Statistik und ihre Anwendungen

Grundlagen der Datenanalyse mit R

Eine anwendungsorientierte Einführung

von
Daniel Wollschläger

1. Auflage

Springer 2012

Verlag C.H. Beck im Internet:
www.beck.de
ISBN 978 3 642 25799 5

Zu Inhaltsverzeichnis

Kapitel 2
Elementare Dateneingabe und -verarbeitung

Die folgenden Abschnitte sollen gleichzeitig die grundlegenden Datenstrukturen in R sowie Möglichkeiten zur deskriptiven Datenauswertung erläutern. Die Reihenfolge der Themen ist dabei so gewählt, dass die abwechselnd vorgestellten Datenstrukturen und darauf aufbauenden deskriptiven Methoden nach und nach an Komplexität gewinnen.

2.1 Vektoren

R ist eine vektorbasierte Sprache, ist also auf die Verarbeitung von in Vektoren angeordneten Daten ausgerichtet. Ein Vektor ist dabei lediglich eine Datenstruktur für eine sequenziell geordnete Menge einzelner Werte und nicht mit dem mathematischen Konzept eines Vektors zu verwechseln. Da sich empirische Daten einer Variable meist als eine linear anzuordnende Wertemenge betrachten lassen, sind Vektoren als Organisationsform gut für die Datenanalyse geeignet. Vektoren sind in R die einfachste Datenstruktur für Werte, d. h. auch jeder Skalar oder andere Einzelwert ist ein Vektor der Länge 1.

2.1.1 Vektoren erzeugen

Vektoren werden durch Funktionen erzeugt, die den Namen eines Datentyps tragen und als Argument die Anzahl der zu speichernden Elemente erwarten, also etwa `numeric(⟨Anzahl⟩)`. Die Elemente des Vektors werden hierbei auf eine Voreinstellung gesetzt, die vom Datentyp abhängt.[1]

[1] Ein leerer Vektor entsteht analog, z. B. durch `numeric(0)`. Ein Vektor kann höchstens `.Machine$integer.max` viele (derzeit $2^{31}-1$) Elemente enthalten. Mit `vector(mode="⟨Klasse⟩", n=⟨Länge⟩)` lassen sich beliebige Objekte der für `mode` genannten Klasse der Länge `n` erzeugen.

D. Wollschläger, *Grundlagen der Datenanalyse mit R, Statistik und ihre Anwendungen,*
DOI 10.1007/978-3-642-25800-8_2, © Springer-Verlag Berlin Heidelberg 2012

```
> numeric(4)
[1] 0 0 0 0
```

Als häufiger genutzte Alternative lassen sich Vektoren auch mit der Funktion `c(⟨Wert1⟩, ⟨Wert2⟩, ...)` erstellen (concatenate), die die Angabe der zu speichernden Werte benötigt. Ein das Alter von sechs Personen speichernder Vektor könnte damit so erstellt werden:

```
> (age <- c(18, 20, 30, 24, 23, 21))
[1] 18 20 30 24 23 21
```

Dabei werden die Werte in der angegebenen Reihenfolge gespeichert und intern mit fortlaufenden Indizes für ihre Position im Vektor versehen. Sollen bereits bestehende Vektoren zusammengefügt werden, ist ebenfalls `c()` zu nutzen, wobei statt eines einzelnen Wertes auch der Name eines bereits bestehenden Vektors angegeben werden kann.

```
> addAge       <- c(27, 21, 19)
> (ageNew      <- c(age, addAge))
[1] 18 30 30 25 23 21 27 21 19
```

Mit der Funktion `length(⟨Vektor⟩)` wird die Länge eines Vektors, d. h. die Anzahl der in ihm gespeicherten Elemente, erfragt.

```
> length(age)
[1] 6
```

Auch Zeichenketten können die Elemente eines Vektors ausmachen.

```
> (chars <- c("lorem", "ipsum", "dolor"))
[1] "lorem" "ipsum" "dolor"
```

2.1.2 Elemente auswählen und verändern

Um ein einzelnes Element eines Vektors abzurufen, wird seine Position im Vektor (sein Index) in eckigen Klammern, dem `[⟨Index⟩]` Operator, hinter dem Objektnamen angegeben.[2] Indizes beginnen bei 1 für die erste Position[3] und enden bei der Länge des Vektors. Werden größere Indizes verwendet, erfolgt als Ausgabe die für einen fehlenden Wert stehende Konstante `NA` (vgl. Abschn. 2.11).

```
> age[4]
[1] 24

> (ageLast <- age[length(age)])
[1] 21
```

[2] Für Hilfe zu diesem Thema vgl. `?Extract`. Auch der Index-Operator ist eine Funktion, kann also gleichermaßen in der Form `"["(⟨Vektor⟩, ⟨Index⟩)` verwendet werden (vgl. Abschn. 1.2.5, Fußnote 13).

[3] Dies mag selbstverständlich erscheinen, in anderen Sprachen wird jedoch oft der Index **0** für die erste Position und allgemein der Index $n - 1$ für die n-te Position verwendet.

```
> age[length(age) + 1]
[1] NA
```

Ein Vektor muss nicht unbedingt einem Objekt zugewiesen werden, um indiziert werden zu können, dies ist auch für unbenannte Vektoren möglich.

```
> c(11, 12, 13, 14)[2]
[1] 12
```

Mehrere Elemente eines Vektors lassen sich gleichzeitig abrufen, indem ihre Indizes in Form eines Indexvektors in die eckigen Klammern eingeschlossen werden. Dazu kann man zunächst einen eigenen Vektor erstellen, dessen Name dann in die eckigen Klammern geschrieben wird. Ebenfalls kann der Befehl zum Erzeugen eines Vektors direkt in die eckigen Klammern verschachtelt werden. Der Indexvektor kann auch länger als der indizierte Vektor sein, wenn einzelne Elemente mehrfach ausgegeben werden sollen. Das Weglassen eines Index mit ⟨`Vektor`⟩`[]` führt dazu, dass alle Elemente des Vektors ausgegeben werden.

```
> idx <- c(1, 2, 4)
> age[idx]
[1] 18 20 24

> age[c(3, 5, 6)]
[1] 30 23 21

> age[c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6)]
[1] 18 18 20 20 30 30 24 24 23 23 21 21
```

Beinhaltet der Indexvektor fehlende Werte (`NA`), erzeugt dies in der Ausgabe ebenfalls einen fehlenden Wert an der entsprechenden Stelle.

```
> age[c(4, NA, 1)]
[1] 25 NA 17
```

Wenn alle Elemente bis auf ein einzelnes abgerufen werden sollen, ist dies am einfachsten zu bewerkstelligen, indem der Index des nicht erwünschten Elements mit negativem Vorzeichen in die eckigen Klammern geschrieben wird.[4] Sollen mehrere Elemente nicht ausgegeben werden, verläuft der Aufruf analog zum Aufruf gewünschter Elemente, wobei mehrere Variationen mit dem negativen Vorzeichen möglich sind.

```
> age[-3]                  # alle Elemente bis auf das 3.
[1] 18 20 24 23 21

> age[c(-1, -2, -4)]       # alle Elemente bis auf das 1., 2. und 4.
[1] 30 23 21

> age[-c(1, 2, 4)]         # alle Elemente bis auf das 1., 2. und 4.
[1] 30 23 21

> age[-idx]                # alle Elemente bis auf die Indizes im Vektor idx
[1] 30 23 21
```

[4] Als Indizes dürfen in diesem Fall keine fehlenden Werte (`NA`) vorkommen, ebenso darf der Indexvektor nicht leer sein.

Die in einem Vektor gespeicherten Werte können nachträglich verändert werden. Dazu muss der Position des zu ändernden Wertes der neue Wert zugewiesen werden.

```
> age[4] <- 25; age
[1] 18 20 30 25 23 21
```

Das Verändern von mehreren Elementen gleichzeitig geschieht analog. Dazu lassen sich die Möglichkeiten zur Auswahl mehrerer Elementen nutzen und diesen in einem Arbeitsschritt passend viele neue Werte zuweisen. Dabei müssen die zugewiesenen Werte ebenfalls durch einen Vektor repräsentiert sein.

```
> age[idx] <- c(17, 30, 25); age
[1] 17 30 30 25 23 21
```

Um Vektoren zu verlängern, also mit neuen Elementen zu ergänzen, kann zum einen der `[⟨Index⟩]` Operator benutzt werden, wobei als Index nicht belegte Positionen angegeben werden.[5] Zum anderen kann auch hier die `c(⟨Wert1⟩, ⟨Wert2⟩,...)` Funktion Verwendung finden. Als Alternative steht die `append(⟨Vektor⟩, values=⟨Vektor⟩)` Funktion zur Verfügung, die an einen Vektor die Werte eines unter `values` genannten Vektors anhängt.

```
> charVec1 <- c("Z", "Y", "X")
> charVec1[c(4, 5, 6)] <- c("W", "V", "U"); charVec1
[1] "Z" "Y" "X" "W" "V" "U"

> (charVec2 <- c(charVec1, "T", "S", "R"))
[1] "Z" "Y" "X" "W" "V" "U" "T" "S" "R"

> (charVec3 <- append(charVec2, c("Q", "P", "O")))
[1] "Z" "Y" "X" "W" "V" "U" "T" "S" "R" "Q" "P" "O"
```

2.1.3 Datentypen in Vektoren

Vektoren können Werte unterschiedlicher Datentypen speichern, etwa `numeric`, wenn sie Zahlen beinhalten oder `character` im Fall von Zeichenketten. Letztere müssen dabei immer in Anführungszeichen eingegeben werden. Jeder Vektor kann aber nur einen Datentyp besitzen, alle seine Elemente haben also denselben Datentyp. Fügt man einem numerischen Vektor eine Zeichenkette hinzu, so werden seine numerischen Elemente automatisch in Zeichenketten umgewandelt,[6] was man an den hinzugekommenen Anführungszeichen erkennt und mit `mode(⟨Vektor⟩)` überprüfen kann.

[5] Bei der Verarbeitung sehr großer Datenmengen ist zu bedenken, dass die schrittweise Vergrößerung von Objekten aufgrund der dafür notwendigen internen Kopiervorgänge ineffizient ist. Objekte sollten deshalb bevorzugt bereits mit der Größe angelegt werden, die sie später benötigen.

[6] Allgemein gesprochen werden alle Elemente in den umfassendsten Datentyp umgewandelt, der notwendig ist, um alle Werte ohne Informationsverlust zu speichern (vgl. Abschn. 1.3.5).

```
> charVec4      <- "word"
> numVec        <- c(10, 20, 30)
> (combVec      <- c(charVec4, numVec))
[1] "word" "10" "20" "30"

> mode(combVec)
[1] "character"
```

Zwei aus Zeichen bestehende Vektoren sind in R bereits vordefiniert, `LETTERS` und `letters`, die jeweils alle Buchstaben A–Z bzw. a–z in alphabetischer Reihenfolge als Elemente besitzen.

```
> LETTERS[c(1, 2, 3)]            # Alphabet in Großbuchstaben
[1] "A" "B" "C"

> letters[c(4, 5, 6)]            # Alphabet in Kleinbuchstaben
[1] "d"  "e"  "f"
```

2.1.4 Elemente benennen

Es ist möglich, die Elemente eines Vektors bei seiner Erstellung zu benennen. Die Elemente können dann nicht nur über ihren Index, sondern auch über ihren in Anführungszeichen gesetzten Namen angesprochen werden.[7] In der Ausgabe wird der Name eines Elements in der über ihm stehenden Zeile mit aufgeführt.

```
> (namedVec1 <- c(elem1="first", elem2="second"))
elem1    elem2
"first"  "second"

> namedVec1["elem1"]
elem1
"first"
```

Elemente lassen sich über den Namen nur auswählen, nicht aber mittels ⟨`Vektor`⟩`[-"`⟨`Name`⟩`"]` ausschließen, hierfür bedarf es des numerischen Index. Auch im Nachhinein lassen sich Elemente benennen, bzw. vorhandene Benennungen ändern – beides geschieht mit der `names(`⟨`Vektor`⟩`)` Funktion.

```
> (namedVec2 <- c(val1=10, val2=-12, val3=33))
val1  val2  val3
  10   -12    33

> names(namedVec2)
[1] "val1"  "val2"  "val3"

> names(namedVec2) <- c("A", "B", "C"); namedVec2
 A    B   C
10  -12  33
```

[7] Namen werden als Attribut gespeichert und sind mit `attributes(`⟨`Vektor`⟩`)` sichtbar (vgl. Abschn. 1.3).

2.1.5 Elemente löschen

Elemente eines Vektors lassen sich nicht im eigentlichen Sinne löschen. Denselben Effekt kann man stattdessen über zwei mögliche Umwege erzielen. Zum einen kann ein bestehender Vektor mit einer Auswahl seiner eigenen Elemente überschrieben werden.

```
> vec <- c(10, 20, 30, 40, 50)
> vec <- vec[c(-4, -5)]; vec
[1] 10 20 30
```

Zum anderen kann ein bestehender Vektor über die `length()` Funktion verkürzt werden, indem ihm eine Länge zugewiesen wird, die kleiner als seine bestehende ist. Gelöscht werden dabei die überzähligen Elemente am Ende des Vektors.

```
> vec             <- c(1, 2, 3, 4, 5)
> length(vec)     <- 3; vec
[1] 1 2 3
```

2.2 Logische Operatoren

Verarbeitungsschritte mit logischen Vergleichen und Werten treten häufig bei der Auswahl von Teilmengen von Daten sowie bei der Recodierung von Datenwerten auf. Dies liegt vor allem an der Möglichkeit, in Vektoren und anderen Datenstrukturen gespeicherte Werte auch mit logischen Indexvektoren auszuwählen.

2.2.1 Logische Operatoren zum Vergleich von Vektoren

Vektoren werden oft mithilfe logischer Operatoren mit einem bestimmten Wert oder auch mit anderen Vektoren verglichen um zu prüfen, ob die Elemente gewisse Bedingungen erfüllen. Als Ergebnis der Prüfung wird ein logischer Vektor mit Wahrheitswerten ausgegeben, der die Resultate der elementweisen Anwendung des Operators beinhaltet.

Als Beispiel seien im Vektor `age` wieder die Daten von sechs Versuchspersonen (VPn) gespeichert. Zunächst sollen jene VPn identifiziert werden, die jünger als 24 Jahre sind. Dazu wird der `<` Operator verwendet, der als Ergebnis einen Vektor mit Wahrheitswerten liefert, der für jedes Element separat angibt, ob die Bedingung < 24 zutrifft. Andere Vergleichsoperatoren, wie gleich (`==`), ungleich (`!=`), etc. funktionieren analog.

```
> age <- c(17, 30, 30, 24, 23, 21)
> age < 24
[1] TRUE FALSE FALSE FALSE TRUE TRUE
```

Wenn zwei Vektoren miteinander logisch verglichen werden, wird der Operator immer auf ein zueinander gehörendes Wertepaar angewendet, also auf Werte, die sich an derselben Position in ihrem jeweiligen Vektor befinden.

```
> x <- c(2, 4, 8)
> y <- c(3, 4, 5)
> x == y
[1] FALSE TRUE FALSE

> x < y
[1] TRUE FALSE FALSE
```

Auch die Prüfung jedes Elements auf mehrere Kriterien ist möglich. Wenn zwei Kriterien gleichzeitig erfüllt sein sollen, wird `&` als Symbol für das logische UND verwendet. Wenn nur eines von zwei Kriterien erfüllt sein muss, ist das Symbol `|` für das logische, d. h. einschließende, ODER zu verwenden. Um sicherzustellen, dass R die zusammengehörenden Ausdrücke auch als Einheit erkennt, ist die Verwendung runder Klammern zu empfehlen.

```
> (age <= 20) | (age >= 30)              # Werte im Bereich bis 20 ODER ab 30?
[1] TRUE TRUE TRUE FALSE FALSE FALSE

> (age > 20) & (age < 30)               # Werte im Bereich zwischen 20 und 30?
[1] FALSE FALSE FALSE TRUE TRUE TRUE
```

UND und ODER dürfen bei zusammengesetzten Prüfungen nicht weggelassen werden: Während man mathematisch also eine Bedingung etwa als $0 \leq x \leq 10$ formulieren würde, müsste sie in R in Form von mit UND verbundenen Einzelprüfungen geschrieben werden, also wie oben als `(0 <= x) & (x <= 10)`.

