihnen möglicherweise selbstverständlich vorkommen, ist es in der Welt der künstlichen Computersprachen aber keineswegs. Tatsächlich kommen viele Programmierfehler durch falsche Indizierung zustande. Falsche Indizierung kann Ihnen natürlich auch in R passieren:

```
> z[8]
[1]  NA
```

Hier wird versucht, auf das achte Element des Vektors zuzugreifen, obwohl er nur aus fünf Elementen besteht. Das achte Element ist demnach nicht verfügbar, R gibt konsequenterweise ein Missing (`NA`) aus.

Bisher haben wir Vektoren mit einfachen natürlichen Zahlen indiziert. Tatsächlich kann der Index aber selbst ein Vektor sein:

```
> index <- c(3,5)
> z[index]
[1] NA  7
```

Wir definieren im ersten Schritt einen Vektor `index` mit zwei Elementen: `3` und `5`. Diesen Vektor nutzen wir zum Zugriff auf die Elemente des Vektors `z`. R gibt uns entsprechend die an den Positionen 3 und 5 gesetzten Elemente des Vektors `z` zurück, das erste der beiden `NA` und den Wert `7`.

Könnten wir die mehrfache Indizierung auch direkt vornehmen, indem wir die Indizes einfach in den eckigen Klammern hintereinander auflisten? Probieren wir es:

```
> z[3,5]
Error in z[3, 5] : incorrect number of dimensions
```

Das geht also nicht. R denkt hier, wir wollten auf ein zweidimensionales Datenkonstrukt zugreifen, und zwar auf das dritte Zeilenelement und das fünfte Spaltenelement. Unser Vektor ist aber ein eindimensionales Gebilde. Tatsächlich haben wir hier versucht, mit zwei verschiedenen Vektoren, die jeweils nur aus einem einzigen Element bestehen, zu indizieren. Wir hätten also genauso gut `z[c(3), c(5)]` schreiben können. Damit wird aber auch die Lösung des Problems deutlich: Wir müssen R einfach mitteilen, dass unsere beiden Indizes `3` und `5` in Wirklichkeit nur *ein* Vektor sind. Und das machen wir natürlich, indem wir einen Vektor mit der Funktion `c` erzeugen:

```
> z[c(3,5)]
[1] NA  7
```

Die Indexvektoren haben natürlich weniger Elemente als die Vektoren, die indiziert werden, denn wir wollen ja eine Teilmenge aller Elemente des indizierten Vektors ermitteln.

Eine zweite Art der Indizierung nutzt einen Vektor von logischen Werten, der genauso lang ist wie der Vektor, der indiziert wird. Jeder einzelne Wahrheitswert gibt an, ob das im indizierten Vektor an der gleichen Stelle stehende Element indiziert ist oder nicht:

```
> index <- c(FALSE, FALSE, FALSE, TRUE, FALSE, TRUE)
> z[index]
[1] NA  7
```

An den Positionen 3 und 5 steht in unserem Indexvektor `index` der logische Wert `TRUE`, an allen anderen Stellen steht `FALSE`. R selektiert nun aus dem indizierten Vektor `z` alle Elemente, die an denselben Positionen wie die `TRUE`-Werte im Indexvektor stehen. Das Ergebnis ist dasselbe wie bei der Indizierung `z[c(3,5)]`.

Diese zweite Art der Indizierung ist viel mehr als nur eine Spielerei. In der Praxis ist sie sogar höchst relevant. So werden Sie zum Beispiel Variablen häufiger filtern müssen, um die Missings zu eliminieren. Das können Sie nun mit der gerade beschriebenen Technik wie folgt bewerkstelligen:

```
> z[is.na(z) == FALSE]
[1] 5 0 7
```

Dieser zusammengesetzte Ausdruck sieht kompliziert aus, lässt sich aber sehr einfach in seine Bestandteile zerlegen und aif diese Weise verstehen. Gehen wir also Schritt für Schritt vor: Der Index, der hier verwendet wird, um Elemente aus `z` zu selektieren, ist `is.na(z)==FALSE`. Der linke Teil ist die mittlerweile bekannte `is.na`-Funktion. Sie liefert

```
> is.na(z)
[1] FALSE FALSE  TRUE TRUE FALSE
```

An den Positionen 3 und 4 stehen in `z` Missings, also ist das Ergebnis von `is.na` dort `TRUE`. Mit dem Gleichheitsoperator `==` wird geprüft, ob der Wert von `is.na` falsch ist, also *kein* Missing vorliegt. Dieser Vergleich wird auf Ebene der einzelnen Elemente des Ergebnisvektors von `is.na` durchgeführt und ist daher selbst wieder ein Vektor:

```
> is.na(z) == FALSE
[1] TRUE TRUE FALSE FALSE TRUE
```

Und genau dieser Vektor wird zur Indizierung verwendet. Damit werden aus `z` die Elemente an den Positionen 1, 2 und 5 selektiert. Die gleiche Wirkung ergibt eine Indizierung, bei der der Indexvektor direkt angegeben wird:

```
> z[c(TRUE, TRUE, FALSE, FALSE, TRUE)]
[1] 5 0 7
```

Der Vorteil an der Formulierung `z[is.na(z)==FALSE]` besteht allerdings darin, dass sie sehr allgemein ist: Man muss nicht wissen, wo genau die Missings stehen, dazu dient die Funktion `is.na`.

Übrigens: Wann immer Sie bei einem Wahrheitswert nicht explizit eine Vergleichsgröße angeben, vergleicht R automatisch mit `TRUE`. Sie können also statt `is.na(z)==TRUE` auch einfach `is.na(z)` schreiben. Und deshalb kann statt der etwas umständlichen Schreibweise `is.na(z)==FALSE` auch einfach `!is.na(z)`eingegeben werden. Das Ausrufezeichen ist in R der *Negationsoperator*, das logische NICHT. Er dreht den Wahrheitswert eines logischen Ausdrucks um:

```
> !TRUE
[1] FALSE
```

Demnach lautet die kürzestmögliche Formulierung Ihrer Indizierung:

```
> z[!is.na(z)]
[1] 5 0 7
```

Dataframes

In diesem Abschnitt erfahren Sie,

- wie ein Dataframe einen ganzen Datensatz aufnimmt und so das zentrale Datenkonstrukt in R ist,
- wie Sie einen Dataframe erzeugen,
- wie Sie Dataframes anzeigen lassen und verändern können,
- wie Sie auf einzelne Variablen (Spalten) und einzelne Beobachtungen (Zeilen) in einem Dataframe zugreifen,
- wie Sie Variablen aus einem Dataframe löschen.

Folgende R-Funktionen werden in diesem Abschnitt behandelt:

- **`data.frame`**(): Erzeugt einen Dataframe aus mehreren Variablen.
- **`View`**(): Zeigt einen Dataframe in einem separaten R-Fenster an.
- **`edit`**(): Öffnet einen Dataframe zur manuellen Bearbeitung im R-Editor.

Nachdem Sie nun ein grundlegendes Verständnis von Vektoren entwickelt haben, ist es Zeit, das letzte wichtige Datenkonstrukt von R kennenzulernen.

Üblicherweise haben Sie es in der Statistik nicht mit einer einzelnen Variablen zu tun, sondern meist mit ganzen Datensätzen, so etwa unserem Beispieldatensatz. Dort gibt es, wie Sie noch sehen werden, für jedes Land eine ganze Reihe von Variablen, zum Beispiel die Staatsausgaben oder den Urbanisierungsgrad. Ein typischer Fall für einen Dataframe.