Während die elementweise Prüfung von Vektoren den häufigsten Fall der Anwendung logischer Kriterien ausmacht, sind vor allem zur Fallunterscheidung (vgl. Abschn. 12.1.1) auch Prüfungen notwendig, die in Form eines einzelnen Wahrheitswertes eine summarische Auskunft darüber liefern, ob Kriterien erfüllt sind. Diese auch bei Anwendung auf Vektoren nur einen Wahrheitswert ergebenden Prüfungen lassen sich mit `&&` für das logische UND bzw. mit `||` für das logische ODER formulieren. Beide Vergleiche werten nur das jeweils erste Element aus, wenn Vektoren beteiligt sind.

```
> c(TRUE, FALSE, FALSE) && c(TRUE, TRUE, FALSE)
[1] TRUE

> c(FALSE, FALSE, TRUE) || c(FALSE, TRUE, FALSE)
[1] FALSE
```

Die `identical()` Funktion prüft zwei übergebene Vektoren summarisch auf Gleichheit und gibt nur dann das Ergebnis `TRUE` aus, wenn diese auch bzgl. ihrer internen Repräsentation exakt identisch sind (vgl. Abschn. 1.3.6).

```
> c(1, 2) == c(1L, 2L)
[1] TRUE TRUE

> identical(c(1, 2), c(1L, 2L))
[1] FALSE
```

Sollen Werte nur auf ungefähre Übereinstimmung geprüft werden, kann dies mit `all.equal()` geschehen (vgl. Abschn. 1.3.6). Dabei ist im Fall von zu vergleichenden Vektoren zu beachten, dass die Funktion keinen Vektor der Ergebnisse der elementweisen Einzelvergleiche ausgibt. Stattdessen liefert sie nur einen einzelnen Wert zurück, entweder `TRUE` im Fall der paarweisen Übereinstimmung aller Elemente oder das mittlere Abweichungsmaß im Fall der Ungleichheit. Um auch in letzterem Fall einen Wahrheitswert als Ausgabe zu erhalten, sollte die `isTRUE()` Funktion verwendet werden.

```
> x <- c(4, 5, 6)
> y <- c(4, 5, 6)
> z <- c(1, 2, 3)
> all.equal(x, y)
[1] TRUE

> all.equal(y, z)
[1] "Mean relative difference: 0.6"

> isTRUE(all.equal(y, z))
[1] FALSE
```

Bei der Prüfung von Elementen auf Kriterien kann mithilfe spezialisierter Funktionen summarisch analysiert werden, ob diese Kriterien zutreffen. Ob mindestens ein Element eines logischen Vektors den Wert `TRUE` besitzt, zeigt `any(⟨Vektor⟩)`, ob alle Elemente den Wert `TRUE` haben, gibt `all(⟨Vektor⟩)` an.[8]

```
> res <- age > 30
> any(res)
[1] FALSE

> any(age < 18)
[1] TRUE

> all(x == y)
[1] TRUE
```

Um zu zählen, auf wie viele Elemente eines Vektors ein Kriterium zutrifft, wird die Funktion `sum(⟨Vektor⟩)` verwendet, die alle Werte eines Vektors aufaddiert (vgl. Abschn. 2.7.1).

```
> res <- age < 24
> sum(res)
[1] 3
```

Alternativ kann verschachtelt in `length()` die `which(⟨Vektor⟩)` Funktion genutzt werden, die die Indizes der Elemente mit dem Wert `TRUE` ausgibt (vgl. Abschn. 2.2.2).

[8] Dabei erzeugt `all(numeric(0))` das Ergebnis `TRUE`, da die Aussage „alle Elemente des leeren Vektors sind WAHR“ logisch WAHR ist – schließlich lässt sich kein Gegenbeispiel in Form eines Elements finden, das FALSCH wäre. Dagegen erzeugt `any(numeric(0))` das Ergebnis `FALSE`, da in einem leeren Vektor nicht mindestens ein Element existiert, das WAHR ist.

```
> which(age < 24)
[1] 1 5 6

> length(which(age < 24))
[1] 3
```

2.2.2 Logische Indexvektoren

Vektoren von Wahrheitswerten können wie numerische Indexvektoren zur Indizierung anderer Vektoren benutzt werden. Diese Art zu indizieren kann z. B. zur Auswahl von Teilstichproben genutzt werden, die durch bestimmte Merkmale definiert sind. Hat ein Element des logischen Indexvektors den Wert `TRUE`, so wird das sich an dieser Position befindliche Element des indizierten Vektors ausgegeben. Hat der logische Indexvektor an einer Stelle den Wert `FALSE`, so wird das zugehörige Element des indizierten Vektors ausgelassen.

```
> age[c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE)]
[1] 17 30 24 21
```

Wie numerische können auch logische Indizes zunächst in einem Vektor gespeichert werden, mit dem die Indizierung dann später geschieht. Statt der Erstellung eines separaten logischen Indexvektors, z. B. als Ergebnis einer Überprüfung von Bedingungen, kann der Schritt aber auch übersprungen und der logische Ausdruck direkt innerhalb des `[⟨Index⟩]` Operators benutzt werden. Dabei ist jedoch abzuwägen, ob der Übersichtlichkeit und Nachvollziehbarkeit der Befehle mit einer separaten Erstellung von Indexvektoren besser gedient ist.

```
> (idx <- (age <= 20) | (age >= 30))        # Werte im Bereich bis 20 ODER ab 30?
[1] TRUE TRUE TRUE FALSE FALSE FALSE

> age[idx]
[1] 17 30 30

> age[(age >= 30) | (age <= 20)]
[1] 17 30 30
```

Bei logischen Indexvektoren kann anders als bei numerischen Indexvektoren die von R automatisch vorgenommene zyklische Verlängerung greifen (vgl. Abschn. 2.5.4.1): Logische Indexvektoren mit weniger Elementen als jene des indizierten Vektors werden durch zyklische Wiederholung soweit verlängert, dass sie mindestens die Länge des indizierten Vektors erreichen.

```
> age[c(TRUE, FALSE)]           # logischer Indexvektor kürzer als age
[1] 18 30 23

# durch zyklische Verlängerung von c(TRUE, FALSE) äquivalent zu
> age[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)]
[1] 18 30 23
```

Logische Indexvektoren bergen den Nachteil, dass sie zu Problemen führen können, wenn der zu prüfende Vektor fehlende Werte enthält. Überall dort, wo dieser NA ist, wird i. d. R. auch das Ergebnis eines logischen Vergleichs NA sein, d. h. der resultierende logische Indexvektor enthält seinerseits fehlende Werte (vgl. Abschn. 2.11.3, Fußnote 42).

```
> vecNA      <- c(-3, 2, 0, NA, -7, 5)          # Vektor mit fehlendem Wert
> (logIdx    <- vecNA > 0)                      # prüfe auf Werte größer 0
[1] FALSE TRUE FALSE NA FALSE TRUE
```

Enthält ein Indexvektor einen fehlenden Wert, erzeugt er beim Indizieren eines anderen Vektors an dieser Stelle ebenfalls ein NA in der Ausgabe (vgl. Abschn. 2.1.2). Dies führt dazu, dass sich der Indexvektor nicht mehr dazu eignet, ausschließlich die Werte auszugeben, die eine bestimmte Bedingung erfüllen.

```
> vecNA[logIdx]              # Auswahl mit logIdx erzeugt NA
[1] 2 NA 5
```

In Situationen, in denen fehlende Werte möglich sind, ist deshalb ein anderes Vorgehen sinnvoller: Statt eines logischen Indexvektors sollten die numerischen Indizes derjenigen Elemente zum Indizieren verwendet werden, die die geprüfte Bedingung erfüllen, an deren Position der logische Vektor also den Wert TRUE besitzt. Logische in numerische Indizes wandelt in diesem Sinne die which(⟨Vektor⟩) Funktion um, die die Indizes der TRUE Werte zurückgibt.[9]

```
> (numIdx <- which(logIdx))          # numer. Indizes der TRUE Werte
[1] 2 6

> vecNA[numIdx]                      # korrekte Auswahl
[1] 2 5
```

2.3 Mengen

Werden Vektoren als Wertemengen im mathematischen Sinn betrachtet, ist zu beachten, dass die Elemente einer Menge nicht geordnet sind und mehrfach vorkommende Elemente wie ein einzelnes behandelt werden, so ist z. B. die Menge $\{1, 1, 2, 2\}$ gleich der Menge $\{2, 1\}$.[10]

[9] Umgekehrt lassen sich auch die in ⟨Indexvektor⟩ gespeicherten numerischen Indizes für ⟨Vektor⟩ in logische verwandeln: seq(along=⟨Vektor⟩) %in% ⟨Indexvektor⟩ (vgl. Abschn. 2.3.2, 2.4.1).

[10] Das Paket sets (Meyer & Hornik, 2009) stellt eine eigene Klasse zur Repräsentation von Mengen zur Verfügung und implementiert auch einige hier nicht behandelte Mengenoperationen – etwa das Bilden der Potenzmenge.

2.3.1 Duplizierte Werte behandeln

Die `duplicated(⟨Vektor⟩)` Funktion gibt für jedes Element eines Vektors an, ob der Wert bereits an einer früheren Stelle des Vektors aufgetaucht ist. Die Funktion `unique(⟨Vektor⟩)` nennt alle voneinander verschiedenen Werte eines Vektors, mehrfach vorkommende Werte werden also nur einmal aufgeführt. Die Funktion eignet sich in Kombination mit `length()` zum Zählen der tatsächlich vorkommenden unterschiedlichen Werte einer Variable.

```
> duplicated(c(1, 1, 1, 3, 3, 4, 4))
[1] FALSE TRUE TRUE FALSE TRUE FALSE TRUE

> unique(c(1, 1, 1, 3, 3, 4, 4))
[1] 1  3  4

> length(unique(c("A", "B", "C", "C", "B", "B", "A", "C", "C", "A")))
[1] 3
```

2.3.2 Mengenoperationen

Die `union(x=⟨Vektor1⟩, y=⟨Vektor2⟩)` Funktion bildet die Vereinigungsmenge von `x` und `y`. Das Ergebnis sind die Werte, die Element mindestens einer der beiden Mengen sind, wobei duplizierte Werte gelöscht werden. Wird das Ergebnis als Menge betrachtet, spielt die Reihenfolge, in der `x` und `y` genannt werden, keine Rolle.

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

> union(y, x)
[1] 5 3 1 4 2
```

Die Schnittmenge zweier Mengen wird mit `intersect(x=⟨Vektor1⟩, y=⟨Vektor2⟩)` erzeugt. Das Ergebnis sind die Werte, die sowohl Element von `x` als auch Element von `y` sind, wobei duplizierte Werte gelöscht werden. Auch hier ist die Reihenfolge von `x` und `y` unerheblich, wenn das Ergebnis als Menge betrachtet wird.

```
> intersect(x, y)
[1] 1 3

> intersect(y, x)
[1] 3 1
```

Mit `setequal(x=⟨Vektor1⟩, y=⟨Vektor2⟩)` lässt sich prüfen, ob als Mengen betrachtete Vektoren identisch sind.

```
> setequal(c(1, 1, 2, 2), c(2, 1))
[1] TRUE
```

Die Funktion `setdiff(x=⟨Vektor1⟩, y=⟨Vektor2⟩)` liefert als Ergebnis all jene Elemente von `x`, die nicht Element von `y` sind. Im Unterschied zu den oben behandelten Mengenoperationen ist die Reihenfolge, in der `x` und `y` angegeben sind, bedeutsam, auch wenn das Ergebnis als Menge betrachtet wird. Die symmetrische Differenz von `x` und `y` erhält man also durch `union(setdiff(x, y), setdiff(`↘
`→y, x))`.

```
> setdiff(x, y)
[1] 2

> setdiff(y, x)
[1] 5 4
```

Soll jedes Element eines Vektors daraufhin geprüft werden, ob es Element einer Menge ist, kann die `is.element(el=⟨Menge1⟩, set=⟨Menge2⟩)` Funktion genutzt werden. Unter `el` ist der Vektor mit den zu prüfenden Elementen einzutragen und unter `set` die durch einen Vektor definierte Menge. Als Ergebnis wird ein logischer Vektor ausgegeben, der für jedes Element von `el` angibt, ob es in `set` enthalten ist. Die Kurzform in Operator-Schreibweise lautet `⟨Menge1⟩ %in%` ↘
`→⟨Menge2⟩`.

```
> is.element(c(29, 23, 30, 17, 30, 10), c(30, 23))
[1] FALSE TRUE TRUE FALSE TRUE FALSE

> c("A", "Z", "B") %in% c("A", "B", "C", "D", "E")
[1] TRUE FALSE TRUE
```

Durch `all(x %in% y)` lässt sich so prüfen, ob `x` eine Teilmenge von `y` darstellt, ob also jedes Element von `x` auch Element von `y` ist. Dabei ist `x` eine echte Teilmenge von `y`, wenn sowohl `all(x %in% y)` gleich `TRUE` als auch `all(y %in% x)` gleich `FALSE` ist.

```
> A <- c(4, 5, 6)
> B <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
> (AinB <- all(A %in% B))            # A Teilmenge von B?
[1] TRUE

> (BinA <- all(B %in% A))            # B Teilmenge von A?
[1] FALSE

> AinB & !BinA                       # A echte Teilmenge von B?
[1] TRUE
```

In Kombination mit dem `ifelse()` Befehl kann `%in%` beispielsweise genutzt werden, um eine kategoriale Variable so umzucodieren, dass alle nicht in einer bestimmten Menge auftauchenden Werte in einer Kategorie „Sonstiges“ zusammengefasst werden. Im konkreten Beispiel sollen nur die ersten 15 Buchstaben des Alphabets als solche beibehalten, alle anderen Werte zu `"other"` recodiert werden.

```
> targetSet   <- c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K")
> response    <- c("Z", "E", "O", "W", "H", "C", "I", "G", "A", "O", "B")
> (respRec    <- ifelse(response %in% targetSet, response, "other"))
[1] "other" "E" "other" "other" "H" "C" "I" "G" "A" "other" "B"
```

2.3.3 Kombinatorik

Aus dem Bereich der Kombinatorik sind im Rahmen der Datenauswertung bisweilen drei Themen von Bedeutung: Zunächst ist dies die Zusammenstellung von Teilmengen aus Elementen einer Grundmenge. Dabei ist die Reihenfolge der Elemente innerhalb einer Teilmenge meist nicht bedeutsam, d. h. es handelt sich um eine *Kombination*. Werden alle Elemente einer Grundmenge ohne Zurücklegen unter Beachtung der Reihenfolge gezogen, handelt es sich um eine *Permutation*. Schließlich kann die Zusammenstellung von Elementen aus verschiedenen Grundmengen notwendig sein, wobei jeweils ein Element aus jeder Grundmenge beteiligt sein soll.

Die Kombination entspricht dem Ziehen aus einer Grundmenge ohne Zurücklegen sowie ohne Berücksichtigung der Reihenfolge. Oft wird die Anzahl der Elemente der Grundmenge mit n, die Anzahl der gezogenen Elemente mit k und die Kombination deshalb mit k-Kombination bezeichnet. Insgesamt gibt es $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ viele k-Kombinationen. Da eine k-Kombinationen die Anzahl der Möglichkeiten darstellt, aus einer Menge mit n Elementen k auszuwählen, spricht man im Englischen beim Binomialkoeffizienten $\binom{n}{k}$ von „n choose k“, woraus sich der Name der `choose(n=⟨Zahl⟩, k=⟨Zahl⟩)` Funktion ableitet, die ihn ermittelt. Die Fakultät einer Zahl wird mit `factorial(⟨Zahl⟩)` berechnet (vgl. Abschn. 2.7.1).

```
> myN <- 5
> myK <- 4
> choose(myN, myK)
[1] 5

> factorial(myN) / (factorial(myK)*factorial(myN-myK))        # Kontrolle
[1] 5
```

Möchte man alle k-Kombinationen einer gegebenen Grundmenge x auch explizit anzeigen lassen, kann dies mit der Funktion `combn()` geschehen.

```
> combn(x=⟨Vektor⟩, m=⟨Zahl⟩, simplify=TRUE, FUN=⟨Funktion⟩, ...)
```

Die Zahl `m` entspricht dabei dem k der bisherigen Terminologie. Mit `simplify=TRUE` erfolgt die Ausgabe auf möglichst einfache Weise, d. h. nicht als Liste (vgl. Abschn. 3.1). Stattdessen wird ein Vektor, oder wie hier eine Matrix ausgegeben, die in jeder Spalte eine der k-Kombinationen enthält (vgl. Abschn. 2.8).

```
> combn(c("a", "b", "c", "d", "e"), myK)
     [,1]  [,2]  [,3]  [,4]  [,5]
[1,]  "a"   "a"   "a"   "a"   "b"
[2,]  "b"   "b"   "b"   "c"   "c"
[3,]  "c"   "c"   "d"   "d"   "d"
[4,]  "d"   "e"   "e"   "e"   "e"
```

Die `combn()` Funktion lässt sich darüber hinaus anwenden, um in einem Arbeitsschritt eine frei wählbare Funktion auf jede gebildete k-Kombination anzuwenden. Das Argument `FUN` erwartet hierfür eine Funktion, die einen Kennwert jedes sich als Kombination ergebenden Vektors bestimmt. Benötigt `FUN` ihrerseits weitere Argumente, so können diese unter `...` durch Komma getrennt an `combn()` übergeben werden.

```
> combn(c(1, 2, 3, 4), 3)              # alle 3-Kombinationen
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    2
[2,]    2    2    3    3
[3,]    3    4    4    4

# jeweilige Summe jeder 3-Kombination
> combn(c(1, 2, 3, 4), 3, sum)
[1] 6 7 8 9

# gewichtetes Mittel jeder 3-Kombination mit Argument w für Gewichte
> combn(c(1, 2, 3, 4), 3, weighted.mean, w=c(0.5, 0.2, 0.3))
[1] 1.8 2.1 2.3 2.8
```

Die Funktion `permutations()` aus dem `e1071` Paket (Dimitriadou, Hornik, Leisch, Meyer & Weingessel, 2011) stellt alle $n!$ Permutationen der natürlichen Zahlen $1, \ldots, n$ als Zeilen einer Matrix zusammen (vgl. Abschn. 2.8). Für eine einzelne zufällige Permutation vgl. Abschn. 2.4.3.

```
> library(e1071)                       # für permutations()
> permutations(3)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    1    3
[3,]    2    3    1
[4,]    1    3    2
[5,]    3    1    2
[6,]    3    2    1
```

Die Funktion `expand.grid(⟨Vektor1⟩, ⟨Vektor2⟩, ...)` bildet alle Kombinationen von Elementen mehrerer Grundmengen, wobei jeweils ein Element aus jeder Grundmenge stammt und die Reihenfolge nicht berücksichtigt wird. Dies entspricht der Situation, dass aus den Stufen mehrerer unabhängiger Variablen (UVn) alle Kombinationen von Faktorstufen gebildet werden. Das Ergebnis von `expand.grid()` ist ein Datensatz (vgl. Abschn. 3.2), bei dem jede Kombination in einer Zeile steht. Die zuerst genannte Variable variiert dabei am schnellsten über die Zeilen, die anderen entsprechend ihrer Position im Funktionsaufruf langsamer.

```
> IV1 <- c("control", "treatment")
> IV2 <- c("f", "m")
> IV3 <- c(1, 2)
> expand.grid(IV1, IV2, IV3)
        Var1  Var2  Var3
1    control     f     1
2  treatment     f     1
3    control     m     1
```

```
4   treatment     m     1
5     control     f     2
6   treatment     f     2
7     control     m     2
8   treatment     m     2
```

Soll es sich beim Ergebnis von `expand.grid()` tatsächlich um vollständig gekreuzte Faktorstufen im versuchsplanerischen Sinn handeln (was hier für die dritte Spalte der Ausgabe nicht der Fall ist), so sollten die einzelnen Variablen Objekte der Klasse `factor` sein (vgl. Abschn. 2.6, insbesondere 2.6.6).

2.4 Systematische und zufällige Wertefolgen erzeugen

Ein häufig auftretender Arbeitsschritt in R ist die Erstellung von Zahlenfolgen nach vorgegebenen Gesetzmäßigkeiten, wie etwa sequenzielle Abfolgen von Zahlen oder Wiederholungen bestimmter Wertmuster.

Aber auch Zufallszahlen und zufällige Reihenfolgen sind ein unverzichtbares Hilfsmittel der Datenauswertung, wobei ihnen insbesondere in der Planung von Analysen anhand simulierter Daten vor einer tatsächlichen Datenerhebung eine große Bedeutung zukommt.[11] Zufällige Datensätze können unter Einhaltung vorgegebener Wertebereiche und anderer Randbedingungen erstellt werden. So können sie empirische Gegebenheiten realistisch widerspiegeln und statistische Voraussetzungen der eingesetzten Verfahren berücksichtigen. Aber auch bei der zufälligen Auswahl von Teilstichproben eines Datensatzes oder beim Erstellen zufälliger Reihenfolgen zur Zuordnung von VPn auf experimentelle Bedingungen kommen Zufallszahlen zum Einsatz.

2.4.1 Numerische Sequenzen erstellen

Zahlenfolgen mit Einerschritten, etwa für eine fortlaufende Nummerierung, können mithilfe des Operators ⟨`Startwert`⟩`:`⟨`Endwert`⟩ erzeugt werden – in aufsteigender wie auch in absteigender Reihenfolge.

```
> 20:26
[1] 20 21 22 23 24 25 26
```

[11] Wenn hier und im Folgenden von Zufallszahlen die Rede ist, sind immer sog. *Pseudozufallszahlen* gemeint. Diese kommen nicht im eigentlichen Sinn zufällig zustande, sind aber von tatsächlich zufälligen Zahlenfolgen im Ergebnis fast nicht zu unterscheiden. Pseudozufallszahlen hängen deterministisch vom Zustand des die Zahlen produzierenden Generators ab. Wird sein Zustand über die Funktion `set.seed(`⟨`Zahl`⟩`)` festgelegt, kommt bei gleicher ⟨`Zahl`⟩ bei späteren Aufrufen von Zufallsfunktionen immer dieselbe Folge von Werten zustande. Dies gewährleistet die Reproduzierbarkeit von Auswertungsschritten bei Simulationen mit Zufallsdaten. Nach welcher Methode Zufallszahlen generiert werden, ist konfigurierbar, vgl. `?RNGkind`.

```
> 26:20
[1] 26 25 24 23 22 21 20
```

Bei Zahlenfolgen im negativen Bereich sollten Klammern Verwendung finden, um nicht versehentlich eine nicht gemeinte Sequenz zu produzieren.

```
> -4:2              # negatives Vorzeichen bezieht sich nur auf die 4
[1] -4 -3 -2 -1 0 1 2

> -(4:2)            # negatives Vorzeichen bezieht sich auf Sequenz 4:2
[1] -4 -3 -2
```

Zahlenfolgen mit beliebiger Schrittweite lassen sich mit `seq()` erzeugen.

```
> seq(from=⟨Zahl⟩, to=⟨Zahl⟩, by=⟨Schrittweite⟩, length.out=⟨Länge⟩)
```

Dabei können Start- (`from`) und Endpunkt (`to`) des durch die Sequenz abzudeckenden Intervalls ebenso gewählt werden wie die gewünschte Schrittweite (`by`) bzw. stattdessen die gewünschte Anzahl der Elemente der Zahlenfolge (`length.out`). Die Sequenz endet vor `to`, wenn die Schrittweite kein ganzzahliges Vielfaches der Differenz von Start- und Endwert ist.

```
> seq(from=2, to=12, by=2)
[1] 2 4 6 8 10 12

> seq(from=2, to=11, by=2)              # Endpunkt wird nicht erreicht
[1] 2 4 6 8 10

> seq(from=0, to=-1, length.out=5)
[1] 0.00 -0.25 -0.50 -0.75 -1.00
```

Eine Möglichkeit zum Erstellen einer bei 1 beginnenden Sequenz in Einerschritten, die genauso lang ist wie ein bereits vorhandener Vektor, besteht mit `seq(along=⟨Vektor⟩)`. Dabei muss `along=⟨Vektor⟩` das einzige Argument von `seq()` sein. Dies ist die bevorzugte Art, für einen vorhandenen Vektor den passenden Vektor seiner Indizes zu erstellen. Vermieden werden sollte dagegen die `1:length(⟨Vektor⟩)` Sequenz, deren Behandlung von Vektoren der Länge 0 meist nicht sinnvoll ist.

```
> age <- c(18, 20, 30, 24, 23, 21)
> seq(along=age)
[1] 1 2 3 4 5 6

> vec <- numeric(0)             # leeren Vektor (Länge 0) erzeugen
> 1:length(vec)                 # hier unerwünschtes Ergebnis: Sequenz 1:0
[1] 1 0

> seq(along=vec)                # sinnvolleres Ergebnis: leerer Vektor
[1] integer(0)
```

2.4.2 Wertefolgen wiederholen

Eine andere Art von Wertefolgen kann mit der `rep()` Funktion (repeat) erzeugt werden, die Elemente wiederholt ausgibt.

```
> rep(x=⟨Vektor⟩, times=⟨Anzahl⟩, each=⟨Anzahl⟩)
```

Für `x` ist ein Vektor einzutragen, der auf zwei verschiedene Arten wiederholt werden kann. Mit dem Argument `times` wird er als Ganzes so oft aneinander gehängt wie angegeben.

```
> rep(1:3, times=5)
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

Wird für das Argument `times` ein Vektor angegeben, so muss dieser dieselbe Länge wie `x` besitzen – hier wird ein kürzerer Vektor durch R nicht selbsttätig zyklisch wiederholt. Jedes Element des Vektors `times` gibt an, wie häufig das an gleicher Position stehende Element von `x` wiederholt werden soll, ehe das nächste Element von `x` wiederholt und angehängt wird.

```
> rep(c("A", "B", "C"), times=c(2, 3, 4))
[1] "A" "A" "B" "B" "B" "C" "C" "C" "C"
```

Wird das Argument `each` verwendet, wird jedes Element von `x` einzeln mit der gewünschten Häufigkeit wiederholt, bevor das nächste Element von `x` einzeln wiederholt und angehängt wird.

```
> rep(age, each=2)
[1] 18 18 20 20 30 30 24 24 23 23 21 21
```

2.4.3 Zufällig aus einer Urne ziehen

Die Funktion `sample()` stellt eine einfache Art zur Verfügung, einen aus zufälligen Werten bestehenden Vektor zu erstellen, indem das Ziehen aus einer Urne simuliert wird.

```
> sample(x=⟨Vektor⟩, size=⟨Anzahl⟩, replace=FALSE, prob=NULL)
```

Für `x` ist ein Vektor zu nennen, der die Elemente der Urne festlegt, aus der gezogen wird. Dies sind die Werte, aus denen sich die Zufallsfolge zusammensetzen soll. Es können Vektoren vom Datentyp `numeric` (etwa `1:50`), `character` (`c("Kopf", "Zahl")`) oder auch `logical` (`c(TRUE, FALSE)`) verwendet werden. Unter `size` ist die gewünschte Anzahl der zu ziehenden Elemente einzutragen. Mit dem Argument `replace` wird die Art des Ziehens festgelegt: Auf `FALSE` gesetzt (Voreinstellung) wird ohne, andernfalls (`TRUE`) mit Zurücklegen gezogen. Natürlich kann ohne Zurücklegen aus einem Vektor der Länge n nicht häufiger als n mal gezogen werden. Wenn `replace=FALSE` und dennoch `size` größer als `length(x)` ist, erzeugt R deswegen eine Fehlermeldung. Für den Fall, dass nicht alle Elemente der Urne

dieselbe Auftretenswahrscheinlichkeit besitzen sollen, existiert das Argument `prob`. Es benötigt einen Vektor derselben Länge wie `x`, dessen Elemente die Auftretenswahrscheinlichkeit für jedes Element von `x` bestimmen.

```
> sample(1:6, size=20, replace=TRUE)
[1] 4 1 2 5 6 5 3 6 6 5 1 6 1 5 1 4 5 4 4 2

> sample(c("rot", "grün", "blau"), size=8, replace=TRUE)
[1] "grün" "blau" "grün" "rot" "rot" "blau" "grün" "blau"
```

Für `sample()` existieren zwei Kurzformen, auf die jedoch aufgrund der Gefahr von Verwechslungen besser verzichtet werden sollte: `sample(⟨Vektor⟩)` ist gleichbedeutend mit `sample(⟨Vektor⟩, size=length(⟨Vektor⟩), replace=FALSE`↘→`)`, erstellt also eine zufällige Permutation der Elemente von `⟨Vektor⟩` (vgl. Abschn. 2.3.3). Darauf aufbauend steht `sample(⟨Zahl⟩)` kurz für `sample(1:⟨Zahl⟩)`, also für `sample(1:⟨Zahl⟩, size=⟨Zahl⟩, replace=FALSE)`. Wenn für `⟨Vektor⟩` ein Objektname übergeben wird, steht oft vor der Ausführung nicht fest, wie viele Elemente er beinhalten wird. Enthält `⟨Vektor⟩` z. B. durch die Auswahl einer Teilmenge unvorhergesehen als einziges Element eine Zahl, wird die Urne durch Elemente definiert, die womöglich nicht im ursprünglichen Vektor vorhanden waren.

```
> x <- c(2, 4, 6, 8)
> sample(x[(x %% 4) == 0])          # äquivalent zu sample(c(4, 8))
[1] 8 4

# Urne mit Elementen, die nicht aus x stammen
> sample(x[(x %% 8) == 0])          # äquivalent zu sample(8), d.h. sample(1:8)
[1] 2 1 7 5 4 8 6 3
```

2.4.4 Zufallszahlen aus bestimmten Verteilungen erzeugen

Abgesehen vom zufälligen Ziehen aus einer vorgegebenen Menge endlich vieler Werte lassen sich auch Zufallszahlen mit bestimmten Eigenschaften generieren. Dazu können mit Funktionen, deren Name nach dem Muster `r⟨Funktionsfamilie⟩` aufgebaut ist, Realisierungen von Zufallsvariablen mit verschiedenen Verteilungen erstellt werden (vgl. Abschn. 5.3). Diese Möglichkeit ist insbesondere für die Simulation empirischer Daten nützlich.

```
>  runif(n=⟨Anzahl⟩,   min=0, max=1)          # Gleichverteilung
> rbinom(n=⟨Anzahl⟩,   size, prob)           # Binomialverteilung
>  rnorm(n=⟨Anzahl⟩,   mean=0, sd=1)         # Normalverteilung
> rchisq(n=⟨Anzahl⟩,   df,         ncp=0)    # chi^2-Verteilung
>     rt(n=⟨Anzahl⟩,   df,         ncp=0)    # t-Verteilung
>     rf(n=⟨Anzahl⟩,   df1, df2,   ncp=0)    # F-Verteilung
```

Als erstes Argument `n` ist immer die gewünschte Anzahl an Zufallszahlen anzugeben. Bei `runif()` definiert `min` die untere und `max` die obere Grenze des Zahlenbereichs, aus dem gezogen wird. Beide Argumente akzeptieren auch Vektoren der Länge n, die für jede einzelne Zufallszahl den zulässigen Wertebereich angeben.

Bei `rbinom()` entsteht jede der n Zufallszahlen als Anzahl der Treffer in einer simulierten Serie von gleichen Bernoulli-Experimenten, die ihrerseits durch die Argumente `size` und `prob` charakterisiert ist. `size` gibt an, wie häufig ein einzelnes Bernoulli-Experiment wiederholt werden soll, `prob` legt die Trefferwahrscheinlichkeit in jedem dieser Experimente fest. Sowohl `size` als auch `prob` können Vektoren der Länge n sein, die dann die Bernoulli-Experimente charakterisieren, deren Simulation zu jeweils einer Zufallszahl führt.

Bei `rnorm()` sind der Erwartungswert `mean` und die theoretische Streuung `sd` der normalverteilten Variable anzugeben, die simuliert werden soll.[12] Auch diese Argumente können Vektoren der Länge n sein und für jede Zufallszahl andere Parameter vorgeben.

Sind Verteilungen über Freiheitsgrade und Nonzentralitätsparameter charakterisiert, werden diese mit den Argumenten `df` (degrees of freedom) respektive `ncp` (non-centrality parameter) ggf. in Form von Vektoren übergeben.

```
> runif(5, min=1, max=6)
[1] 4.411716 3.893652 2.412720 5.676668 2.446302

> rbinom(20, size=5, prob=0.3)
[1] 2 0 3 0 2 2 1 0 1 0 2 1 1 4 2 2 1 1 3 3

> rnorm(6, mean=100, sd=15)
[1] 101.13170 102.25592 88.31622 101.22469 93.12013 100.52888
```

2.5 Daten transformieren

Häufig sind für spätere Auswertungen neue Variablen auf Basis der erhobenen Daten zu bilden. Im Rahmen solcher Datentransformationen können etwa Werte sortiert, umskaliert, ersetzt, ausgewählt, oder verschiedene Variablen zu einer neuen verrechnet werden. Genauso ist es möglich, kontinuierliche Variablen in Kategorien einzuteilen, oder in Rangwerte umzuwandeln.

2.5.1 Werte sortieren

Um die Reihenfolge eines Vektors umzukehren, kann die `rev(⟨Vektor⟩)` Funktion (reverse) benutzt werden.

```
> vec <- c(10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)
> rev(vec)
[1] 20 19 18 17 16 15 14 13 12 11 10
```

[12] Der die Breite (Dispersion) einer Normalverteilung charakterisierende Parameter ist in R-Funktionen immer die Streuung σ, in der Literatur dagegen häufig die Varianz σ^2.