Dataframes sind tabellenartige Datenschemata, die man sich aus mehreren Vektoren als Spalten zusammengesetzt vorstellen kann. Jeder Vektor stellt dabei eine

Variable des Datensatzes dar. In den Zeilen des Datensatzes stehen die einzelnen Dateneinträge, die dadurch gekennzeichnet sind, dass sie für jede der Variablen einen einzelnen Wert besitzen. Wir werden diese Zeilen fortan als *Beobachtungen* bezeichnen. Dieser Ausdruck spiegelt natürlich die aus der wissenschaftlicen Anwendung herrührende Vorstellung wider, dass man bestimmte Variablen für unterschiedliche »Objekte« in der Realität *beobachtet* und ihre Ausprägungen pro »Objekt« notiert hat. Das »Objekt« kann zum Beispiel der Befragte einer Meinungsumfrage, ein Land, eine Materialprobe oder auch ein Zeitpunkt sein. Jedes »Objekt« bekommt eine eigene Zeile im Datensatz. Wenn Sie Erfahrung mit Datenbanken haben, können Sie diese Beobachtungen natürlich auch als *Records* bezeichnen. Wie auch immer Sie sie nennen mögen – letztendlich sind es die Träger jener Eigenschaften, die wir in den Variablen (das heißt den Spalten des Datensatzes) messen.

Wenn wir einen Datensatz als ein Konstrukt mehrerer Variablen betrachten, können wir seine Repräsentation in R, den Dataframe, natürlich sehr einfach aus Vektoren erzeugen:

```
> vorname <- c("Hanna", "Peter", "Sophie", "Ulrike", "Thomas")
> alter <- c(21, 20, 22, 20, 19)
> semester <- c(3, 3, 5, 3, 1)
> studenten <- data.frame(vorname, alter, semester)
```

Hier legen wir zunächst drei Vektoren an, `vorname`, `alter` und `semester`. So weit dürfte Ihnen alles bekannt vorkommen. Im nächsten Schritt erzeugen wir einen Dataframe mithilfe der Funktion `data.frame`. Diese Funktion übernimmt die Vektoren, die Sie zu einem Dataframe verbinden wollen, als Argumente und gibt als Wert einen Dataframe zurück, in unserem Beispiel den Dataframe `studenten`. Mit einer einfachen Anweisung haben wir also die drei Variablen zu einem einzigen Datensatz »zusammenmontiert«. Achten Sie auf den Punkt im Funktionsnamen! Wie Sie zuvor schon gelernt haben, ist der Punkt ein gültiges Zeichen im Namen von Variablen, und Gleiches gilt für Funktionen. Der Punkt hat, anders als in anderen künstlichen Computersprachen, keine besondere Bedeutung. Es ist also mitnichten so, dass die Funktion `data.frame` »aus zwei Teilen besteht« oder Ähnliches. Viel einfacher: Die Funktion heißt einfach `data.frame`. Punkt.

Ein häufiger Fehler beim Aufbau von Dataframes aus einzelnen Vektoren besteht darin, dass nicht alle Vektoren die gleiche Länge besitzen. Betrachten wir das folgende leicht abgewandelte Beispiel von oben:

```
> vorname <- c("Hanna", "Peter", "Sophie", "Ulrike", "Thomas")
> alter <- c(21, 20, 22, 20)
> semester <- c(3, 3, 5, 3, 1)
> studenten <- data.frame(vorname, alter, semester)
Error in data.frame(vorname, alter, semester) :
  Argumente implizieren unterschiedliche Anzahl Zeilen: 5, 4
```

R weist Sie darauf, dass Ihr Datensatz nicht in allen Variablen die gleiche Zahl von Zeilen hat. Und in der Tat haben wir, wie Sie leicht sehen können, beim Vektor

alter das letzte Element gelöscht, er ist im Vergleich zum ursprünglichen Beispiel oben also um ein Element kürzer als die anderen beiden Vektoren. Aus Vektoren unterschiedlicher Länge kann natürlich kein sinnvoller Datensatz aufgebaut werden. In unserem Beispiel hätte Thomas kein Alter. Könnte es nicht aber sein, dass wir das Alter von Thomas nicht kennen und dass deshalb der Vektor alter kürzer ist als die anderen beiden Variablen? Selbstverständlich könnte Thomas eitel sein und uns sein Alter nicht genannt haben, aber in diesem Fall müsste der Vektor alter an der Stelle, an der Thomas' Alter erfasst ist, ein Missing (NA) haben. Sie erinnern sich, dass fehlende Daten in R mit einem Missing dargestellt werden. Stünde bei Thomas ein Missing, hätte der Vektor alter wieder die gleiche Länge wie die beiden anderen Vektoren, und wir könnten ohne Probleme einen Datensatz aus diesen drei Variablen aufbauen.

Leider sagt uns R nicht, welche Variable die zu kurz geratene ist. Sie müssten dies im Zweifel mit den Funktionen length() oder NROW() ermitteln, wie Sie es bereits gelernt haben.

Wir haben in den Beispielen oben Datensätze aus einzelnen Vektoren zusammengebaut. Von Zeit zu Zeit werden Sie tatsächlich auch einmal einen Datensatz aus einzelnen Vektoren erzeugen müssen. Meistens aber werden Sie Ihre Daten natürlich aus einer externen Quelle einlesen. Wie das genau funktioniert, werden wir uns im folgenden Abschnitt anschauen. Dann beginnen wir auch die Arbeit mit unserem Beispieldatensatz.

Wie bei jedem anderen Variablentyp in R können Sie sich auch bei Dataframes den Inhalt der Variablen dadurch anzeigen lassen, dass Sie deren Namen als Anweisung in die R-Konsole eingeben:

```
> studenten
  vorname alter semester
1   Hanna    21        3
2   Peter    20        3
3  Sophie    22        5
4  Ulrike    20        3
5  Thomas    19        1
```

Hier sehen Sie, dass wir nun einen sauberen Datensatz haben, der aus den drei Ausgangsvariablen besteht. R gibt Ihnen – wie jeden anderen Output auch – den Datensatz direkt in der Konsole aus. Das kann bei großen Datensätzen wie unserem Beispieldatensatz aber etwas unpraktisch sein. Wenn Sie viele Variablen haben, können diese nicht alle nebeneinander in der Konsole dargestellt werden. Auch ist wiederholtes Hoch- und Runterscrollen des Konsoleninhalts nicht gerade bequem, wenn Sie sich den Datensatz, nachdem Sie bereits in der Konsole weitergearbeitet haben, noch mal anschauen möchten. Daher kennt R eine alternative Möglichkeit, sich Datensätze anzeigen zu lassen. Der Befehl

```
> View(studenten)
```

öffnet ein Extrafenster, in dem Ihr Datensatz übersichtlich als Tabelle dargestellt wird. Das funktioniert sowohl in RGui als auch in RStudio (Abbildung 3-1). Allerdings ist die Funktion in RStudio etwas komfortabler ausgestaltet. So kann man in RStudio in der angezeigten Tabelle die Variablen auch noch filtern.