Die Elemente eines Vektors lassen sich auch entsprechend ihrer Reihenfolge sortieren, die wiederum vom Datentyp des Vektors abhängt: Bei numerischen Vektoren bestimmt die Größe der gespeicherten Zahlen, bei Vektoren aus Zeichenketten die alphabetische Reihenfolge der Elemente die Ausgabe. Zum Sortieren stehen die Funktionen `sort()` und `order()` zur Verfügung.

```
> sort(⟨Vektor⟩,        decreasing=FALSE)
> order(⟨Vektor⟩,       decreasing=FALSE)
```

`sort()` gibt den sortierten Vektor direkt aus. Dagegen ist das Ergebnis von `order()` ein Indexvektor, der die Indizes des zu ordnenden Vektors in der Reihenfolge seiner Elemente enthält. Im Gegensatz zu `sort()` gibt `order()` also nicht schon die sortierten Datenwerte, sondern nur die zugehörigen Indizes aus, die anschließend zum Indizieren des Vektors verwendet werden können. Daher ist bei Vektoren `sort(⟨Vektor⟩)` äquivalent zu `⟨Vektor⟩[order(⟨Vektor⟩)]`. Der Vorteil von `order()` tritt beim Umgang mit Matrizen und Datensätzen zutage (vgl. Abschn. 2.8.6). Die Sortierreihenfolge wird über das Argument `decreasing` kontrolliert. Per Voreinstellung auf `FALSE` gesetzt, wird aufsteigend sortiert. Mit `decreasing=TRUE` ist die Reihenfolge absteigend.

```
> vec <- c(10, 12, 1, 12, 7, 16, 6, 19, 10, 19)
> sort(vec)
[1] 1 6 7 10 10 12 12 16 19 19

> (idxDec <- order(vec, decreasing=TRUE))
[1] 8 10 6 2 4 1 9 5 7 3

> vec[idxDec]
[1] 19 19 16 12 12 10 10 7 6 1
```

Wenn Vektoren vom Datentyp `character` sortiert werden, so geschieht dieses in alphabetischer Reihenfolge. Auch als Zeichenkette gespeicherte Zahlen werden hierbei alphabetisch sortiert, d. h. die Zeichenkette `"10"` käme vor `"4"`.

```
> sort(c("D", "E", "10", "A", "F", "E", "D", "4", "E", "A"))
[1] "10" "4" "A" "A" "D" "D" "E" "E" "E" "F"
```

2.5.2 Werte in zufällige Reihenfolge bringen

Zufällige Reihenfolgen können mit Kombinationen von `rep()` und `sample()` erstellt werden und entsprechen der zufälligen Permutation einer Menge (vgl. Abschn. 2.3.3). Sie sind z. B. bei der randomisierten Zuteilung von VPn zu Gruppen, beim Randomisieren der Reihenfolge von Bedingungen oder beim Ziehen einer Zufallsstichprobe aus einer Datenmenge nützlich.

```
# randomisiere Reihenfolge von 5 Farben
> myColors      <- c("red", "green", "blue", "yellow", "black")
> (randCols     <- sample(myColors, length(myColors), replace=FALSE))
[1] "yellow" "green" "red" "blue" "black"
```

```
# teile 12 VPn auf 3 unterschiedlich große Gruppen auf
> P      <- 3                          # Anzahl Gruppen
> Nj     <- c(4, 3, 5)                 # Gruppengrößen
> (IV    <- rep(1:P, Nj))              # Gruppenzugehörigkeiten
[1] 1 1 1 1 2 2 2 3 3 3 3 3

# zufällige Permutation
> (IVrand <- sample(IV, length(IV), replace=FALSE))
[1] 2 1 1 3 3 1 3 1 2 2 3 3
```

Um allgemein n VPn auf p möglichst ähnlich große Gruppen aufzuteilen, können zunächst mit `sample()` die Indizes $1, \ldots, n$ permutiert werden, um dann mit `%%` den Rest der ganzzahligen Division jedes Index mit p zu bilden, der p Werte $0, \ldots, p-1$ annehmen kann.

```
> N <- 20
> (sample(1:N, N, replace=FALSE) %% P) + 1
[1] 2 2 2 3 3 3 1 3 1 2 3 2 1 1 2 3 2 1 3 1
```

2.5.3 Teilmengen von Daten auswählen

Soll aus einer vorhandenen Datenmenge eine Teilstichprobe gezogen werden, hängt das Vorgehen von der genau intendierten Art des Ziehens ab. Grundsätzlich können zur Auswahl von Werten logische wie numerische Indexvektoren zum Einsatz kommen, die sich systematisch oder zufällig erzeugen lassen.

Eine rein zufällige Unterauswahl bestimmten Umfangs ohne weitere Nebenbedingungen kann mit `sample()` erstellt werden.[13] Dazu betrachtet man den Datenvektor als Urne, aus der ohne Zurücklegen die gewünschte Anzahl von Beobachtungen gezogen wird.

```
> vec <- rep(c("rot", "grün", "blau"), 10)
> sample(vec, 5, replace=FALSE)
[1] "blau" "grün" "blau" "grün" "rot"
```

Ein anderes Ziel könnte darin bestehen, z. B. jedes zehnte Element einer Datenreihe auszugeben. Hier bietet sich die `seq()` Funktion an, um die passenden Indizes zu erzeugen

```
> selIdx1 <- seq(1, length(vec), by=10)
> vec[selIdx1]
[1] "rot"  "grün" "blau"
```

Soll nicht genau, sondern nur im Mittel jedes zehnte Element ausgegeben werden, eignet sich die Funktion `rbinom()` zum Erstellen eines geeigneten Indexvektors. Dazu kann der Vektor der Trefferanzahlen aus einer Serie von jeweils nur einmal durchgeführten Bernoulli-Experimenten mit Trefferwahrscheinlichkeit $\frac{1}{10}$ in einen logischen Indexvektor umgewandelt werden:

[13] Das Paket `car` (Fox & Weisberg, 2011a) bietet hierfür die Funktion `some()`, die sich auch für Matrizen (vgl. Abschn. 2.8) oder Datensätze (vgl. Abschn. 3.2) eignet.

```
> selIdx2 <- rbinom(length(vec), 1, 0.1) == 1
> vec[selIdx2]
[1] "blau" "grün" "blau" "grün" "grün"
```

2.5.4 Daten umrechnen

Auf Vektoren lassen sich alle elementaren Rechenoperationen anwenden, die in Abschn. 1.2.4 für Skalare aufgeführt wurden. Vektoren können also in den meisten Rechnungen wie Einzelwerte verwendet werden, wodurch sich Variablen leicht umskalieren lassen. Die Berechnungen einer Funktion werden dafür elementweise durchgeführt: Die Funktion wird zunächst auf das erste Element des Vektors angewendet, dann auf das zweite, usw., bis zum letzten Element. Das Ergebnis ist ein Vektor, der als Elemente die Einzelergebnisse besitzt. In der Konsequenz ähnelt die Schreibweise zur Transformation von in Vektoren gespeicherten Werten in R sehr der aus mathematischen Formeln gewohnten.

```
> age <- c(18, 20, 30, 24, 23, 21)
> age/10
[1] 1.8 2.0 3.0 2.4 2.3 2.1

> (age/2) + 5
[1] 14.0 15.0 20.0 17.0 16.5 15.5
```

Auch in Situationen, in denen mehrere Vektoren in einer Rechnung auftauchen, können diese wie Einzelwerte verwendet werden. Die Vektoren werden dann elementweise entsprechend der gewählten Rechenoperation miteinander verrechnet. Dabei wird das erste Element des ersten Vektors mit dem ersten Element des zweiten Vektors z. B. multipliziert, ebenso das zweite Element des ersten Vektors mit dem zweiten Element des zweiten Vektors, usw.

```
> vec1 <- c(3, 4, 5, 6)
> vec2 <- c(-2, 2, -2, 2)
> vec1*vec2
[1] -6 8 -10 12

> vec3 <- c(10, 100, 1000, 10000)
> (vec1 + vec2) / vec3
[1] 1e-01 6e-02 3e-03 8e-04
```

Die Zahlen der letzten Ausgabe sind in verkürzter Exponentialschreibweise dargestellt (vgl. Abschn. 1.2.4).

2.5.4.1 Zyklische Verlängerung von Vektoren (recycling)

Die Verrechnung mehrerer Vektoren scheint aufgrund der elementweisen Zuordnung zunächst vorauszusetzen, dass die Vektoren dieselbe Länge haben. Tatsächlich ist dies nicht unbedingt notwendig, weil R in den meisten Fällen diesen Zustand ggf.

selbsttätig herstellt. Dabei wird der kürzere Vektor intern von R zyklisch wiederholt (also sich selbst angefügt, sog. *recycling*), bis er mindestens die Länge des längeren Vektors besitzt. Eine Warnmeldung wird in einem solchen Fall nur dann ausgegeben, wenn die Länge des längeren Vektors kein ganzzahliges Vielfaches der Länge des kürzeren Vektors ist. Dies ist gefährlich, weil meist Vektoren gleicher Länge miteinander verrechnet werden sollen und die Verwendung von Vektoren ungleicher Länge ein Hinweis auf fehlerhafte Berechnungen sein kann.

```
> vec1 <- c(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24)
> vec2 <- c(2, 4, 6, 8, 10)

> c(length(age), length(vec1), length(vec2))
[1] 6  12  5

> vec1*age
[1] 36 80 180 192 230 252 252 320 540 480 506 504

> vec2*age
[1] 36 80 180 192 230 42

Warning message:
Länge des längeren Objektes ist kein Vielfaches der Länge des kürzeren
Objektes in: vec2 * age
```

2.5.4.2 z-Transformation

Durch eine z-Transformation wird eine quantitative Variable X so normiert, dass sie den Mittelwert $M = 0$ und die Standardabweichung $s = 1$ besitzt. Dies geschieht für jeden Einzelwert x_i durch $\frac{x_i - M}{s}$. Die Funktion `mean(⟨Vektor⟩)` berechnet den Mittelwert (vgl. Abschn. 2.7.3), `sd(⟨Vektor⟩)` ermittelt die korrigierte Streuung (vgl. Abschn. 2.7.6).

```
> (zAge <- (age - mean(age)) / sd(age))
[1] -1.1166106 -0.6380632 1.7546739 0.3190316 0.0797579 -0.3987895
```

Eine andere Möglichkeit bietet die `scale(x=⟨Vektor⟩, center=TRUE, scale=TRUE)` Funktion. Diese berechnet die z-Werte mithilfe der korrigierten Streuung, gibt sie jedoch nicht in Form eines Vektors, sondern als Matrix mit einer Spalte aus.[14] Weiterhin werden Mittelwert und korrigierte Streuung von `x` in Form von Attributen mit angegeben. Standardisierung und Zentrierung können unabhängig voneinander ausgewählt werden: Für die zentrierten, nicht aber standardisierten Werte von `x` ist etwa `scale=FALSE` zu setzen und `center=TRUE` zu belassen.

```
> (zAge <- scale(age))
            [,1]
[1,]   -1.1166106
[2,]   -0.6380632
```

[14] Für `x` kann auch eine Matrix übergeben werden, deren jeweils z-transformierte Spalten dann die Spalten der ausgegebenen Matrix ausmachen (vgl. Abschn. 2.8).

```
[3,]    1.7546739
[4,]    0.3190316
[5,]    0.0797579
[6,]   -0.3987895

attr(,"scaled:center")
[1] 22.66667

attr(,"scaled:scale")
[1] 4.179314
```

Um die ausgegebene Matrix wieder in einen Vektor zu verwandeln, muss sie wie in Abschn. 2.8.2 dargestellt mit `as.vector(⟨Matrix⟩)` konvertiert werden.

```
> as.vector(zAge)
[1] -1.1166106 -0.6380632 1.7546739 0.3190316 0.0797579 -0.3987895
```

Durch Umkehrung des Prinzips der z-Transformation lassen sich empirische Datenreihen so skalieren, dass sie einen beliebigen Mittelwert M_{neu} und eine beliebige Streuung s_{neu} besitzen. Dies geschieht für eine z-transformierte Variable Z mit $Z \cdot s_{\text{neu}} + M_{\text{neu}}$.

```
> newSd       <- 15
> newMean     <- 100
> (newAge     <- (as.vector(zAge)*newSd) + newMean)
[1] 83.25084 90.42905 126.32011 104.78547 101.19637 94.01816

> mean(newAge)                              # Kontrolle: Mittelwert
[1] 100

> sd(newAge)                                # Kontrolle: Streuung
[1] 15
```

2.5.4.3 Rangtransformation

Die `rank(⟨Vektor⟩)` Funktion gibt für jedes Element eines Vektors seinen Rangplatz an, der sich an der Position des Wertes im sortierten Vektor orientiert und damit der Ausgabe von `order()` ähnelt. Anders als bei `order()` erhalten identische Werte in der Voreinstellung jedoch denselben Rang. Das Verhalten, mit dem bei solchen sog. *Bindungen* Ränge ermittelt werden, kontrolliert das Argument `ties.method` – Voreinstellung sind mittlere Ränge.

```
> rank(c(3, 1, 2, 3))
[1] 3.5 1.0 2.0 3.5
```

2.5.5 Neue aus bestehenden Variablen bilden

Das elementweise Verrechnen mehrerer Vektoren kann, analog zur z-Transformation, allgemein zur flexiblen Neubildung von Variablen aus bereits bestehenden Daten genutzt werden.

Ein Beispiel sei die Berechnung des Body-Mass-Index (BMI) einer Person, für den ihr Körpergewicht in kg durch das Quadrat ihrer Körpergröße in m geteilt wird.

```
> height    <- c(1.78, 1.91, 1.89, 1.83, 1.64)
> weight    <- c(65, 89, 91, 75, 73)
> (bmi      <- weight / (height^2))
[1] 20.51509 24.39626 25.47521 22.39541 27.14158
```

In einem zweiten Beispiel soll die Summenvariable aus drei dichotomen Items („trifft zu“: TRUE, „trifft nicht zu“: FALSE) eines an 8 Personen erhobenen Fragebogens gebildet werden. Dies ist die Variable, die jeder Person den Summenscore aus ihren Antworten zuordnet, also angibt, wie viele Items von der Person als zutreffend angekreuzt wurden. Logische Werte verhalten sich bei numerischen Rechnungen wie 1 (TRUE) bzw. 0 (FALSE).

```
> quest1     <- c(FALSE,   FALSE,  FALSE,  TRUE,   FALSE,  TRUE, FALSE,   TRUE)
> quest2     <- c(TRUE,    FALSE,  FALSE,  FALSE,  TRUE,   TRUE, TRUE,    FALSE)
> quest3     <- c(TRUE,    TRUE,   TRUE,   TRUE,   FALSE,  TRUE, FALSE,   FALSE)
> (sumVar    <- quest1 + quest2 + quest3)
[1] 2 1 1 2 1 3 1 1
```

2.5.6 Werte ersetzen oder recodieren

Mitunter werden Variablen zunächst auf eine bestimmte Art codiert, die sich später für manche Auswertungen als nicht zweckmäßig erweist und deswegen geändert werden soll. Dann müssen bestimmte Werte gesucht und ersetzt werden, was sich auf verschiedenen Wegen erreichen lässt. Dabei sollte die Variable mit recodierten Werten stets als neues Objekt erstellt werden, statt die Werte des alten Objekts zu überschreiben.

Logische Indexvektoren bieten eine von mehreren Methoden, um systematisch bestimmte Werte durch andere zu ersetzen. In einem Vektor seien dazu die Lieblingsfarben von sieben englischsprachigen VPn erhoben worden. Später soll die Variable auf deutsche Farbnamen recodiert werden.

```
> myColors     <- c("red", "purple", "blue", "blue", "orange", "red", "orange")
> farben       <- character(length(myColors))            # neuen Vektor erstellen
> farben[myColors == "red"]                 <- "rot"
> farben[myColors == "purple"]              <- "violett"
> farben[myColors == "blue"]                <- "blau"
> farben[myColors == "orange"]              <- "orange"
> farben
[1] "rot" "violett" "blau" "blau" "orange" "rot" "orange"
```

Mit `replace()` können Werte eines Vektors ebenfalls über Indexvektoren ausgetauscht werden.

```
> replace(x=⟨Vektor⟩, list=⟨Indexvektor⟩, values=⟨neue Werte⟩)
```

Der Vektor mit den auszutauschenden Elementen ist unter `x` zu nennen. Welche Werte geändert werden sollen, gibt der Indexvektor `list` über numerische oder logische Indizes an. Der Vektor `values` definiert für jeden in `list` genannten Index mit je einem Element, welcher Wert an dieser Stelle neu einzufügen ist. Da `replace()` den unter `x` angegebenen Vektor nicht verändert, muss das Ergebnis ggf. einem neuen Objekt zugewiesen werden.

```
> replace(c(1, 2, 3, 4, 5), list=c(2, 4), values=c(200, 400))
[1] 1 200 3 400 5
```

Die Funktion `recode()` aus dem `car` Paket ermöglicht es auf bequemere Weise, gleichzeitig viele Werte nach einem Muster zu ändern, ohne dabei selbst logische Indexvektoren bilden zu müssen.

```
> recode(var=⟨Vektor⟩, recodes="⟨Muster⟩")
```

Der Vektor mit zu ändernden Werten ist für `var` anzugeben. Die Recodierung erfolgt anhand eines Musters, das das Argument `recodes` bestimmt. Hierbei handelt es sich um eine besonders aufgebaute Zeichenkette: Sie besteht aus durch Semikolon getrennten Elementen, von denen jedes eine Zuordnung von alten und neuen Werten in der Form ⟨`alt`⟩=⟨`neu`⟩ definiert. ⟨`alt`⟩ nennt in Form eines Vektors Werte in `var`, die durch denselben neuen Wert zu ersetzen sind. Sie müssen in `var` nicht unbedingt auch tatsächlich vorkommen, wodurch sich auch potenzielle Wertebereiche austauschen lassen. ⟨`neu`⟩ ist der gemeinsame neue Wert für die unter ⟨`alt`⟩ genannten.

Bei alten und neuen Werten sind Zeichenketten jeweils in einfache Anführungszeichen '⟨`Zeichen`⟩' zu setzen, wenn `recodes` insgesamt in doppelten Anführungszeichen "⟨`Zeichen`⟩" steht. Statt einem konkreten alten Wert kann auch dem Schlüsselwort `else` ein neuer zugewiesen werden, der dann für alle nicht anderweitig umcodierten Werte gilt. Tauchen Werte von `var` nicht im Muster `recodes` auf, bleiben sie unverändert. Auch `recode()` verändert den Vektor mit auszutauschenden Werten selbst nicht, weshalb das Ergebnis ggf. einem neuen Objekt zugewiesen werden muss.

```
> library(car)                                    # für recode()
> recode(myColors, "'red'='rot'; 'blue'='blau'; 'purple'='violett'")
[1] "rot" "violett" "blau" "blau" "orange" "rot" "orange"

# Einteilung der Farben in Basisfarben und andere
> recode(myColors, "c('red', 'blue')='basic'; else='complex'")
[1] "basic" "complex" "basic" "basic" "complex" "basic" "complex"
```

Gilt es, Werte entsprechend einer dichotomen Entscheidung durch andere zu ersetzen, kann dies mit der `ifelse()` Funktion geschehen.

```
> ifelse(test=⟨logischer Ausdruck⟩, yes=⟨Wert⟩, no=⟨Wert⟩)
```

Für das Argument `test` muss ein Ausdruck angegeben werden, der sich zu einem logischen Wert auswerten lässt, der also WAHR (`TRUE`) oder FALSCH (`FALSE`) ist. Ist `test` WAHR, wird der unter `yes` eingetragene Wert zurückgegeben, andernfalls der unter `no` genannte. Ist `test` ein Vektor, wird jedes seiner Elemente daraufhin geprüft, ob es `TRUE` oder `FALSE` ist und ein Vektor mit den passenden, unter `yes` und `no` genannten Werten als Elementen zurückgegeben. Die Ausgabe hat also immer dieselbe Länge wie die von `test`.

Die Argumente `yes` und `no` können selbst Vektoren derselben Länge wie `test` sein – ist dann etwa das dritte Element von `test` gleich `TRUE`, wird als drittes Element des Ergebnisses das dritte Element von `yes` zurückgegeben, andernfalls das dritte Element von `no`. Indem für `yes` ebenfalls der in `test` zu prüfende Vektor eingesetzt wird, können so bestimmte Werte eines Vektors ausgetauscht, andere dagegen unverändert gelassen werden. Dies erlaubt es etwa, alle Werte größer einem Cutoff-Wert auf denselben Maximalwert zu setzen und die übrigen Werte beizubehalten.

```
> orgVec <- c(5, 9, 11, 8, 9, 3, 1, 13, 9, 12, 5, 12, 6, 3, 17, 5, 8, 7)
> cutoff <- 10
> (reVec <- ifelse(orgVec <= cutoff, orgVec, cutoff))
[1] 5 9 10 8 9 3 1 10 9 10 5 10 6 3 10 5 8 7
```

2.5.7 Kontinuierliche Variablen in Kategorien einteilen

Als Spezialfall des Recodierens können neue Variablen dadurch entstehen, dass der Wertebereich von ursprünglich kontinuierlichen Variablen in Klassen eingeteilt wird. Auf diese Weise lässt sich eine quantitative in eine kategoriale Variable umwandeln. Hier soll der IQ-Wert mehrerer VPn zu einer Klasseneinteilung genutzt werden.

```
> IQ      <- c(112, 103, 87, 86, 90, 101, 90, 89, 122, 103)
> IQcls   <- numeric(length(IQ))                      # neuen Vektor erstellen
> IQcls[IQ <= 100]                          <- 1      # Intervall bis inkl. 100
> IQcls[(IQ > 100) & (IQ <= 115)]           <- 2      # Intervall (100, 115]
> IQcls[IQ > 115]                           <- 3      # Intervall größer 115
> IQcls
[1] 2 2 1 1 1 2 1 1 3 2

# Klasseneinteilung der IQ-Werte mit recode()
> library(car)                                        # für recode()
> recode(IQ, "0:100=1; 101:115=2; else=3")
[1] 2 2 1 1 1 2 1 1 3 2

# Dichotomisierung mit ifelse()
> ifelse(IQ >= 100, "hi", "lo")
[1] "hi" "hi" "lo" "lo" "lo" "hi" "lo" "lo" "hi" "hi"
```

Besonders leicht lassen sich quantitative Variablen zudem mit der Funktion `cut()` diskretisieren (vgl. Abschn. 2.6.7). Hierdurch werden sie zu Faktoren, dem Gegenstand des folgenden Abschnitts.

2.6 Gruppierungsfaktoren

Die Klasse `factor` existiert, um die Eigenschaften kategorialer Variablen abzubilden. Sie eignet sich insbesondere für Gruppierungsfaktoren im versuchsplanerischen Sinn. Ein Objekt dieser Klasse nimmt die jeweilige Gruppenzugehörigkeit von Beobachtungsobjekten auf und enthält Informationen darüber, welche Stufen die Variable prinzipiell umfasst. Für den Gebrauch in inferenzstatistischen Analysefunktionen ist es wichtig, dass Gruppierungsvariablen auch tatsächlich die Klasse `factor` besitzen. Insbesondere bei numerisch codierter Gruppenzugehörigkeit besteht sonst die Gefahr der Verwechslung mit echten quantitativen Variablen, was etwa bei linearen Modellen (z. B. lineare Regression oder Varianzanalyse) unentdeckt bleiben und für falsche Ergebnisse sorgen könnte.

2.6.1 Ungeordnete Faktoren

Als Beispiel für eine Gruppenzugehörigkeit soll das qualitative Merkmal Geschlecht dienen, dessen Ausprägungen in einer Stichprobe zunächst als `character` Werte eingegeben und in einem Vektor gespeichert werden.

```
> sex <- c("m", "f", "f", "m", "m", "m", "f", "f")
```

Um den aus Zeichen bestehenden Vektor zu einem Gruppierungsfaktor mit zwei Ausprägungen zu machen, dient der Befehl `factor()`.

```
> factor(x=⟨Vektor⟩, levels=⟨Stufen⟩, labels=levels, exclude=NA)
```

Unter `x` ist der umzuwandelnde Vektor einzutragen. Welche Stufen der Faktor prinzipiell annehmen kann, bestimmt das Argument `levels`. In der Voreinstellung werden die Faktorstufen automatisch anhand der in `x` tatsächlich vorkommenden Werte mit `sort(unique(x))` bestimmt. Fehlende Werte werden nicht als eigene Faktorstufe gewertet, es sei denn es wird das Argument `exclude=NULL` gesetzt. An `exclude` übergebene Werte werden nämlich nicht als Stufe berücksichtigt, wenn der Faktor erstellt wird.

```
> (sexFac <- factor(sex))
[1] m f f m m m f f
Levels: f m
```

Da die in `x` gespeicherten empirischen Ausprägungen nicht notwendigerweise auch alle theoretisch möglichen Kategorien umfassen müssen, kann an `levels` auch ein Vektor mit allen möglichen Stufen übergeben werden.

```
# 2 und 5 kommen nicht vor, sollen aber mögliche Ausprägungen sein
> factor(c(1, 1, 3, 3, 4, 4), levels=1:5)
[1] 1 1 3 3 4 4
Levels: 1 2 3 4 5
```

Die Stufenbezeichnungen stimmen in der Voreinstellung mit den `levels` überein, sie können aber auch durch einen für das Argument `labels` übergebenen Vektor umbenannt werden. Dies könnte etwa sinnvoll sein, wenn die Faktorstufen in einem Vektor numerisch codiert sind, im Faktor aber inhaltlich aussagekräftigere Namen erhalten sollen.

```
> (sexNum <- rbinom(10, 1, 0.5))                    # 0=Mann, 1=Frau
[1] 0 1 1 0 1 0 0 0 1 1

> factor(sexNum, labels=c("man", "woman"))
[1] man woman woman man woman man man man woman woman
Levels: man woman
```

Wenn die Stufenbezeichnungen eines Faktors im Nachhinein geändert werden sollen, so kann dem von der Funktion `levels(`⟨`Faktor`⟩`)` ausgegebenen Vektor ein Vektor mit passend vielen neuen Namen zugewiesen werden.

```
> levels(sexFac) <- c("female", "male"); sexFac         # vorherige Stufen: f, m
[1] male female female male male male female female
Levels: female male
```

Die Anzahl der Stufen eines Faktors wird mit der Funktion `nlevels(`⟨`Faktor`⟩`)` ausgegeben; wie häufig jede Stufe vorkommt, erfährt man durch `summary(`⟨`Faktor`⟩`)`.

```
> nlevels(sexFac)
[1] 2

> summary(sexFac)
female   male
     4      4
```

Die im Faktor gespeicherten Werte werden intern auf zwei Arten repräsentiert – zum einen mit den Namen der Faktorstufen in einem `character` Vektor im Attribut `levels`, zum anderen mit einer internen Codierung der Stufen über fortlaufende natürliche Zahlen, die der (ggf. alphabetischen) Reihenfolge der Ausprägungen entspricht. Dies wird in der Ausgabe der internen Struktur eines Faktors mit `str(`⟨`Faktor`⟩`)` deutlich. Die Namen der Faktorstufen werden mit `levels(`⟨`Faktor`⟩`)` ausgegeben, die interne numerische Repräsentation mit `unclass(`⟨`Faktor`⟩`)`.[15]

```
> levels(sexFac)
[1] "female" "male"
```

[15] Trotz dieser Codierung können Faktoren keinen mathematischen Transformationen unterzogen werden. Wenn die Namen der Faktorstufen aus Zahlen gebildet werden, kann es zu Diskrepanzen zwischen Levels und interner Codierung kommen: `unclass(factor(10:15))` ergibt `1 2 3 4 5 6`. Dies ist bei der üblichen Verwendung von Faktoren aber irrelevant.

```
> str(sexFac)
Factor w/ 2 levels "female","male": 2 1 1 2 2 2 1 1

> unclass(sexFac)
[1] 2 1 1 2 2 2 1 1

attr(,"levels")
[1] "female" "male"
```

Bei der Umwandlung eines Faktors mit der `as.character(⟨Faktor⟩)` Funktion erhält der Ergebnisvektor des Datentyps `character` als Elemente die Namen der entsprechenden Faktorstufen. Sind die Namen aus Zahlen gebildet und sollen letztlich in numerische Werte umgewandelt werden, so ist dies durch einen zusätzlichen Schritt mit `as.numeric(as.character(⟨Faktor⟩))` oder mit `as.numeric(levels(⟨Faktor⟩)[⟨Faktor⟩])` möglich.[16]

```
> as.character(sexFac)
[1] "male" "female" "female" "male" "male" "male" "female" "female"
```

2.6.2 Faktoren kombinieren

Faktoren lassen sich nicht wie Vektoren mit `c()` aneinanderhängen, da `c()` die Attribute der übergebenen Argumente (bis auf die Elementnamen) entfernt. Im Fall von Faktoren wären deren Klasse `factor` und die Stufen `levels` davon betroffen. Stattdessen muss ein komplizierterer Weg gewählt werden: Zunächst sind aus den zu kombinierenden Faktoren alle Ausprägungen in Form von `character` Vektoren zu extrahieren und diese dann mit `c()` zu verbinden, bevor aus dem kombinierten Vektor wieder ein Faktor erstellt wird. Dabei gehen allerdings mögliche Faktorstufen verloren, die nicht auch als Ausprägung tatsächlich vorkommen.

```
> (fac1 <- factor(sample(LETTERS, 5)))              # Faktor 1
[1] Q A P U J
Levels: A J P Q U

> (fac2 <- factor(sample(letters, 3)))              # Faktor 2
[1] z g t
Levels: g t z

> (charVec1 <- levels(fac1)[fac1])                  # character Vektor zu Faktor 1
[1] "Q" "A" "P" "U" "J"

> (charVec2 <- levels(fac2)[fac2])                  # character Vektor zu Faktor 2
[1] "z" "g" "t"

# Faktor der aneinandergehängten character Vektoren
> factor(c(charVec1, charVec2))
```

[16] Sind die Namen der Faktorstufen dagegen nicht aus Zahlen, sondern aus anderen Zeichen gebildet, ist das Ergebnis `NA` (vgl. Abschn. 1.3.5).

```
[1] Q A P U J z g t
Levels: A g J P Q t U z
```

Gilt es, lediglich einen bereits bestehenden Faktor zu vervielfachen, eignet sich dagegen wie bei Vektoren `rep()`.

```
> rep(fac1, times=2)                    # Faktor 1 wiederholen
[1] Q A P U J Q A P U J
Levels: A J P Q U
```

In Situationen, in denen sich experimentelle Bedingungen aus der Kombination von zwei oder mehr Faktoren ergeben, ist es bisweilen nützlich, die mehrfaktorielle in eine geeignete einfaktorielle Struktur zu überführen. Dabei werden alle Kombinationen von Faktorstufen als Stufen eines neuen einzelnen Faktors betrachtet – etwa im Kontext einer assoziierten einfaktoriellen Varianzanalyse bei einem eigentlich zweifaktoriellen Design (vgl. Abschn. 7.5). Dies ist mit `interaction()` möglich.

```
> interaction(⟨Faktor1⟩, ⟨Faktor2⟩, ..., drop=FALSE)
```

Als Argument sind zunächst mehrere Faktoren gleicher Länge zu übergeben, die die Gruppenzugehörigkeit derselben Beobachtungsobjekte bzgl. verschiedener Gruppierungsfaktoren codieren. Auf ihren Stufenkombinationen basieren die Ausprägungen des neuen Faktors. Dabei kann es vorkommen, dass nicht für jede Stufenkombination Beobachtungen vorhanden sind, da einige Zellen im Versuchsdesign leer sind, oder kein vollständig gekreuztes Design vorliegt. In der Voreinstellung `drop=FALSE` erhält der neue Faktor auch für leere Zellen eine passende Faktorstufe. Mit `drop=TRUE` werden zu leeren Zellen gehörende Stufen dagegen weggelassen.

```
> (IV1 <- factor(rep(c("lo", "hi"), each=6)))     # Faktor 1
[1] lo lo lo lo lo lo hi hi hi hi hi hi
Levels: hi lo

> (IV2 <- factor(rep(1:3, times=4)))              # Faktor 2
[1] 1 2 3 1 2 3 1 2 3 1 2 3
Levels: 1 2 3

> interaction(IV1, IV2)      # assoziiertes einfaktorielles Design
[1] lo.1 lo.2 lo.3 lo.1 lo.2 lo.3 hi.1 hi.2 hi.3 hi.1 hi.2 hi.3
Levels: hi.1 lo.1 hi.2 lo.2 hi.3 lo.3
```

2.6.3 Faktorstufen nachträglich ändern

Einem bestehenden Faktor können nicht beliebige Werte als Element hinzugefügt werden, sondern lediglich solche, die einer bereits existierenden Faktorstufe entsprechen. Bei Versuchen, einem Faktorelement einen anderen Wert zuzuweisen, wird das Element auf `NA` gesetzt und eine Warnung ausgegeben. Die Menge möglicher Faktorstufen kann jedoch über `levels()` erweitert werden, ohne dass es bereits zugehörige Beobachtungen gäbe.

```
> (status <- factor(c("hi", "lo", "hi")))
[1] hi lo hi
Levels: hi lo

# bisher nicht existierende Faktorstufe "mid"
> status[4] <- "mid"; status
Warning message:
In `[<-.factor`(`*tmp*`, 4, value = "mid") :
invalid factor level, NAs generated

[1] hi lo hi <NA>
Levels: hi lo

> levels(status) <- c(levels(status), "mid")              # Stufe "mid" hinzufügen
> status[4] <- "mid"; status                              # neue Beobachtung "mid"
[1] hi lo hi mid
Levels: hi lo mid
```

Stufen eines bestehenden Faktors lassen sich nicht ohne Weiteres löschen. Die erste Möglichkeit, um einen gegebenen Faktor in einen Faktor mit weniger möglichen Stufen umzuwandeln, besteht im Zusammenfassen mehrerer ursprünglicher Stufen zu einer gemeinsamen neuen Stufe. Hierzu muss dem von `levels()` ausgegebenen Objekt eine Liste zugewiesen werden, die nach dem Muster `list(⟨neueStufe⟩=c("⟨alteStufe1⟩", "⟨alteStufe2⟩", ...))` aufgebaut ist (vgl. Abschn. 3.1). Alternativ eignet sich die in Abschn. 2.5.6 vorgestellte `recode()` Funktion aus dem Paket `car`, mit der sich numerische Vektoren und Faktoren gleichermaßen umcodieren lassen.

```
# kombiniere Stufen "mid" und "lo" zu "notHi", "hi" bleibt unverändert
> hiNotHi <- status                                       # Kopie des Faktors
> levels(hiNotHi) <- list(hi="hi", notHi=c("mid", "lo")); hiNotHi
[1] hi notHi hi notHi
Levels: hi notHi

> library(car)                                            # für recode()
> (statNew <- recode(status, "'hi'='high'; c('mid', 'lo')='notHigh'"))
[1] high notHigh high notHigh
Levels: high notHigh
```

Sollen dagegen Beobachtungen samt ihrer Stufen gelöscht werden, muss eine Teilmenge der Elemente des Faktors ausgegeben werden, die nicht alle Faktorstufen enthält. Zunächst umfasst diese Teilmenge jedoch nach wie vor alle ursprünglichen Stufen, wie in der Ausgabe unter `Levels` deutlich wird. Sollen nur die in der gewählten Teilmenge tatsächlich auftretenden Ausprägungen auch mögliche `Levels` sein, kann dies mit der Funktion `droplevels()` erreicht werden.

```
> status[1:2]
[1] hi lo
Levels: hi lo mid

> (newStatus <- droplevels(status[1:2]))
[1] hi lo
Levels: hi lo
```

2.6.4 Geordnete Faktoren

Besteht eine Reihenfolge in den Stufen eines Gruppierungsfaktors i. S. einer ordinalen Variable, so lässt sich dieser Umstand mit der Funktion `ordered(⟨Vektor⟩,↘ → levels)` abbilden, die einen geordneten Gruppierungsfaktor erstellt. Dabei muss die inhaltliche Reihenfolge im Argument `levels` explizit angegeben werden, weil R sonst die Reihenfolge selbst bestimmt und ggf. die alphabetische heranzieht. Für die Elemente geordneter Faktoren sind auch die üblichen Ordnungsrelationen $<, \leq, >, \geq$ definiert.

```
> (ordStat <- ordered(status, levels=c("lo", "mid", "hi")))
[1] hi  lo  hi  mid
Levels: lo < mid < hi

> ordStat[1] > ordStat[2]                          # hi > lo?
[1] TRUE
```

Manche Funktionen zur inferenzstatistischen Analyse nehmen bei geordneten Faktoren Gleichabständigkeit in dem Sinne an, dass die inhaltliche Unterschiedlichkeit zwischen zwei benachbarten Stufen immer dieselbe ist. Trifft dies nicht zu, sollte im Zweifel auf ungeordnete Faktoren zurückgegriffen werden.