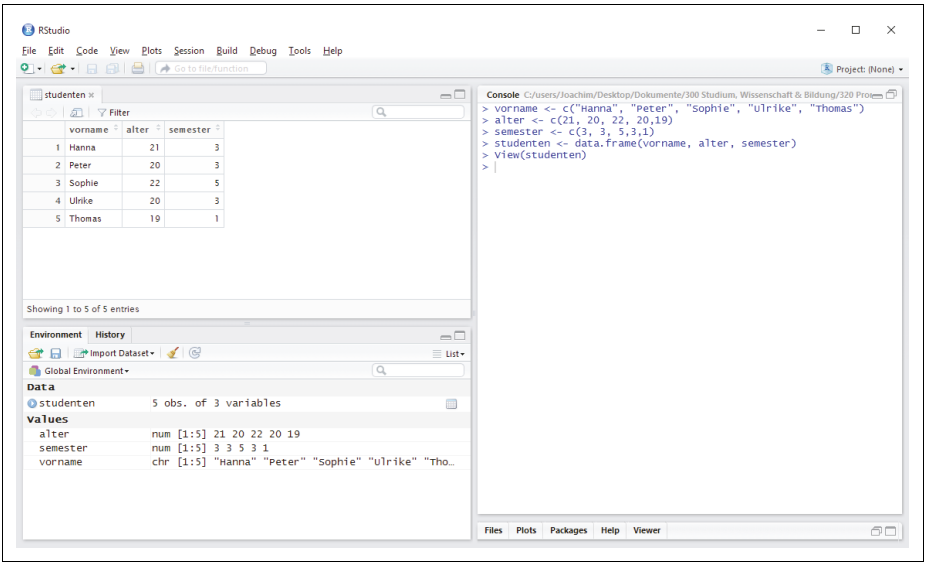


Abbildung 3-1: Datenfenster in RStudio

Beachten Sie beim Einsatz von `View`, dass diese Funktion zur seltenen Spezies jener R-Funktionen gehört, die mit großem Anfangsbuchstaben geschrieben werden. Schreiben Sie das »v« hingegen klein, wird R nicht verstehen, was Sie wollen.

Wir haben `View` dazu verwendet, Datensätze zu betrachten. Die Funktion `View` können Sie aber genauso gut dazu nutzen, sich einzelne Vektoren anzeigen zu lassen. Das ist gerade dann hilfreich, wenn ein Vektor sehr lang ist. Stellen Sie sich vor, Sie haben eine Vektor `n`, der einfach die natürlichen Zahlen von 1 bis 70 enthält.

Möchten Sie sich dessen Inhalt nun wie gewohnt durch Eingabe seines Namens in die R-Konsole anzeigen lassen, gibt R Ihnen folgenden Output:

```
> n
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
[21] 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

R stellt also die Elemente des Vektors einfach in einer Zeile hintereinander dar. Wenn eine Zeile nicht ausreicht, bricht R sie um und schreibt zu Beginn der Zeile die Nummer des ersten Elements der neuen Zeile in eckige Klammern. Das mag zwar bei der Navigation durch die Elemente des Vektors etwas helfen (in unserem Beispiel mit der Zahlenfolge natürlich ohnehin kein großes Problem), aber stellen Sie sich vor, Sie wollten aus einem solchen Vektor schnell mal eben das 33. Ele-

ment herausfinden. Dann beginnt das große Zählen. Wenn Sie sich den Vektor dagegen mit `View` anschauen, zeigt Ihnen R den Vektor senkrecht und vor jedem Element dessen Nummer, das heißt die Position innerhalb des Vektors, an. Das 33. Element wäre damit schnell gefunden.

Nun wollen Sie Ihren Datensatz natürlich nicht nur anschauen, sondern auch mit ihm arbeiten. Dazu müssen Sie häufig auf einzelne Variablen (also Spalten) aus dem Datensatz zugreifen. Wenn wir zum Beispiel später Regressionsmodelle bauen, müssen wir R mitteilen, welche die unabhängigen (die erklärenden) Variablen sind und welche die abhängige (die erklärte) Variable ist.

Angenommen, Sie wollten sich aus Ihrem Datensatz die Variable `alter` betrachten, ohne zugleich den gesamten Datensatz anzuschauen. Irgendwie müssen Sie R also mitteilen, dass Sie nur auf die Variable `alter` innerhalb des Datensatzes `studenten` zugreifen wollen. Und das machen Sie so:

```
> View(studenten$alter)
```

Das Dollarzeichen trennt dabei den Datensatz vom Namen der Variablen. Sie können also diesen Codeausdruck lesen als »zeige mir die Variable `alter` im Datensatz `studenten` an«. Damit ist auch klar, dass das Dollarzeichen kein zulässiges Zeichen in Variablennamen sein kann. Hätten Sie eine Variable `aktueller$kurs`, würde R denken, Sie wollten auf die Variable `kurs` im Datensatz `aktueller` zugreifen. Da weder ein Datensatz noch ein Vektor solchen Namens existiert, würden Sie eine Fehlermeldung erhalten.

`studenten$alter` ist also ein Vektor, und Sie können mit ihm alles tun, was Sie mit einem Vektor in R anstellen können. Insbesondere können Sie `studenten$alter` einer Funktion übergeben, die explizit einen Vektor als Argument erwartet. Wir haben schon einmal zuvor die Funktion `mean` angewendet, die den Mittelwert eines Vektors berechnet. Um das Durchschnittsalter der Studenten in unserem Datensatz zu ermitteln, können Sie also wie folgt vorgehen:

```
> mean(studenten$alter)
[1] 20.4
```

Wie Sie zuvor bereits gesehen haben, können Sie auf die Elemente von Vektoren zugreifen, indem Sie den Index des Elements, also die Position des Elements innerhalb des Vektors, in eckige Klammern schreiben. Da `studenten$alter` ein ganz normaler Vektor ist, geht das natürlich auch hier. Sollten Sie also feststellen, dass die Ulrike in unserem kleinen Datensatz doch nicht 20, sondern bereits 21 Jahre alt ist, können Sie das leicht korrigieren:

```
> studenten$alter[4]<-21
```

Da Ulrikes Beobachtung an der vierten Position im Datensatz steht, ändert diese Zuweisung die Variable `alter` in Ulrikes Datensatzzeile auf 21 Jahre.

Wir können bei der Indizierung unseres Datensatzes sogar noch einen Schritt weitergehen: Im Gegensatz zu den eindimensionalen Vektoren haben wir ja mit dem

Dataframe ein zweidimensionales Gebilde vor uns. Die Variablen stehen in den Spalten und die einzelnen Einträge des Datensatzes, die Beobachtungen, in den Zeilen. Dementsprechend können Sie nun auch zweidimensional indizieren:

```
> studenten[4,2]
[1] 21
```

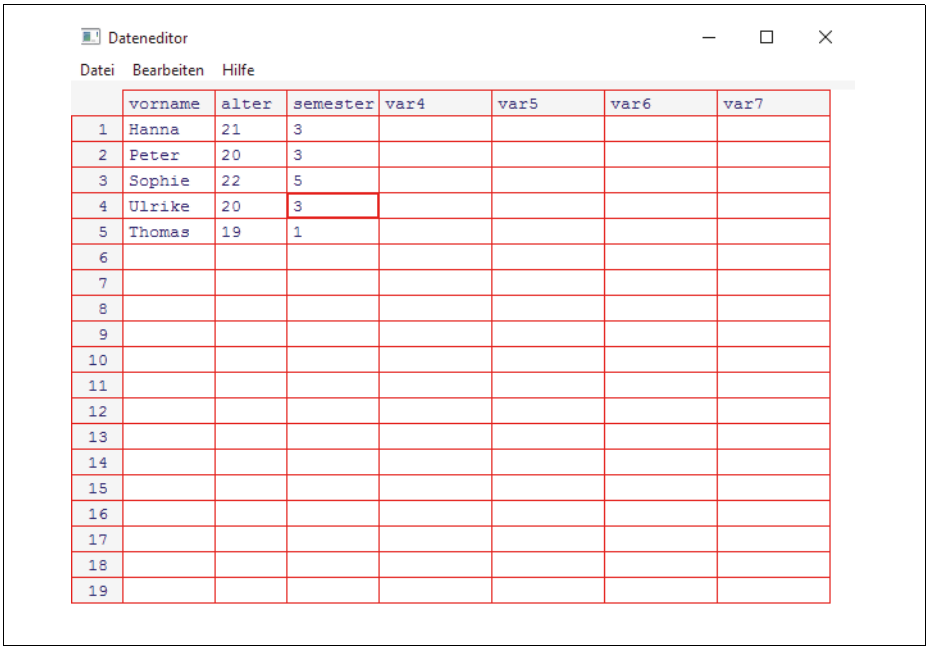
Sie sehen sofort den Unterschied zu der Indizierung aus dem vorangegangenen Beispiel: Wir indizieren nicht mehr einen speziellen Vektor aus dem Datensatz, sondern den Datensatz selbst. Dazu benötigen wir einen weiteren Index, denn jetzt müssen wir R nicht nur sagen, auf welche Zeile, auf welche Behachtung im Datensatz wir zugreifen wollen, sondern auch, auf welche Spalte, also auf welche Variable. Diese beiden Indizes stehen in den eckigen Klammern. Getreu der Ihnen vielleicht aus der Schule oder aus Lineare-Algebra-Vorlesungen bekannten Eselsbrücke »Zeile zuerst, Spalte später« bezeichnet der erste Index die angesprochene Zeile, der zweite die Spalte, also die Variable. Die Information, die wir aus dem Datensatz abfragen wollen, steht also in der vierten Zeile (das ist Ulrikes Beobachtung) und in der zweiten Spalte (das ist die Variable `alter`).

Im Sinne der Lesbarkeit Ihres R-Codes ist es empfehlenswert, wann immer möglich mit einer Notation der Form `studenten$alter[4]` zu arbeiten, denn so wissen Sie sofort, auf welche Variable Ihres Datensatzes hier zugegriffen wird. Dennoch gibt es Anwendungsfälle, in denen die zuletzt gesehene zweidimensionale Indizierung große Vorteile birgt, insbesondere wenn Sie einen Datensatz in einem R-Skript systematisch mit einer Schleife durchlaufen wollen. Schleifen schauen wir uns aber erst in Kapitel 9 etwas genauer an.

Sie haben eben gesehen, wie Sie durch eine Zuweisung Ihren Datensatz verändern können. Auf den ersten Blick ist die Arbeit mit solchen Zuweisungen natürlich etwas mühselig, vor allem wenn Sie mehrere Änderungen vornehmen wollen. Wäre es nicht viel schöner, Sie könnten die Daten ähnlich wie in einer Tabellenkalkulation bearbeiten? R bietet mit der Funktion `edit` tatsächlich eine Möglichkeit, Daten in einem Spreadsheet zu bearbeiten. Die Zeile

```
> studenten<-edit(studenten)
```

öffnet einen Dateneditor, wie Sie ihn im Fall des Aufrufs aus RStudio heraus in Abbildung 3-2 sehen. Die Funktion `edit` gibt den nach dem Bearbeiten geänderten Datensatz zurück. Um ihren Rückgabewert »aufzufangen«, weisen Sie ihn einer Variablen zu, in diesem Fall unserem ursprünglichen Datensatz. Der wird dadurch mit dem geänderten Datensatz überschrieben. Beachten Sie, dass ohne diese Zuweisung (bei einem auf `edit(studenten)` beschränkten Aufruf) die Änderungen, die Sie am Datensatz vornehmen, zwar nach dem Schließen des Editorfensters in der Konsole angezeigt werden, aber für die weitere Verwendung verloren sind, weil sie nicht in einer Variablen gesichert worden sind. Der ursprüngliche Datensatz bliebe so unverändert.



	vorname	alter	semester	var4	var5	var6	var7
1	Hanna	21	3				
2	Peter	20	3				
3	Sophie	22	5				
4	Ulrike	20	3				
5	Thomas	19	1				
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							

Abbildung 3-2: Dateneditor in RStudio

Was zunächst wie eine bequeme Funktionalität aussieht, die den Umgang mit Daten in R leichter macht, birgt gleichzeitig eine immense Gefahr: Änderungen, die Sie hier vornehmen, sind nicht dokumentiert und dementsprechend später sehr viel schwerer nachvollziehbar. Es ist deshalb sehr zu empfehlen, möglichst wenig mit dem Dateneditor zu arbeiten und stattdessen Änderungen an den Daten in abgespeicherten R-Skripten vorzunehmen, sodass die Änderungen dokumentiert und reproduzierbar sind.

Manchmal werden Sie nicht nur die Werte einzelner Variablen in einzelnen Beobachtungen Ihres Datensatzes ändern, sondern gleich eine ganze Variable hinzufügen wollen, die Sie erst generieren, nachdem Sie den Datensatz bereits in R geladen haben. So könnten Sie sich im Beispiel von oben besonders für jene Studenten interessieren, die im dritten Semester sind. Um nun diese Datensätze zu »markieren«, führen Sie eine weitere Variable `drittes.semester` ein:

```
> studenten$drittes.semester <- studenten$semester == 3
```

Danach sieht Ihr Datensatz so aus:

```
> studenten
  vorname alter semester drittes.semester
1   Hanna    21        3             TRUE
2   Peter    20        3             TRUE
3  Sophie    22        5            FALSE
4  Ulrike    20        3             TRUE
5  Thomas    19        1            FALSE
```

Die Anweisung ist eine Zuweisung an eine bis jetzt noch nicht existierende Variable `drittes.semester` in unserem Datensatz `studenten`. Sie hatten bereits zuvor gesehen, dass in R Variablen dadurch generiert werden, dass man ihnen einfach einen Wert zuweist. Genau das tun wir hier. Nur ist die Variable dieses Mal Teil eines Datensatzes.

Der Wert der Variablen ist `studenten$semester==3`. Hier wird also, wie wir es ebenfalls schon zuvor gesehen haben, ein Vergleich der Variablen `studenten$semester` mit dem Wert `3` durchgeführt. Das Ergebnis dieses Vergleichs ist ein Vektor vom Typ `logical`, wie sie mit

```
> class(studenten$drittes.semester)
[1] "logical"
```

leicht überprüfen können. Diese neue Variable hat in all jenen Beobachtungen den Wert `TRUE`, in denen der Wert von `semester` gleich `3` ist, was Sie an der obigen Ausgabe des modifizierten Datensatzes leicht nachvollziehen können. Die Anweisung `studenten$drittes.semester <- c(studenten$semester==3)`, in der Sie mithilfe der Funktion `c` einen neuen Vektor erzeugen, hätte im Übrigen den gleichen Effekt. Allerdings erkennt R hier sofort, dass Sie einen neuen Vektor anlegen möchten, weshalb Sie auf den Einsatz von `c` auch verzichten können.

Irgendwann, wenn Sie mit der gerade angelegten Variablen nicht mehr arbeiten wollen, möchten Sie sie vielleicht aus dem Datensatz entfernen. Wie Sie zuvor schon gelernt haben, ist das Löschen von Variablen zwar nicht unbedingt nötig, da sie in aller Regel die Grenzen, die durch den verfügbaren Speicher gesetzt sind, nicht erreichen werden. Doch trägt natürlich das Löschen von nicht mehr benötigten Variablen zur Übersichtlichkeit Ihres Datensatzes bei. Unsere Variable `drittes.semester` löschen Sie wie folgt:

```
> studenten$drittes.semester <- NULL
```

Durch Anzeigen des Datensatzes können Sie sich leicht davon überzeugen, dass die Variable »verschwunden« ist. Erreicht wurde das durch die Zuweisung des Werts `NULL`. Genauso wie `TRUE`, `FALSE` und `NA` ist auch `NULL` ein spezieller Wert in R. `NULL` repräsentiert nicht etwa den Wert `0`, denn ansonsten hätten wir mit der obigen Anweisung die Variable `drittes.semester` nicht gelöscht, sondern ihr einfach den Wert `0` zugewiesen (den sie dann in allen fünf Beobachtungen unseres Datensatzes hätte) und damit auch zugleich den Typ der Variablen in `numeric` geändert, weil die Variable statt eines Wahrheitswerts nun plötzlich eine Zahl aufnehmen würde. `NULL` meint auch nicht »Missing«, denn sonst hätten wir die Variable `drittes.semester` sozusagen »geleert«, indem wir alle ihre Elemente als nicht verfügbar (`NA`) markiert hätten, ohne aber die Existenz der Variablen an sich anzutasten. Vielmehr bedeutet `NULL` einfach »nicht existent«, und genau in diesen Zustand haben wir unsere Variable `drittes.semester` überführt. Beachten Sie, dass die nach dem bisher Gelernten naheliegende Anweisung zu einer Fehlermeldung führt:

```
> rm(studenten$drittes.semester)
Error in rm(studenten$drittes.semester) :
  ... muss Namen oder Zeichenketten enthalten
```

Nachdem Sie nun mit den Dataframes das zentrale Konzept zur Arbeit mit Datensätzen in R kennengelernt haben, sind wir bereit, endlich einen »echten« Datensatz, nämlich unseren Beispieldatensatz, einzulesen und mit der Arbeit zu beginnen.

Einlesen von Daten nach R

Bisher haben Sie einiges darüber gelernt, wie R Daten speichert. Im Besonderen haben Sie die unterschiedlichen Datentypen sowie die beiden grundlegenden Datenkonstrukte kennengelernt, mit denen Sie in R tagtäglich zu tun haben werden – Vektoren und Dataframes. Dataframes, die im Zentrum der Arbeit mit *ganzen Datensätzen* (im Unterschied zu nur *einzelnen Variablen*, die in Vektoren abgelegt werden) stehen, haben wir bislang stets aus Vektoren zusammengebaut, die wir in R erzeugt und befüllt haben. In der praktischen Arbeit werden Sie aber typischerweise Daten »von außerhalb« nach R laden. Genau darum geht es in diesem Abschnitt.

Damit beginnen wir zugleich die Arbeit mit unserem Beispieldatensatz, den wir uns zunächst etwas genauer anschauen wollen und dessen Analyse Sie durch den Rest des Buchs begleiten wird. Alle R-Skripte, die wir von nun an verwenden, finden Sie auch im Internet, sodass Sie sie nicht abtippen müssen, wenn Sie die Beispiele nachvollziehen wollen.

Der Beispieldatensatz

Nichts ist trockener als Statistik, die nur theoretisch betrieben wird. Nichts ist unanschaulicher als abstrakte Erklärungen zu einem Statistikprogramm, das man nicht in Aktion sieht. Keine Sorge, beides wird Ihnen mit diesem Buch nicht widerfahren. Im Gegenteil: Sie werden R in den folgenden Kapiteln sehr praxisnah kennenlernen. Und deshalb arbeiten wir natürlich auch mit realen Daten und versuchen, mit diesen Daten beispielhaft eine echte wissenschaftliche Fragestellung empirisch zu beantworten.

Diese Fragestellung, die Sie in vielen Beispielen durch das ganze Buch hinweg begleiten wird, ist eine ökonomische, und zwar die Frage nach den Faktoren, die die Höhe der Staatsausgaben bestimmen. Warum geben einige Länder einen höheren Anteil ihrer Wirtschaftskraft für öffentliche Leistungen aus als andere? Diese Frage ist nicht nur deshalb relevant, weil vom Umfang der Staatsausgaben natürlich der Finanzierungsbedarf des Staats abhängt und von diesem wiederum die Höhe der Steuerlast, die Sie und wir alle in unterschiedlicher Form zu tragen haben. Auch politisch ist die Frage interessant, ist doch der Umfang der Staatstä-

tigkeit, der sich in der Höhe der Staatsausgaben widerspiegelt, regelmäßig Gegenstand heftiger politischer Diskussionen, und das nicht nur in den USA, sondern auch hierzulande.

Um die Frage nach den Gründen für die unterschiedliche Höhe der Staatsausgaben beantworten zu können, wurde ein Datensatz zusammengestellt. Die Daten kommen (wie wahrscheinlich bei Ihrem Forschungsvorhaben auch) aus unterschiedlichen Quellen, in unserem Fall aus dem World Economic Outlook des Internationalen Währungsfonds, aus der Datenbank der Weltbank, der *Database on Political Institutions* (DBPI), die ebenfalls von der Weltbank bereitgestellt wird, sowie von Transparency International.

Teilweise sind die Daten ökonomischer Natur, teilweise benutzen wir aber auch Daten der Landes- und Bevökerungsstatistik sowie Daten, die politisch-institutionelle Gegebenheiten abbilden.

Im Einzelnen beinhaltet unser Datensatz folgende Daten:

- `ISO` und `COUNTRY`: Das Land, dessen Daten wir betrachten. `ISO` ist der Landescode der International Standards Organization, `COUNTRY` der Langname des Landes. Für Deutschland zum Beispiel ergibt sich so ein Pärchen aus »DEU« und »Germany« für diese beiden Variablen.
- `EXPEND`: Der Anteil der Staatsausgaben am Inlandsprodukt. Wir verwenden einen Fünfjahresdurchschnitt (2009 bis 2013), damit »zufällige« Schwankungen unsere Analyse nicht zu sehr verzerren.
- `GDPTOTAL`: Das Bruttoinlandsprodukt in Milliarden US-Dollar (ebenfalls ein Fünfjahresdurchschnitt).
- `GROWTH`: Das Wirtschaftswachstum als Durchschnitt der jährlichen Wachstumsraten in dem von uns betrachteten Fünfjahreszeitraum.
- `UNEMPLOY`: Die durchschnittliche Arbeitslosenquote zwischen 2009 und 2013.
- `INCOMEGROUP`: Die von der Weltbank vorgenommene Klassifizierung der Länder in »Low income«, »Lower middle income«, »Upper middle income«, »High income: OECD« und »High income: nonOECD«. Hier wird also die wirtschaftliche Leistungsfähigkeit bewertet, wobei bei den High-Income-Ländern noch mal unterschieden wird zwischen OECD-Ländern und solchen Ländern, die nicht zu den 34 Mitgliedern der Organisation für wirtschaftliche Zusammenarbeit und Entwicklung gehören.
- `POPULATION`: Die Anzahl der Einwohner (in Millionen).
- `POPSENIORS`: Der Anteil der Einwohner, die 65 Jahre alt sind oder älter.
- `POPCHILDREN`: Der Anteil der Einwohner, die jünger sind als 15 Jahre.
- `URBAN`: Der Urbanisierungsgrad, also der Anteil der Bevölkerung, der in Städten wohnt.
- `AREA`: Die Fläche des Landes in Quadratkilometern.

- `MILITARY`: Gibt an, ob an der Spitze der Exekutive des jeweiligen Landes ein Militäroffizier steht.
- `PLURALITY`: Gibt an, ob die Abgeordneten des nationalen Parlaments nach dem Mehrheitswahlrecht gewählt werden.
- `DEMOCRATIC`: Gibt an, ob das Land demokratisch ist. Dabei wird sowohl berücksichtigt, wie das nationale Parlament gewählt wird, als auch, wie die Spitze der Regierung bestellt wird.
- `WOMENPARL`: Der Anteil der weiblichen Abgeordneten im nationalen Parlament.
- `CPIVAL`: Der Index der wahrgenommenen Korruption (*Corruption Perception Index*, CPI) von Transparency International. Dieser bewegt sich innerhalb des Intervalls 0 bis 100, wobei 100 eine besonders niedrig wahrgenommene Korruption bedeutet. Der Einfachheit halber stammen diese Daten aus dem Jahr 2015, sind also ausnahmsweise kein Fünfjahresdurchschnitt).
- `CPIRANK`: Die Rangziffer, die das Land gemäß dem Index der wahrgenommenen Korruption einnimmt (eine höhere Platzierung bedeutet dabei ein geringeres Maß an wahrgenommener Korruption).

Der Datensatz deckt 189 Länder ab, jede Zeile im Datensatz repräsentiert ein Land. Das bedeutet aber keineswegs, dass für jedes Land jede der eben vorgestellten Variablen vorliegt. Der Datensatz ist also in gewissem Sinne »löchrig«. Mit anderen Worten: Er enthält Missings. Diese schränken die Zahl der Länder, die wir in unseren Analysen berücksichtigen können, mitunter ein.

Einige Tipps zur Arbeit mit Daten

Sie genießen hier den unschätzbar wertvollen Komfort, einen fertigen Datensatz vorzufinden. In aller Regel ist das natürlich anders, und Sie werden – teilweise mit großer Mühe – Ihren Datensatz selbst zusammenbauen müssen.

Insbesondere dann, wenn Sie Primärdaten erheben, das heißt nicht auf andere (sogenannte sekundäre) Datenquellen zurückgreifen (wie wir es hier in unserem Beispiel tun), sondern Ihre Daten selbst generieren (zum Beispiel indem Sie eine Befragung durchführen oder eine Messung vornehmen), werden Sie einen erheblichen Aufwand investieren müssen, um Ihre Daten zu erzeugen und vorzubereiten. Das kann, gerade im Fall der Primärerhebung, deutlich mehr Zeit in Anspruch nehmen als die eigentliche Datenanalyse. Aber auch wenn Sie ausschließlich mit Sekundärdaten arbeiten, müssen Sie immer mit einigem Aufwand rechnen, weil die Daten aus den unterschiedlichen Datenquellen möglicherweise in vollkommen unterschiedlichen Formaten vorliegen (verschiedene Dateiformate, unterschiedliche Spalten-/Zeilenstrukturen, uneinheitliche Dezimaltrennzeichen etc.). Berücksichtigen Sie also bei der Planung Ihres Forschungsvorhabens, dass bereits einige Arbeit zu leisten ist, bevor Sie richtig mit Ihren Analysen loslegen können!

Sie werden später lernen, wie Sie in R Daten aus unterschiedlichen Datensätzen kombinieren und weiterverarbeiten können. Auch wenn R eine Vielzahl nützlicher Funktionen bereitstellt, um mit Daten zu arbeiten, so ist doch eine pragmatische Herangehensweise an das wichtige Thema der Datenzusammenstellung und -aufbereitung zu empfehlen: Wann immer Sie mit anderen Werkzeugen als R schneller und einfacher zum Ziel kommen, setzen Sie diese Werkzeuge ein! Der Datensatz, der hier im Buch verwendet wird, wurde mit Microsoft Excel generiert. Es wäre zweifellos aufwendiger gewesen, alle Datensätze der unterschiedlichen Datenquellen nach R zu laden und sie dort zum Gesamtdatensatz zusammenzubauen. Trotzdem hätte der Einsatz von R auch einen großen Vorteil mit sich gebracht: Alle Schritte wären sauber dokumentiert worden und dadurch später leichter nachvollziehbar.

Wenn Sie also mit anderen Werkzeugen als R arbeiten, dokumentieren Sie nicht nur, woher Ihre Daten kommen (das sollten Sie in jedem Fall tun – auch wenn Sie mit R arbeiten), sondern auch, welche Bearbeitungsschritte Sie genau vollzogen haben. Sie müssen diese Dokumentation natürlich nicht zwingend in Fließtextform niederschreiben. Auch Excel-Formeln machen die Bearbeitung, die Sie Ihren Daten haben angedeihen lassen, sehr transparent. Wichtig ist nur, dass Sie (und gegebenenfalls andere) später noch nachvollziehen können, was mit den Daten geschehen ist.

Importieren der Daten

In diesem Abschnitt erfahren Sie,

- wie Sie Daten nach aus uncodierten (Text-)Dateien (zum Beispiel im CSV-Format) nach R importieren können,
- wie Sie die tpyischen Fallstricke beim Datenimport, die oft eine lange Fehlersuche nach sich ziehen, vermeiden,
- wie Sie Daten in codierten – also nicht von Menschen lesbaren – Formaten wie Excel und aus anderen Statistikpaketen (z. B. SPSS, Stata) importieren.

Folgende R-Funktionen werden in diesem Abschnitt behandelt:

- **`read.table`**(): Liest Daten aus einer uncodierten (Text-)Datei ein und bietet maximale Auswahlmöglichkeiten, um R die Struktur der einzulesenden Daten zu beschreiben.
- **`read.csv`**(), **`read.csv2`**(), **`read.delim`**(), **`read.delim2`**(): Lesen Daten aus einer uncodierten (Text-)Datei ein und setzen dabei unterschiedliche Standardwerte für die wichtigsten Argumente von `read.table`.
- **`read.spss`**(), **`read.dta`**(), **`read.ssd`**(), **`read.xls`**(): Lesen Daten aus den Statistikpaketen SPSS, Stata und SAS sowie aus Excel ein.
- **`names`**(): Zeigt die Namen der Variablen eines Dataframes an.

Die gute Nachricht gleich vorweg: Daten nach R einzulesen, ist ein sehr einfaches Unterfangen. Trotzdem gibt es einige mögliche Fallstricke, die einem problemlosen Datenimport im Weg stehen und die wir uns deshalb im Folgenden genauer anschauen werden.

Um zu verstehen, was beim Import nach R passiert, ist es ratsam, sich zunächst einmal den Datensatz anzuschauen. Wenn Sie Microsoft Excel installiert haben und doppelt auf die Datei unseres Beispieldatensatz klicken, wird sich aller Voraussicht nach Excel öffnen und Ihnen den Datensatz in einem hübsch aufbereiten Tabellenblatt mit klarer Zeilen- und Spaltenstruktur darstellen. Das liegt daran, dass Excel seinerseits beim Einlesen des Datensatzes die importierten Informationen in einer bestimmten Weise interpretiert. Tatsächlich aber sieht unser Datensatz – zumindest auf den ersten Blick – sehr viel weniger attraktiv aus. Öffnen Sie ihn mit einem Texteditor, wird sich Ihnen ein Bild wie das in Abbildung 3-3 bieten.

Abbildung 3-3: Beispieldatensatz im Texteditor

Was zunächst auffällt, ist, dass die erste Zeile deutlich anders aussieht als die übrigen. Sie erkennen in dieser Zeile sofort die Namen unserer Variablen, getrennt jeweils durch ein Semikolon. Unter dieser ersten Zeile, die als *Header* bezeichnet wird, weil sie den Tabellenkopf beschreibt, sehen Sie die eigentlichen Daten. Auch hier sind die einzelnen Spalten, das heißt die Variablen unseres Datensatzes, jeweils durch ein Semikolon getrennt. Solcherlei Art von Dateien, in denen Daten als Text in einer von Menschen lesbaren Form dargestellt werden und einzelne Spalten durch ein *Trennzeichen*, in unserem Beispiel ein Semikolon, getrennt sind, werden auch als CSV-Dateien (vom englischen Begriff *Comma Separated Values* – kommagetrennte Werte) bezeichnet, auch wenn das Trennzeichen tatsächlich gar kein Komma ist, wie der Name eigentlich nahelegt.

Beim Einlesen dieser CSV-Datei interpretiert Excel die Semikola als Spaltentrennzeichen und weiß daher genau, wo eine Spalte beginnt. Excel kann aber nicht nur

CSV-Datensätze lesen, sondern auch erzeugen. Das erlaubt Ihnen, beim Erstellen Ihres Datensatz auf die gut ausgebaute Funktionalität von Excel zurückzugreifen. Wenn Sie beim Speichern Ihrer Daten in Excel dann als Dateityp im *Speichern unter*-Dialog *CSV (Trennzeichen-getrennt) (*.csv)* auswählen, erzeugt Excel eine CSV-Datei für Sie.

So einfach das Erzeugen von CSV-Dateien erscheint, ein Wort der Vorsicht ist dennoch angebracht: Je nach Spracheinstellungen in Excel und Ihrem Betriebssystem entscheidet sich Excel beim Erzeugen der CSV-Datei für jeweils unterschiedliche Spaltentrennzeichen und Dezimaltrennzeichen. Letzteres ist in unserem Datensatz das Komma, eine englischsprachige Excel-Version mag hier einen Punkt setzen. Öffnen Sie deshalb vor dem Einlesen der Datei Ihre Daten einmal in einem Texteditor und vergewissern Sie sich, welches Spaltentrennzeichen und welches Dezimaltrennzeichen tatsächlich verwendet worden sind!

Das eigentliche Einlesen der Daten ist mithilfe der Funktion `read.table` sehr einfach:

```
> x <- read.table("daten_final.csv", header = TRUE, sep = ";",
  dec = ",")
```

Wenn Sie diese Anweisung in die Konsole eingeben, erhalten Sie einige Warnmeldungen. Bevor wir deren Ursachen genauer betrachten, schauen wir uns erst den Funktionsaufruf selbst etwas genauer an. Zunächst wird der Name der Datei angegeben, die eingelesen werden soll. Die Datei wird standardmäßig in dem zuvor mithilfe von `setwd` gesetzten Arbeitsverzeichnis gesucht. Liegt die Datei tatsächlich an einem anderen Ort, müssen Sie den vollständigen Verzeichnispfad zu Ihrer Datei angeben. Beachten Sie dabei, dass – ebenso wie bei `setwd` – in R-Pfadangaben stets der normale Schrägstrich (`/`) anstelle des sonst in der MS-DOS- und Windows-Welt üblichen Backslashs (`\`) verwendet werden muss.

Die übrigen Argumente der Funktion sind optional. Werden sie nicht explizit angegeben, verwendet R für sie festgelegte Standardwerte. Welche das sind, können Sie jederzeit in der Hilfe nachlesen, indem Sie die Anweisung `?read.table` ausführen. Trotz dieser Standardwerte ergibt eine explizite Angabe insbesondere des Spaltenseparators `sep` sowie des Dezimaltrennzeichens `dec` Sinn, nicht nur, weil die von Ihnen gewünschten Werte natürlich von den Standardwerten abweichen können (und es in unserem Beispiel auch tun!), sondern auch, weil hier beim Einlesen der Daten die meisten Fehlerquellen liegen. Daher schadet es nicht, auf den ersten Blick im R-Code sehen zu können, welches Spalten- und welches Dezimaltrennzeichen verwendet worden ist. Die Argumentangabe `header=TRUE` teilt R mit, dass in der ersten Zeile unserer Daten die Variablennamen stehen, also zum Beispiel `EXPEND` für die Staatsausgabenquote und `UNEMPLOY` für die Arbeitslosenquote. Auch dieses optionale Argument müssen wir hier explizit angeben, denn standardmäßig geht R davon aus, dass die Daten keinen Spaltenheader enthalten (`header=FALSE`).

Wie Sie beim Aufruf von `?read.table` sehen werden, besitzt die Funktion `read.table` eine ganze Reihe weiterer Argumente, mit denen Sie den Import der Daten noch genauer steuern können. Für unsere Zwecke reicht hier aber der oben dargestellte einfache Aufruf der Funktion.

Lesen wir nun also unsere Daten ein:

```
> x <- read.table("daten_final.csv", header = TRUE, sep = ";",
  dec = ",")
Warning messages:
1: In scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings,  :
  EOF within quoted string
2: In scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings,  :
  number of items read is not a multiple of the number of columns
```

Wir erhalten eine eher kryptisch anmutende Warnmeldung. Was ist passiert?

Zeichenkettenvariablen und Faktorvariablen können in CSV-Dateien in Anführungszeichen (engl. *Quote*) stehen. Das hat den Vorteil, dass dann innerhalb des durch die Anführungszeichen abgegrenzten Bereichs auch das Spaltentrennzeichen verwendet werden kann, ohne dass es R »aus dem Tritt« bringt, weil es fälschlicherweise eine neue Spalte anzeigt. Ein in einer Character-Variablen gespeicherter Text wie zum Beispiel der Kommentar `Interessante Studie; bitte informieren Sie mich über die Ergebnisse` würde, da wir als Spaltenseparator das Semikolon verwenden, von R in zwei Spalten getrennt werden. Um so etwas zu vermeiden, kann der Text in Anführungszeichen gesetzt werden. Der Funktion `read.table` können wir über das Argument `quote` sagen, welches Zeichen es als Anführungszeichen interpretieren soll. Auch dieses Argument ist ein optionales. Weil wir es beim Aufruf von `read.table` weggelassen haben, verwendet R einen Standardwert. Für das Argument `quote` gibt es sogar zwei Standardwerte: das doppelte Anführungszeichen (`"`) sowie das einfache Anführungszeichen (`'`). Leider kommt Letzteres in unserem Datensatz vor, und zwar im französischen Namen der Elfenbeinküste, Côte d'Ivoire. Dort findet allerdings nur ein einziges einfaches Anführungszeichen Verwendung, während R natürlich eine links und rechts durch Anführungszeichen begrenzte Zeichenkette erwartet. R kann damit nicht umgehen und gibt dementsprechend eine Warnung aus. Die Lösung des Problems ist zum Glück einfach: Wir sagen R, dass das Anführungszeichen (das in unserem Datensatz überhaupt nicht vorkommt) nicht das einfache, sondern das doppelte Anführungszeichen ist:

```
> x <- read.table("daten_final.csv", header = TRUE, sep = ";", quote = "\"",
  dec = ",", na.strings = "")
```

Da Zeichenketten in R selbst von Anführungszeichen umschlossen werden, müssen wir dabei im Argument `quote` einen Backslash (`\`) vor das Anführungszeichen setzen. Würden wir stattdessen `"""` schreiben, würde R denken, die ersten beiden Anführungszeichen begrenzten eine Zeichenkette, und das dritte Anführungszeichen würde eine weitere Zeichenkette öffnen, die dann aber nicht mehr geschlossen wird. Daher teilen wir R durch den vorangestellten Backslash mit, dass das

Anführungszeichen an dieser Stelle nicht dazu dient, eine Zeichenkette zu begrenzen, sondern gewissermaßen der Inhalt des Arguments quote ist. Außerdem fügen wir noch ein weiteres Argument hinzu: na.strings. Damit sagen wir R, dass Elemente einer Faktorvariablen, die leer sind, also keinen Text enthalten, als Missing betrachtet werden sollen. Machen wir das nicht, haben wir nachher in unserer Faktorvariablen INCOMEGROUP ein Level "". Das Argument na.strings nimmt einen ganzen Vektor entgegen, wir hätten also zum Beispiel mit na.strings=c("","-") dafür sorgen können, dass auch Werte, die nur einen Bindestrich enthalten, kein eigenes Faktorlevel erhalten, sondern zu Missings gemacht werden.

Lernen können Sie aus diesem kleinen und bewusst herbeigeführten Problem, dass, obwohl das Einlesen von Daten in R mit einer einzigen Anweisung zu bewerkstelligen ist, trotzdem unerwartet Probleme auftreten können, deren Ursache sich auf den ersten Blick nicht erschließt. Schauen Sie sich daher beim Einlesen Ihrer Daten den Datensatz stets genau an und überlegen Sie, ob die Spalten- und Dezimaltrennzeichen sowie das von R verwendete Quote-Zeichen Probleme bereiten könnten, weil diese Zeichen im Datensatz möglicherweise auch noch anders verwendet werden.

Ihnen wird aufgefallen sein, dass wir beim Aufruf der Funktion read.table die Namen der Funktionsargumente explizit angegeben haben, z. B. sep=";" für den Dezimalseparator. Das ist in R nicht zwingend erforderlich, solange Sie die Reihenfolge der Argumente einhalten.

Statt