2.6.5 Reihenfolge von Faktorstufen

Beim Sortieren von Faktoren wie auch in manchen statistischen Analysefunktionen ist die Reihenfolge der Faktorstufen bedeutsam. Werden die Faktorstufen beim Erstellen eines Faktors explizit mit `labels` angegeben, bestimmt die Reihenfolge der Elemente in `labels` die Reihenfolge der Stufen. Ohne Verwendung von `labels` ergibt sich die Reihenfolge aus den sortierten Elementen des Vektors, der zum Erstellen des Faktors verwendet wird.[17]

Um die Reihenfolge der Stufen nachträglich zu ändern, kann ein Faktor in einen geordneten Faktor unter Verwendung des `levels` Arguments konvertiert werden (s. o.). Als weitere Möglichkeit wird mit `relevel(⟨Faktor⟩, ref="⟨↘ →Referenzstufe⟩")` die für `ref` übergebene Faktorstufe zur ersten Stufe des Faktors. Die `reorder()` Funktion ändert die Reihenfolge der Faktorstufen ebenfalls nachträglich. Die Stufen werden so geordnet, dass ihre Reihenfolge durch die gruppenweise gebildeten empirischen Kennwerte einer Variable bestimmt ist.

```
> reorder(x=⟨Faktor⟩, X=⟨Vektor⟩, FUN=⟨Funktion⟩)
```

Als erstes Argument `x` wird der Faktor mit den zu ordnenden Stufen erwartet. Für das Argument `X` ist ein numerischer Vektor derselben Länge wie `x` zu übergeben, der auf Basis von `x` in Gruppen eingeteilt wird. Pro Gruppe wird die mit `FUN` bezeichnete

[17] Sind dessen Elemente Zeichenketten mit numerischer Bedeutung, so ist zu beachten, dass die Reihenfolge dennoch alphabetisch bestimmt wird – die Stufe `"10"` käme demnach vor der Stufe `"4"`.

Funktion angewendet, die einen Vektor zu einem skalaren Kennwert verarbeiten muss. Als Ergebnis werden die Stufen von `x` entsprechend der Kennwerte geordnet, die sich aus der gruppenweisen Anwendung von `FUN` ergeben (vgl. Abschn. 2.7.8 für die Funktion `tapply()`).

```
> fac1    <- factor(rep(LETTERS[1:3], each=10))
> vec     <- rnorm(30, rep(c(10, 5, 15), each=10), 3)
> tapply(vec, fac1, mean)               # Mittelwerte pro Gruppe
       A        B        C
10.18135  6.47932  13.50108

> reorder(fac1, vec, mean)
 [1] A A A A A A A A A A B B B B B B B B B B C C
[23] C C C C C C C C
Levels: B < A < C
```

Beim Sortieren von Faktoren wird die Reihenfolge der Elemente durch die Reihenfolge der Faktorstufen bestimmt, die nicht mit der numerischen oder alphabetischen Reihenfolge der Stufenbezeichnungen übereinstimmen muss. Damit kann das Sortieren eines Faktors zu einem anderen Ergebnis führen als das Sortieren eines Vektors, auch wenn diese oberflächlich dieselben Elemente enthalten.

```
> (fac2 <- factor(sample(1:2, 10, replace=TRUE), labels=c("B", "A")))
[1] A A A B B A B B B B
Levels: B A

> sort(fac2)
[1] B B B B B B A A A A

> sort(as.character(fac2))
[1] "A" "A" "A" "A" "B" "B" "B" "B" "B" "B"
```

2.6.6 Faktoren nach Muster erstellen

Da Faktoren im Zuge der Einteilung von Beobachtungsobjekten in Gruppen oft nach festem Muster erstellt werden müssen, lassen sich als Alternative zur manuellen Anwendung von `rep()` und `factor()` mit der `gl()` Funktion Faktoren automatisiert erzeugen.

```
> gl(n=⟨Stufen⟩, k=⟨Zellbesetzung⟩, labels=1:n, ordered=FALSE)
```

Das Argument `n` gibt die Anzahl der Stufen an, die der Faktor besitzen soll. Mit `k` wird festgelegt, wie häufig jede Faktorstufe realisiert werden soll, wie viele Beobachtungen also jede Bedingung umfasst. Für `labels` kann ein Vektor mit so vielen Gruppenbezeichnungen angegeben werden, wie Stufen vorhanden sind. In der Voreinstellung werden die Gruppen nummeriert. Um einen geordneten Faktor zu erstellen, ist `ordered=TRUE` zu setzen.

```
> (fac1 <- factor(rep(c("A", "B"), c(5, 5))))              # manuell
[1] A A A A A B B B B B
Levels: A B

> (fac2 <- gl(2, 5, labels=c("less", "more"), ordered=TRUE))
[1] less less less less less more more more more more
Levels: less < more
```

Sollen die Elemente des Faktors in eine zufällige Reihenfolge gebracht werden, um die Zuordnung von VPn zu Gruppen zu randomisieren, kann dies wie in Abschn. 2.5.2 beschrieben geschehen.

```
> sample(fac2, length(fac2), replace=FALSE)
[1] more more less more less less less more more less
Levels: less < more
```

Bei mehreren UVn mit vollständig gekreuzten Faktorstufen kann die Funktion `expand.grid()` verwendet werden, um alle Stufenkombinationen zu erstellen (vgl. Abschn. 2.3.3). Dabei ist die angestrebte Gruppenbesetzung pro Zelle nur bei einem der hier im Aufruf durch `gl()` erstellten Faktoren anzugeben, beim anderen ist sie auf 1 zu setzen. Das Ergebnis ist ein Datensatz (vgl. Abschn. 3.2).

```
> expand.grid(IV1=gl(2, 2, labels=c("a", "b")), IV2=gl(3, 1))
   IV1  IV2
1    a    1
2    a    1
3    b    1
4    b    1
5    a    2
6    a    2
7    b    2
8    b    2
9    a    3
10   a    3
11   b    3
12   b    3
```

2.6.7 Quantitative in kategoriale Variablen umwandeln

Aus den in einem Vektor gespeicherten Werten einer quantitativen Variable lässt sich mit `cut()` ein Gruppierungsfaktor erstellen – die quantitative Variable wird so in eine kategoriale umgewandelt (vgl. auch Abschn. 2.5.7). Dazu muss zunächst der Wertebereich des Vektors in disjunkte Intervalle eingeteilt werden. Die einzelnen Werte werden dann entsprechend ihrer Zugehörigkeit zu diesen Intervallen Kategorien zugeordnet und erhalten als Wert die zugehörige Faktorstufe.

```
> cut(x=⟨Vektor⟩, breaks=⟨Intervalle⟩, labels=⟨Bezeichnungen⟩, ordered=FALSE)
```

Die Intervalle werden über das Argument `breaks` festgelegt. Hier ist entweder die Anzahl der (dann gleich breiten) Intervalle anzugeben, oder die Intervallgrenzen

selbst als Vektor. Die Intervalle werden in der Form $(a, b\,]$ gebildet, sind also nach unten offen und nach oben geschlossen. Anders gesagt ist die untere Grenze nicht Teil des Intervalls, die obere schon. Die Unter- und Obergrenze des insgesamt möglichen Wertebereichs müssen bei der Angabe von `breaks` berücksichtigt werden, ggf. sind dies `-Inf` und `Inf` für negativ und positiv unendliche Werte.

Wenn die Faktorstufen andere Namen als die zugehörigen Intervallgrenzen tragen sollen, können sie über das Argument `labels` explizit angegeben werden. Dabei ist darauf zu achten, dass die Reihenfolge der neuen Benennungen der Reihenfolge der ursprünglichen Stufen entspricht. Soll es sich im Ergebnis um einen geordneten Faktor handeln, ist `ordered=TRUE` zu setzen.

```
> IQ       <- rnorm(100, mean=100, sd=15)
> IQfac    <- cut(IQ, breaks=c(0, 85, 115, Inf), labels=c("lo", "mid", "hi"))
> IQfac[1:5]
[1] hi lo mid mid mid lo
Levels: lo mid hi
```

Sollen die entstehenden Gruppen annähernd gleich groß sein, können für die Intervallgrenzen bestimmte Quantile der Daten gewählt werden, etwa der Median für den Median-Split (vgl. Abschn. 2.7.3).

```
> medSplit <- cut(IQ, breaks=c(-Inf, median(IQ), Inf))
> summary(medSplit)                # Kontrolle: Häufigkeiten der Kategorien
medSplit
(-Inf,97.6]  (97.6,Inf]
         50          50
```

Für mehr als zwei etwa gleich große Gruppen lässt sich die Ausgabe von `quantile()` (vgl. Abschn. 2.7.5) direkt an das Argument `breaks` übergeben. Dies ist möglich, da `quantile()` neben den Quantilen auch das Minimum und Maximum der Werte ausgibt. Damit das unterste Intervall auch das Minimum einschließt – und nicht wie alle übrigen Intervalle nach unten offen ist, muss das zusätzliche Argument `include.lowest=TRUE` gesetzt werden.

```
# 4 ähnliche große Gruppen, unterstes Intervall dabei nach unten geschlossen
> IQdiscr <- cut(IQ, quantile(IQ), include.lowest=TRUE)
> summary(IQdiscr)                 # Kontrolle: Häufigkeiten der Kategorien
IQdiscr
[62.1,87.1] (87.1,97.6]  (97.6,107]   (107,154]
         25          25          25          25
```

2.7 Deskriptive Kennwerte numerischer Daten

Die deskriptive Beschreibung von Variablen ist ein wichtiger Teil der Analyse empirischer Daten, die gemeinsam mit der grafischen Darstellung (vgl. Kap. 11) hilft, deren Struktur besser zu verstehen. Die hier umgesetzten statistischen Konzepte und Techniken werden als bekannt vorausgesetzt und finden sich in vielen Lehrbüchern der Statistik (Eid, Gollwitzer & Schmitt, 2010; Hays, 1994).

R stellt für die Berechnung aller gängigen Kennwerte separate Funktionen bereit, die meist erwarten, dass die Daten in Vektoren gespeichert sind. Es sei an dieser Stelle daran erinnert, dass sich logische Wahrheitswerte ebenfalls in einem numerischen Kontext verwenden lassen, wobei der Wert `TRUE` wie eine 1, der Wert `FALSE` wie eine 0 behandelt wird.

Mit der Funktion `summary(⟨Vektor⟩)` können die wichtigsten deskriptiven Kennwerte einer Datenreihe abgerufen werden – dies sind Minimum, erstes Quartil, Median, Mittelwert, drittes Quartil und Maximum. Die Ausgabe ist ein Vektor mit benannten Elementen.

```
> age <- c(17, 30, 30, 25, 23, 21)
> summary(age)
 Min.  1st Qu.  Median   Mean  3rd Qu.   Max.
17.00    21.50   24.00  24.33    28.75  30.00
```

2.7.1 Summen, Differenzen und Produkte

Mit der `sum(⟨Vektor⟩)` Funktion wird die Summe aller Elemente eines Vektors berechnet. Die kumulierte Summe erhält man mit `cumsum(⟨Vektor⟩)`.

```
> sum(age)
[1] 146

> cumsum(age)
[1] 17 47 77 102 125 146
```

Um die Differenzen aufeinander folgender Elemente eines Vektors (also eines Wertes zu einem Vorgänger) zu berechnen, kann die Funktion `diff(⟨Vektor⟩, lag=1)` verwendet werden. Mit ihrem Argument `lag` wird kontrolliert, über welchen Abstand die Differenz gebildet wird. Die Voreinstellung `1` bewirkt, dass die Differenz eines Wertes zum unmittelbar vorhergehenden berechnet wird. Die Ausgabe umfasst `lag` Werte weniger, als der Vektor Elemente besitzt.

```
> diff(age)
[1] 13 0 -5 -2 -2

> diff(age, lag=2)
[1] 13 -5 -7 -4
```

Das Produkt aller Elemente eines Vektors wird mit `prod(⟨Vektor⟩)` berechnet, das kumulierte Produkt mit `cumprod(⟨Vektor⟩)`.[18] `factorial(⟨Zahl⟩)` ermittelt

[18] Bei Gleitkommazahlen begünstigt die wiederholte Multiplikation die Kumulation von Rundungsfehlern, die durch die interne Darstellungsart solcher Zahlen unvermeidlich sind (vgl. Abschn. 1.3.6). Numerisch stabiler als `prod(⟨Vektor⟩)` ist deswegen u. U. die Rücktransformation der Summe der logarithmierten Werte mit `exp(sum(log(⟨Vektor⟩)))` als Umsetzung von $e^{\sum_i \ln x_i}$.

die Fakultät $n!$ einer Zahl n, ist für natürliche Zahlen also identisch zu `prod(1:⟨Zahl⟩)`.[19]

```
> prod(age)
[1] 184747500

> cumprod(age)
[1] 17 510 15300 382500 8797500 184747500

> factorial(5)
[1] 120
```

2.7.2 Extremwerte

Mit `min(⟨Vektor⟩)` und `max(⟨Vektor⟩)` können die Extremwerte eines Vektors erfragt werden. `range(⟨Vektor⟩)` gibt den größten und kleinsten Wert zusammengefasst als Vektor aus. Den Index des größten bzw. kleinsten Wertes liefern die Funktionen `which.max(⟨Vektor⟩)` bzw. `which.min(⟨Vektor⟩)`. Die letztgenannte Funktion lässt sich etwa nutzen, um herauszufinden, welches Element eines Vektors am nächsten an einem vorgegebenen Wert liegt.

```
> max(age)                          # Maximum
[1] 30

> which.min(age)                    # Position des Minimums
[1] 1

> vec <- c(-5, -8, -2, 10, 9)       # Vektor
> val <- 0                          # Referenzwert
> which.min(abs(vec-val))           # welches Element liegt am nächsten an 0?
[1] 3

> range(c(17, 30, 30, 25, 23, 21))
[1] 17 30
```

Um die Spannweite (range) von Werten eines Vektors, also die Differenz von kleinstem und größtem Wert zu ermitteln, ist die `diff()` Funktion nützlich.

```
> diff(range(c(17, 30, 30, 25, 23, 21)))
[1] 13
```

Die Funktionen `pmin(⟨Vektor1⟩, ⟨Vektor2⟩, ...)` und `pmax(⟨Vektor1⟩, ⟨Vektor2⟩, ...)` (parallel min/max) vergleichen zwei oder mehr Vektoren elementweise hinsichtlich der Größe der in ihnen gespeicherten Werte. Sie liefern einen Vektor aus den pro Position größten bzw. kleinsten Werten zurück.

[19] Genauso gilt für natürliche Zahlen $n! = \Gamma(n+1)$, in R als `gamma(⟨Zahl⟩ + 1)` berechenbar. Wegen $0! = 1$ erzeugt `prod(numeric(0))` ebenso wie `factorial(numeric(0))` das Ergebnis `1`.

```
> vec1 <- c(5, 2, 0, 7)
> vec2 <- c(3, 3, 9, 2)
> pmax(vec1, vec2)
[1] 5 3 9 7

> pmin(vec1, vec2)
[1] 3 2 0 2
```

2.7.3 Mittelwert, Median und Modalwert

Mit der Funktion `mean(x=⟨Vektor⟩)` wird das arithmetische Mittel $\frac{1}{n} \cdot \sum_i x_i$ eines Vektors x der Länge n berechnet,[20] was manuell auch mit `sum(x) / length(x)` möglich ist.

```
> age <- c(17, 30, 30, 25, 23, 21)
> mean(age)
[1] 24.33333
```

Für die Berechnung eines gewichteten Mittels, bei dem die Gewichte nicht wie bei `mean()` für alle Werte identisch sind ($\frac{1}{n}$), eignet sich die Funktion `weighted.mean(x=⟨Vektor⟩, w=⟨Gewichte⟩)`. Ihr zweites Argument `w` muss ein Vektor derselben Länge wie `x` sein und die Gewichte benennen. Der Einfluss jedes Wertes auf den Mittelwert ist gleich dem Verhältnis seines Gewichts zur Summe aller Gewichte.

```
> weights <- c(0.6, 0.6, 0.3, 0.2, 0.4, 0.6)
> weighted.mean(age, weights)
[1] 23.70370
```

Um beim geometrischen Mittel $\frac{1}{n} \cdot \prod_i x_i$ sich fortsetzende Rundungsfehler zu vermeiden, eignet sich `exp(mean(log(x)))` als Umsetzung von $e^{\frac{1}{n} \cdot \sum_i \ln x_i}$ (vgl. Fußnote 18). Das harmonische Mittel $n / \sum_i \frac{1}{x_i}$ berechnet sich als Kehrwert des Mittelwertes der Kehrwerte, also mit `1 / mean(1/x)`. Alternativ stellt das `psych` Paket (Revelle, 2011) die Funktionen `geometric.mean()` und `harmonic.mean()` bereit.

Die Funktion `median(x=⟨Vektor⟩)` gibt den Median, also das 50 %-Quantil einer empirischen Verteilung aus. Dies ist der Wert, für den die empirische kumulative Häufigkeitsverteilung von `x` erstmalig mindestens den Wert 0.5 erreicht (vgl. Abschn. 2.10.5), der also $\geq$ 50 % (und $\leq$ 50 %) der Werte ist. Im Fall einer geraden Anzahl von Elementen in `x` wird zwischen den mittleren beiden Werten von `sort(⟨Vektor⟩)` gemittelt, andernfalls das mittlere Element von `sort(⟨Vektor⟩)` ausgegeben.

[20] Hier ist zu beachten, dass `x` tatsächlich ein etwa mit `c(...)` gebildeter Vektor ist: Der Aufruf `mean(1, 7, 3)` gibt nämlich anders als `mean(c(1, 7, 3))` nicht den Mittelwert der Daten 1, 7, 3 aus. Stattdessen ist die Ausgabe gleich dem ersten übergebenen Argument.

```
> sort(age)
[1] 17 21 23 25 30 30

> median(age)
[1] 24

> ageNew <- c(age, 22)
> sort(ageNew)
[1] 17 21 22 23 25 30 30

> median(ageNew)
[1] 23
```

Für die Berechnung des Modalwertes, also des am häufigsten vorkommende Wertes eines Vektors, stellt R keine separate Funktion bereit. Eine Möglichkeit, um den Modalwert auf andere Weise zu erhalten, bietet die `table()` Funktion zur Erstellung von Häufigkeitstabellen (vgl. Abschn. 2.10). Die folgende Methode gibt zunächst den Index des Maximums der Häufigkeitstabelle aus. Den Modalwert erhält man zusammen mit seiner Auftretenshäufigkeit durch Indizieren der mit `unique()` gebildeten Einzelwerte des Vektors mit diesem Index.

```
> vec   <- c(11, 22, 22, 33, 33, 33, 33)       # häufigster Wert: 33
> (tab  <- table(vec))                         # Häufigkeitstabelle
vec
11  22  33
 1   2   4

> (modIdx <- which.max(tab))                   # Index des Modalwerts
33
 3

> unique(vec)[modIdx]                          # Modalwert selbst
[1] 33
```

2.7.4 Robuste Maße der zentralen Tendenz

Wenn Daten Ausreißer aufweisen, kann dies den Mittelwert stark verzerren, sodass er die zentrale Tendenz der Daten nicht mehr gut repräsentiert. Aus diesem Grund existieren Maße der zentralen Tendenz von Daten, die weniger stark durch einzelne extreme Werte beeinflusst werden.

Ein Beispiel ist der gestutzte Mittelwert, der ohne einen bestimmten Anteil an Extremwerten berechnet wird. Mit dem Argument `mean(x, trim=⟨Zahl⟩)` wird der gewünschte Anteil an Extremwerten aus der Berechnung des Mittelwerts ausgeschlossen. ⟨`Zahl`⟩ gibt dabei den Anteil der Werte an, der auf jeder der beiden Seiten der empirischen Verteilung verworfen werden soll. Um etwa den Mittelwert ohne die extremen 5 % der Daten zu berechnen, ist also `trim=0.025` zu setzen.

Bei der Winsorisierung von Daten wird ein bestimmter Anteil an Extremwerten auf beiden Seiten der Verteilung durch jeweils den letzten Wert ersetzt, der noch

nicht als Extremwert gilt. Vergleiche dafür die Funktion `winsor()` aus dem Paket `psych`. Der übliche Mittelwert dieser Daten ist dann der winsorisierte Mittelwert, für die winsorisierte Varianz gilt dies analog.

Der Hodges-Lehmann-Schätzer des Lageparameters einer Variable berechnet sich als Median der $\frac{n(n+1)}{2}$ Mittelwerte aller Wertepaare (x_i, x_j) der Daten (mit $j \geq i$) mit sich selbst. Vergleiche hierfür die Funktion `hl.loc()` aus dem Paket `ICSNP` (Nordhausen, Sirkia, Oja & Tyler, 2010) sowie `wilcox.test()` in Abschn. 8.5.3. Als Schätzer für die Differenz der Lageparameter von zwei Variablen wird der Hodges-Lehmann-Schätzer als Median aller $n_1 \cdot n_2$ paarweisen Differenzen der Werte aus beiden Datenreihen gebildet (vgl. Abschn. 8.5.4).

2.7.5 Prozentrang, Quartile, Quantile und Interquartilabstand

Angewendet auf Vektoren mit logischen Werten lassen sich mit `sum()` Elemente zählen, die eine bestimmte Bedingung erfüllen. So kann der Prozentrang eines Wertes als prozentualer Anteil derjenigen Werte ermittelt werden, die nicht größer als er sind (vgl. auch Abschn. 2.10.5).

```
> age <- c(17, 30, 30, 25, 23, 21)
> sum(age <= 25)                                  # wie viele Elemente <= 25?
[1] 4

> 100 * (sum(age <= 25) / length(age))            # Prozentrang von 25
[1] 66.66667
```

Mit der Funktion `quantile(x=⟨Vektor⟩, probs=seq(0, 1, 0.25))` werden in der Voreinstellung die Quartile eines Vektors bestimmt. Dies sind jene Werte, die größer oder gleich einem ganzzahligen Vielfachen von 25 % der Datenwerte und kleiner oder gleich den Werten des verbleibenden Anteils der Daten sind. Das erste Quartil ist $\geq 25\,\%$ (und $\leq 75\,\%$) der Daten, das zweite Quartil (der Median) $\geq 50\,\%$ (und $\leq 50\,\%$) und das dritte Quartil $\geq 75\,\%$ (und $\leq 25\,\%$) der Werte. Das Ergebnis von `quantile()` ist ein Vektor mit benannten Elementen.

```
> (quant <- quantile(age))
   0%   25%   50%   75%  100%
17.00 21.50 24.00 28.75 30.00

> quant[c("25%", "50%")]
 25%  50%
21.5 24.0
```

Über das `probs=⟨Vektor⟩` Argument können statt der Quartile auch andere Anteile eingegeben werden, deren Wertegrenzen gewünscht sind. Zur Berechnung der Werte, die einen bestimmten Anteil der Daten abschneiden, wird ggf. zwischen den in `x` tatsächlich vorkommenden Werten interpoliert.[21]

[21] Zur Berechnung von Quantilen stehen verschiedene Rechenwege zur Verfügung, vgl. `?quantile`.

```
> vec <- sample(seq(0, 1, by=0.01), 1000, replace=TRUE)
> quantile(vec, probs=c(0, 0.2, 0.4, 0.6, 0.8, 1))
   0%      20%      40%      60%      80%     100%
0.000    0.190    0.400    0.604    0.832    1.000
```

Mit der Funktion `IQR(⟨Vektor⟩)` wird der Interquartilabstand erfragt, also die Differenz von drittem und erstem Quartil.

```
> IQR(age)
[1] 7.25
```

2.7.6 Varianz, Streuung, Schiefe und Wölbung

Mit der Funktion `var(x=⟨Vektor⟩)` wird die korrigierte Varianz $s^2 = \frac{1}{n-1} \cdot \sum_i (x_i - M)^2$ zur erwartungstreuen Schätzung der Populationsvarianz auf Basis einer Variable X der Länge n mit Mittelwert M ermittelt.[22] Die Umrechnungsformel zur Berechnung der unkorrigierten Varianz $S^2 = \frac{1}{n} \cdot \sum_i (x_i - M)^2$ aus der korrigierten lautet $S^2 = \frac{n-1}{n} \cdot s^2$.[23]

```
> age    <- c(17, 30, 30, 25, 23, 21)          # Daten
> N      <- length(age)                        # Anzahl Beobachtungen
> M      <- mean(age)                          # Mittelwert

> var(age)                                     # korrigierte Varianz
[1] 26.26667

> sum((age-M)^2) / (N-1)                       # manuelle Berechnung
[1] 26.26667

> ((N-1) / N) * var(age)                       # unkorrigierte Varianz
[1] 21.88889

> sum((age-M)^2) / N                           # manuelle Berechnung
[1] 21.88889
```

Die korrigierte Streuung s kann durch Ziehen der Wurzel aus der korrigierten Varianz oder mit der `sd(x=⟨Vektor⟩)` Funktion berechnet werden. Auch hier basiert das Ergebnis auf der bei der Varianz erläuterten Korrektur zur Schätzung der Populationsstreuung auf Basis einer empirischen Stichprobe. Die Umrechnungsformel zur Berechnung der unkorrigierten Streuung $S = \sqrt{\frac{1}{n} \cdot \sum_i (x_i - M)^2}$ aus der korrigierten lautet $S = \sqrt{\frac{n-1}{n}} \cdot s$.

```
> sqrt(var(age))                        # Wurzel aus Varianz
[1] 5.125102
```

[22] Für die Berechnung der winsorisierten Varianz vgl. Abschn. 2.7.4. Für ein Diversitätsmaß kategorialer Daten vgl. Abschn. 2.10.6.

[23] Als Alternative ließe sich die `cov.wt()` Funktion verwenden (vgl. Abschn. 2.8.10).

```
> sd(age)                                  # korrigierte Streuung
[1] 5.125102

> sqrt((N-1) / N) * sd(age)                # unkorrigierte Streuung
[1] 4.678556

> sqrt(sum((age-M)^2) / N)                 # manuelle Berechnung
[1] 4.678556
```

Die mittlere absolute Abweichung vom Mittelwert oder Median ist manuell zu berechnen, während für den Median der absoluten Abweichungen vom Median die Funktion `mad(⟨Vektor⟩)` bereit steht.[24]

```
> mean(abs(age-M))                         # mittlere abs. Abweichung vom Mittelwert
[1] 4

> mean(abs(age-median(age)))               # mittlere absolute Abweichung vom Median
[1] 4

> mad(age)                                 # Median der abs. Abweichungen vom Median
[1] 6.6717
```

Schiefe und Wölbung (Kurtosis) als höhere zentrale Momente empirischer Verteilungen lassen sich mit den Funktionen `skewness()` und `kurtosis()` aus dem `e1071` Paket ermitteln.

2.7.7 Kovarianz und Korrelation

Mit der Funktion `cov(x=⟨Vektor1⟩, y=⟨Vektor2⟩)` wird die korrigierte Kovarianz $\frac{1}{n-1} \cdot \sum_i ((x_i - M_x) \cdot (y_i - M_y))$ zweier Variablen X und Y derselben Länge n berechnet. Die unkorrigierte Kovarianz muss nach der bereits für die Varianz genannten Umrechnungsformel ermittelt werden.[25]

```
> x <- c(17, 30, 30, 25, 23, 21)           # Daten Variable 1
> y <- c(1, 12, 8, 10, 5, 3)               # Daten Variable 2
> cov(x, y)                                # korrigierte Kovarianz
[1] 19.2

> N    <- length(x)                        # Anzahl Objekte
> Mx   <- mean(x)                          # Mittelwert Var 1
> My   <- mean(y)                          # Mittelwert Var 2
> sum((x-Mx) * (y-My)) / (N-1)             # korrigierte Kovarianz manuell
[1] 19.2
```

[24] Die Funktion multipliziert den Median der absoluten Abweichungen automatisch mit dem Korrekturfaktor 1.4826, der über das Argument `constant` auf einen anderen Wert gesetzt werden kann. Der Faktor ist so gewählt, dass der Kennwert bei normalverteilten Variablen asymptotisch mit der Streuung übereinstimmt.

[25] Als Alternative ließe sich die `cov.wt()` Funktion verwenden (vgl. Abschn. 2.8.10).

```
> ((N-1) / N) * cov(x, y)                # unkorrig. Kovarianz aus korrigierter
[1] 16

> sum((x-Mx) * (y-My)) / N               # unkorrigierte Kovarianz manuell
[1] 16
```

Neben der voreingestellten Berechnungsmethode für die Kovarianz nach Pearson kann auch die Rang-Kovarianz nach Spearman oder Kendall berechnet werden (vgl. Abschn. 8.3.1).

Analog zur Kovarianz kann mit `cor(x=⟨Vektor1⟩, y=⟨Vektor2⟩)` die herkömmliche Produkt-Moment-Korrelation $r_{XY} = \frac{\text{Kov}_{XY}}{s_X \cdot s_Y}$ oder die Rangkorrelation berechnet werden.[26] Für die Korrelation gibt es keinen Unterschied beim Gebrauch von korrigierten und unkorrigierten Streuungen, sodass sich nur ein Kennwert ergibt.

```
> cor(x, y)                              # Korrelation
[1] 0.8854667

> cov(x, y) / (sd(x) * sd(y))            # manuelle Berechnung
[1] 0.8854667
```

Für die Berechnung der Partialkorrelation zweier Variablen X und Y ohne eine dritte Variable Z kann die Formel

$$r_{(XY).Z} = (r_{XY} - (r_{XZ} \cdot r_{YZ}))/\sqrt{(1 - r_{XZ}^2) \cdot (1 - r_{YZ}^2)}$$

umgesetzt werden, da die Basisinstallation von R hierfür keine eigene Funktion bereitstellt.[27] Für eine alternative Berechnungsmethode, die sich die Eigenschaft der Partialkorrelation als Korrelation der Residuen der Regressionen von X auf Z und Y auf Z zunutze macht, vgl. Abschn. 6.7.

```
> NN <- 100
> zz <- runif(NN)
> xx <- zz + rnorm(NN, 0, 0.5)
> yy <- zz + rnorm(NN, 0, 0.5)
> (cor(xx, yy) - (cor(xx, zz)*cor(yy, zz))) /
+      sqrt((1-cor(xx, zz)^2) * (1-cor(yy, zz)^2))
[1] 0.0753442
```

Die Semipartialkorrelation einer Variable Y mit einer Variable X ohne eine dritte Variable Z unterscheidet sich von der Partialkorrelation dadurch, dass nur von X der i. S. der linearen Regression durch Z aufklärbare Varianzanteil auspartialisiert wird, nicht aber von Y. Die Semipartialkorrelation berechnet sich durch $r_{(X.Z)Y} =$

[26] Das Paket `polycor` (Fox, 2010) beinhaltet Funktionen für die polychorische und polyseriale Korrelation zur Schätzung der latenten Korrelation von künstlich in Kategorien eingeteilten Variablen, die eigentlich stetig sind. Für die multiple Korrelation i. S. der Wurzel aus dem Determinationskoeffizienten R^2 in der multiplen linearen Regression vgl. Abschn. 6.2.2. Die kanonische Korrelation zweier Gruppen von Variablen, die an denselben Beobachtungsobjekten erhoben wurden, ermittelt `cancor()`.

[27] Wohl aber das Paket `ggm` mit der `pcor()` Funktion (Marchetti & Drton, 2010).

$(r_{XY} - (r_{XZ} \cdot r_{YZ}))/\sqrt{1 - r_{XZ}^2}$, oder als Korrelation von Y mit den Residuen der Regression von X auf Z (vgl. Abschn. 6.7).

```
> (cor(xx, yy) - (cor(xx, zz) * cor(yy, zz))) / sqrt(1-cor(xx, zz)^2)
[1] 0.06275869
```

2.7.8 Kennwerte getrennt nach Gruppen berechnen

Oft sind die Werte einer in verschiedenen Bedingungen erhobenen Variable in einem Vektor `x` gespeichert, wobei sich die zu jedem Wert gehörende Beobachtungsbedingung aus einem Faktor oder der Kombination mehrerer Faktoren ergibt. Jeder Faktor besitzt dabei dieselbe Länge wie `x` und codiert die Zugehörigkeit der Beobachtungen in `x` zu den Stufen einer Gruppierungsvariable. Dabei müssen nicht für jede Bedingung auch gleich viele Beobachtungen vorliegen.