```
> x <- read.table("daten_final.csv", header = TRUE, sep = ";",
  quote = "\"", dec = ",")
```

hätten wir also auch schreiben können:

```
> x <- read.table("daten_final.csv", TRUE, ";", "\"", ",")
```

Allerdings ist hier natürlich nicht mehr erkennbar, welcher Wert zu welchem Argument gehört. Zudem müssen Sie die vorgeschriebene Reihenfolge der Argumente genau einhalten, wenn Sie auf die Argumentnamen verzichten. Denn ohne Argumentnamen kann R nur auf Basis der Reihenfolge die Werte den unterschiedlichen Argumenten zuordnen. Schreiben Sie stattdessen die Argumentnamen aus, können Sie die Reihenfolge der Argumente vollkommen selbst bestimmen. Deshalb ist es empfehlenswert, die Argumentnamen zu verwenden. Ganz konsequent sind wir damit aber nicht gewesen: Das erste Argument, der Dateiname, wird nicht mit seinem Argumentnamen, file, übergeben. Das könnnte man natürlich machen (file="daten_final.csv"). Da allerdings an erster Stelle in der Argumentenliste immer das wichtigste Argument steht und üblicherweise recht klar ist, welches das ist, verzichten wir hier darauf.

Wenn Sie die R-Hilfe durchstöbern oder andere R-Bücher zurate ziehen, werden Sie vielleicht feststellen, dass neben read.table noch diverse andere Funktionen

existieren, um Daten einzulesen, zum Beispiel `read.csv`, `read.csv2`, `read.delim` und `read.delim2`. Diese sind im Grunde identisch mit der Funktion `read.table`, haben aber jeweils unterschiedliche Standardwerte für die wichtigsten optionalen Argumente `header`, `sep`, `quote` und `dec`. Diese Standardwerte sehen Sie in Tabelle 3-1. So geht beispielsweise die Funktion `read.csv2` von genau der Kombination von Argumenten aus, die wir auch oben in unserem Aufruf von `read.table` verwenden. Die Anweisung `read.csv("daten_final.csv")` hätte also die gleiche Wirkung wie unser `read.table` gehabt. Trotzdem ist es vorteilhaft, die Funktion `read.table` einzusetzen und die Argumente ausdrücklich aufzuschreiben, weil es die Lesbarkeit des R-Codes und damit die Fehlersuche und Problembehebung erleichtert. Dabei müssen Sie natürlich darauf achten, auch wirklich alle (relevanten) Argumente anzugeben, sonst verwendet R für diese Argumente die jeweiligen Standardwerte, und Sie haben keine zusätzliche Transparenz gewonnen.

Tabelle 3-1: Standardwerte der wichtigsten Argumente der von read.table abgeleiteten Funktionen

Funktion	Header (Tabellenkopf vorhanden)	sep (Spaltentrennzeichen)	quote (Zeichenkettenbegrenzer)	dec (Dezimalseparator)
read.csv	Ja	Komma	Anführungszeichen	Punkt
read.csv2	Ja	Semikolon	Anführungszeichen	Komma
read.delim	Ja	Tabulator	Anführungszeichen	Punkt
read.delim2	Ja	Tabulator	Anführungszeichen	Komma

Neben der genannten Funktion, die Dateien im Textformat (also solche, die man mit einem einfachen Texteditor öffnen und lesen kann) zu importieren, gibt es noch einige weitere Funktionen, die spezielle Dateiformate wie etwa die der Statistiksoftwarepakete SPSS (`read.spss`), Stata (`read.dta`) und SAS (`read.ssd`) einlesen können.

Wenn Sie Ihre Daten in Microsoft Excel bearbeiten und sie direkt aus einer Excel-Datei einlesen wollen, statt den Umweg über eine CSV-Datei gehen zu müssen, können Sie mit der Funktion `read.xls` aus dem Package `gdata` arbeiten, das Sie in diesem Fall zunächst mit `install.packages("gdata")` installieren und mit `library("gdata")` laden müssen. Die Funktion `read.xls` kommt, anders als der Name suggeriert, auch mit dem neuen XLSX-Format von Excel klar.

Wenn Ihre Excel-Datei beispielsweise *daten.xlsx* heißt und der Datensatz, den Sie einlesen wollen, auf dem Excel-Arbeitsblatt *Final* zu finden ist, können Sie die Daten mit folgender Anweisung einlesen:

```
> x <- read.xls("daten.xslx", sheet = "Final")
```

Die Funktion `read.xls` tut übrigens auch nichts anderes, als die Daten aus Excel in einer CSV-Datei zwischenzuspeichern und von dort aus nach R einzulesen.

Eine letzte Möglichkeit, irgendwie Daten nach R zu schaffen – die ich Ihnen zwar nicht empfehlen, aber dennoch nicht vorenthalten möchte –, ist, die Daten in die Zwischenablage zu kopieren (zum Beispiel wiederum aus Excel) und sie von dort in R einzufügen. So können Sie mit der Anweisung

```
> x <- readClipboard
```

die aktuell in der Zwischenablage befindlichen Daten als Dataframe `x` in R einfügen. Das mag hilfreich sein, um Daten schnell zum »Ausprobieren« nach R zu befördern. Wenn Sie aber dauerhaft mit einem stabilen Datensatz arbeiten wollen, sollten Sie diesen in einer Datei speichern und ihn dann aus dieser Datei in R einlesen. Das ist nicht nur sicherer, sondern vor allem auch gut in ein längeres R-Skript integrierbar, an dessen Anfang das Einlesen der Daten steht. Ansonsten müssten Sie immer sicherstellen, dass sich, wenn Sie Ihr R-Skript laufen lassen, Ihre Daten auch wirklich gerade in der Zwischenablage befinden.

Aber nun wirklich zurück zu unseren Daten. Es ist endlich geschafft! Der Datensatz ist eingelesen, und zwar in einen Dataframe mit dem Namen `x`. Wenn Sie nicht mit mehreren Datensätzen hantieren, wählen Sie am besten einen kurzen Namen für Ihren Dataframe. Auch wenn dieser Name dann nicht selbsterklärend ist, so spricht doch ein gewichtiges Argument für ihn: Sie werden ihn sehr oft tippen müssen.

Den eingelesenen Dataframe können Sie sich jetzt wie gewohnt durch Eingabe des Namens in R oder mit der Anweisung `View(x)` anzeigen lassen. Letztere hat den Vorteil, dass sie zu einer übersichtlichen tabellarischen Darstellung führt, die bei großen Datensätzen, vor allem solchen mit vielen Variablen, auf jeden Fall zu bevorzugen ist.

Wenn Sie nicht die gesamten Daten, sondern lediglich die Namen der Variablen, die im Datensatz enthalten sind, angezeigt bekommen möchten, verwenden Sie die Funktion `names`.

```
> names(x)
 [1] "ISO"          "COUNTRY"      "EXPEND"       "GDPTOTAL"
 [5] "GROWTH"       "UNEMPLOY"     "INCOMEGROUP"  "POPULATION"
 [9] "POPSENIORS"   "POPCHILDREN"  "URBAN"        "AREA"
[13] "MILITARY"     "PLURALITY"    "DEMOCRATIC"   "WOMENPARL"
[17] "CPIVAL"       "CPIRANK"
```