Sollen Kennwerte von `x` jeweils getrennt für jede Bedingung bzw. Kombination von Bedingungen berechnet werden, können die Funktionen `ave()` und `tapply()` herangezogen werden.

```
> ave(x=⟨Vektor⟩, ⟨Faktor1⟩, ⟨Faktor2⟩, ...,          FUN=⟨Funktion⟩)
> tapply(X=⟨Vektor⟩, INDEX=⟨Liste mit Faktoren⟩,     FUN=⟨Funktion⟩, ...)
```

Als Argumente werden neben dem zuerst zu nennenden Datenvektor die Faktoren übergeben. Bei `ave()` geschieht dies einfach in Form mehrerer durch Komma getrennter Gruppierungsfaktoren. Bei `tapply()` müssen die Faktoren in einer Liste zusammengefasst werden, deren Komponenten die einzelnen Faktoren sind (vgl. Abschn. 3.1). Mit dem Argument `FUN` wird schließlich die pro Gruppe auf die Daten anzuwendende Funktion angegeben. Der Argumentname `FUN` ist bei `ave()` immer zu nennen, andernfalls wäre nicht ersichtlich, dass kein weiterer Faktor gemeint ist. In der Voreinstellung von `ave()` wird die `mean()` Funktion angewendet.

In der Ausgabe von `ave()` wird jeder Einzelwert von `x` durch den für die entsprechende Gruppe berechneten Kennwert ersetzt, was etwa in der Berechnung von Quadratsummen linearer Modelle Anwendung finden kann (vgl. Abschn. 7.3.6.1).

Im Beispiel sei ein IQ-Test mit Personen durchgeführt worden, die aus einer Treatment- (`T`), Wartelisten- (`WL`) oder Kontrollgruppe (`CG`) stammen. Weiterhin sei das Geschlecht als Faktor berücksichtigt worden.

```
> Nj       <- 2                       # Zellbesetzung
> P        <- 2                       # Anzahl Stufen Geschlecht
> Q        <- 3                       # Anzahl Stufen Treatment
> sex      <- factor(rep(c("f", "m"), times=Q*Nj))
> group    <- factor(rep(c("T", "WL", "CG"), each=P*Nj))
> table(sex, group)
      group
sex  CG  T  WL
  f   2  2   2
  m   2  2   2
```

```
> IQ <- round(rnorm(Nj*P*Q, mean=100, sd=15))
> ave(IQ, sex, FUN=mean)                    # Mittelwerte nach Geschlecht
[1] 96.83333 100.33333 96.83333 100.33333 96.83333 100.33333 96.83333
[8] 100.33333 96.83333 100.33333 96.83333 100.33333
```

Die Ausgabe von `tapply()` dient der Übersicht über die gruppenweise berechneten Kennwerte, wobei das Ergebnis ein Objekt der Klasse `array` ist (vgl. Abschn. 2.9). Bei einem einzelnen Gruppierungsfaktor ist dies einem benannten Vektor ähnlich und bei zweien einer zweidimensionalen Kreuztabelle (vgl. Abschn. 2.10).

```
> (tapRes <- tapply(IQ, group, FUN=mean))          # Mittelwert pro Gruppe
   CG       T       WL
94.25   99.75   101.75

# Mittelwert pro Bedingungskombination
> tapply(IQ, list(sex, group), FUN=mean)
      CG       T      WL
f   98.0    91.0   101.5
m   90.5   108.5   102.0
```

Wenn das Ergebnis ein eindimensionales array ist, lässt sich auch `tapply()` dazu verwenden, um jeden Wert durch einen für seine Gruppe berechneten Kennwert zu ersetzen: Die Elemente der Ausgabe tragen dann als Namen die zugehörigen Gruppenbezeichnungen und lassen sich über diese Namen indizieren. Die als Indizes verwendbaren Gruppenbezeichnungen finden sich im Faktor, wobei jeder Index entsprechend der zugehörigen Gruppengröße mehrfach auftaucht.

```
> tapRes[group]
    T       T       T       T       WL       WL       WL       WL
99.75   99.75   99.75   99.75   101.75   101.75   101.75   101.75

   CG      CG      CG      CG
94.25   94.25   94.25   94.25
```

Da für `FUN` beliebige Funktionen an `tapply()` übergeben werden können, muss man sich nicht darauf beschränken, für Gruppen getrennt einzelne Kennwerte zu berechnen. Genauso sind Funktionen zugelassen, die pro Gruppe mehr als einen einzelnen Wert ausgeben, wobei das Ergebnis von `tapply()` dann eine Liste ist (vgl. Abschn. 3.1): Jede Komponente der Liste beinhaltet die Ausgabe von `FUN` für eine Gruppe.

So ist es etwa auch möglich, sich die Werte jeder Gruppe selbst ausgeben zu lassen, indem man sich den Umstand zunutze macht, dass auch der `[⟨Index⟩]` Operator als Funktion `"["` geschrieben werden kann (vgl. Abschn. 1.2.5, Fußnote 13 sowie Abschn. 3.3.5 für die `split()` Funktion).

```
> tapply(IQ, sex, "[")                  # IQ-Werte pro Geschlecht
$f
[1] 100 82 88 115 86 110

$m
[1] 120 97 97 107 80 101
```

```
> split(IQ, sex)                    # Kontrolle ...
> IQ[sex == "f"]                    # Kontrolle ...
> IQ[sex == "m"]                    # Kontrolle ...
```

2.7.9 Funktionen auf geordnete Paare von Werten anwenden

Eine Verallgemeinerung der Anwendung einer Funktion auf jeden Wert eines Vektors stellt die Anwendung einer Funktion auf alle geordneten Paare aus den Werten zweier Vektoren dar.

```
> outer(X=⟨Vektor1⟩, Y=⟨Vektor2⟩, FUN="*", ...)
```

Für die Argumente `X` und `Y` ist dabei jeweils ein Vektor einzutragen, unter `FUN` eine Funktion, die zwei Argumente in Form von Vektoren verarbeiten kann.[28] Da Operatoren nur Funktionen mit besonderer Schreibweise sind, können sie hier ebenfalls eingesetzt werden, müssen dabei aber in Anführungszeichen stehen (vgl. Abschn. 1.2.5, Fußnote 13). Voreinstellung ist die Multiplikation, für diesen Fall existiert auch die Kurzform in Operatorschreibweise `X %o% Y`. Sollen an `FUN` weitere Argumente übergeben werden, kann dies anstelle der `...` geschehen, wobei mehrere Argumente durch Komma zu trennen sind. Die Ausgabe erfolgt in Form einer Matrix (vgl. Abschn. 2.8).

Als Beispiel sollen alle Produkte der Zahlen 1–5 als Teil des kleinen 1×1 ausgegeben werden.

```
> outer(1:5, 1:5, "*")
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

2.8 Matrizen

Wenn für jedes Beobachtungsobjekt Daten von mehreren Variablen vorliegen, könnten die Werte jeder Variable in einem separaten Vektor gespeichert werden. Eine andere Möglichkeit zur gemeinsamen Repräsentation aller Variablen bieten Objekte der Klasse `matrix` als Spezialfall von arrays (vgl. Abschn. 2.9).[29]

---

[28] Vor dem Aufruf von `FUN` verlängert `outer() X` und `Y` so, dass beide die Länge `length(X)*`↘ →`length(Y)` besitzen und sich aus der Kombination der Elemente mit gleichem Index alle geordneten Paare ergeben.

[29] Eine Matrix ist in R zunächst nur eine rechteckige Anordnung von Werten und nicht mit dem gleichnamigen mathematischen Konzept zu verwechseln. Wie `attributes(⟨Matrix⟩)` zeigt,

```
> matrix(data=⟨Vektor⟩, nrow=⟨Anzahl⟩, ncol=⟨Anzahl⟩, byrow=FALSE)
```

Unter `data` ist der Vektor einzutragen, der alle Werte der zu bildenden Matrix enthält. Mit `nrow` wird die Anzahl der Zeilen dieser Matrix festgelegt, mit `ncol` die der Spalten. Die Länge des Vektors muss gleich dem Produkt von `nrow` und `ncol` sein, das gleich der Zahl der Zellen ist. Mit dem auf `FALSE` voreingestellten Argument `byrow` wird die Art des Einlesens der Daten aus dem Vektor in die Matrix bestimmt – es werden zunächst die Spalten nacheinander gefüllt. Mit `byrow=TRUE` werden die Werte über die Zeilen eingelesen.

```
> age <- c(17, 30, 30, 25, 23, 21)
> matrix(age, nrow=3, ncol=2, byrow=FALSE)
     [,1]  [,2]
[1,]   17    25
[2,]   30    23
[3,]   30    21

> (ageMat <- matrix(age, nrow=2, ncol=3, byrow=TRUE))
     [,1]  [,2]  [,3]
[1,]   17    30    30
[2,]   25    23    21
```

2.8.1 Datentypen in Matrizen

Wie Vektoren können Matrizen verschiedene Datentypen besitzen, etwa `numeric`, wenn sie Zahlen beinhalten, oder `character` im Fall von Zeichenketten. Jede einzelne Matrix kann dabei aber ebenso wie ein Vektor nur einen einzigen Datentyp haben, alle Matrixelemente müssen also vom selben Datentyp sein. Fügt man einer numerischen Matrix eine Zeichenkette als Element hinzu, so werden die numerischen Matrixelemente automatisch in Zeichenketten umgewandelt,[30] was an den hinzugekommenen Anführungszeichen zu erkennen ist. Auf die ehemals numerischen Werte können dann keine Rechenoperationen mehr angewendet werden. Dieser Umstand macht Matrizen letztlich weniger geeignet für empirische Datensätze, für die stattdessen Objekte der Klasse `data.frame` bevorzugt werden sollten (vgl. Abschn. 3.2).[31]

---

sind Matrizen intern lediglich Vektoren mit einem Attribut (vgl. Abschn. 1.3), das Auskunft über die Dimensionierung der Matrix, also die Anzahl ihrer Zeilen und Spalten liefert. Eine Matrix kann deshalb maximal so viele Werte speichern, wie ein Vektor Elemente besitzen kann (vgl. Abschn. 2.1.1, Fußnote 1). Für Rechenoperationen mit Matrizen im Kontext der linearen Algebra vgl. Abschn. 10.1.

[30] Allgemein gesprochen werden alle Elemente in den umfassendsten Datentyp umgewandelt, der notwendig ist, um alle Werte ohne Informationsverlust zu speichern (vgl. Abschn. 1.3.5).

[31] Da Matrizen numerisch effizienter als Objekte der Klasse `data.frame` verarbeitet werden können, sind sie dagegen bei der Analyse sehr großer Datenmengen vorzuziehen.

2.8.2 Dimensionierung, Zeilen und Spalten

Die Dimensionierung einer Matrix (die Anzahl ihrer Zeilen und Spalten) liefert die Funktion `dim(⟨Matrix⟩)`, die auch auf arrays (vgl. Abschn. 2.9) oder Datensätze (vgl. Abschn. 3.2) anwendbar ist. Sie gibt einen Vektor aus, der die Anzahl der Zeilen und Spalten in dieser Reihenfolge als Elemente besitzt. Über die Funktionen `nrow(⟨Matrix⟩)` und `ncol(⟨Matrix⟩)` kann die Anzahl der Zeilen bzw. Spalten auch einzeln ausgegeben werden.

```
> age       <- c(17, 30, 30, 25, 23, 21)
> ageMat    <- matrix(age, nrow=2, ncol=3, byrow=FALSE)
> dim(ageMat)                          # Dimensionierung
[1] 2 3

> nrow(ageMat)                         # Anzahl der Zeilen
[1] 2

> ncol(ageMat)                         # Anzahl der Spalten
[1] 3

> prod(dim(ageMat))                    # Anzahl der Elemente
[1] 6
```

Eine Matrix wird mit der `t(⟨Matrix⟩)` Funktion transponiert, wodurch ihre Zeilen zu den Spalten der Transponierten und entsprechend ihre Spalten zu Zeilen der Transponierten werden.

```
> t(ageMat)
     [,1]  [,2]
[1,]   17    30
[2,]   30    25
[3,]   23    21
```

Wird ein Vektor über die Konvertierungsfunktion `as.matrix(⟨Vektor⟩)` in eine Matrix umgewandelt, entsteht als Ergebnis eine Matrix mit einer Spalte und so vielen Zeilen, wie der Vektor Elemente enthält.

```
> as.matrix(1:4)
      [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
```

Um eine Matrix in einen Vektor umzuwandeln, sollte entweder die Konvertierungsfunktion `as.vector(⟨Matrix⟩)` oder einfach `c(⟨Matrix⟩)` verwendet werden.[32] Die Anordnung der Elemente entspricht dabei dem Aneinanderhängen der Spalten der Matrix.

```
> c(ageMat)
[1] 17 30 30 25 23 21
```

[32] `c()` entfernt die Attribute der übergebenen Argumente bis auf ihre Elementnamen. Matrizen verlieren damit ihre Dimensionierung `dim` und ihre Klasse `matrix`.

Mitunter ist es nützlich, zu einer gegebenen Matrix zwei zugehörige Matrizen zu erstellen, in denen jedes ursprüngliche Element durch seinen Zeilen- bzw. Spaltenindex ersetzt wurde. Dieses Ziel lässt sich mit den Funktionen `row(⟨Matrix⟩)` und `col(⟨Matrix⟩)` erreichen.

```
> P          <- 2                        # Anzahl der Zeilen
> Q          <- 3                        # Anzahl der Spalten
> (pqMat     <- matrix(1:(P*Q), nrow=P, ncol=Q))
     [,1]  [,2]  [,3]
[1,]    1     3     5
[2,]    2     4     6

> (rowMat    <- row(pqMat))
     [,1]  [,2]  [,3]
[1,]    1     1     1
[2,]    2     2     2

> (colMat    <- col(pqMat))
     [,1]  [,2]  [,3]
[1,]    1     2     3
[2,]    1     2     3
```

Die so gewonnenen Matrizen können etwa dazu verwendet werden, eine dreispaltige Matrix zu erstellen, die in einer Spalte alle Elemente der ursprünglichen Matrix besitzt und in den anderen beiden Spalten die zugehörigen Zeilen- und Spaltenindizes enthält (vgl. Abschn. 2.8.5).

```
> cbind(rowIdx=c(rowMat), colIdx=c(colMat), val=c(pqMat))
     rowIdx  colIdx  val
[1,]      1       1    1
[2,]      2       1    2
[3,]      1       2    3
[4,]      2       2    4
[5,]      1       3    5
[6,]      2       3    6
```

Eine andere Anwendungsmöglichkeit besteht darin, logische untere bzw. obere Dreiecksmatrizen zu erstellen, die auch von den Funktionen `lower.tri()` und `upper.tri()` erzeugt werden können.

```
> mat <- matrix(sample(1:10, 16, replace=TRUE), nrow=4, ncol=4)
> col(mat) >= row(mat)                    # obere Dreiecksmatrix
      [,1]   [,2]   [,3]  [,4]
[1,]  TRUE   TRUE   TRUE  TRUE
[2,] FALSE   TRUE   TRUE  TRUE
[3,] FALSE  FALSE   TRUE  TRUE
[4,] FALSE  FALSE  FALSE  TRUE
```

2.8.3 Elemente auswählen und verändern

In einer Matrix ist es ähnlich wie bei einem Vektor möglich, sich einzelne Elemente mit dem `[⟨Zeile⟩, ⟨Spalte⟩]` Operator anzeigen zu lassen. Der erste Index in der eckigen Klammer gibt dabei die Zeile des gewünschten Elements an, der zweite dessen Spalte.[33]

```
> ageMat[2, 2]
[1] 25
```

Analog zum Vorgehen bei Vektoren können auch bei Matrizen einzelne Elemente unter Angabe ihres Zeilen- und Spaltenindex durch die Zuweisung eines Wertes verändert werden.

```
> ageMat[2, 2] <- 24; ageMat[2, 2]
[1] 24
```

Man kann sich Zeilen oder Spalten auch vollständig ausgeben lassen. Dafür wird für die vollständig aufzulistende Dimension kein Index eingetragen, jedoch das Komma trotzdem gesetzt. Es können dabei auch beide Dimensionen weggelassen werden, was für die Ausgabe denselben Effekt wie das gänzliche Weglassen des `[⟨Index⟩]` Operators hat.

```
> ageMat[2, ]                          # Werte 2. Zeile
[1] 30 24 21

> ageMat[ , 1]                         # Werte 1. Spalte
[1] 17 30

> ageMat[ , ]                          # gesamte Matrix, äquivalent: ageMat[]
     [,1]  [,2]  [,3]
[1,]   17    30    23
[2,]   30    24    21
```

Bei der Ausgabe einer einzelnen Zeile oder Spalte wird diese automatisch in einen Vektor umgewandelt, verliert also eine Dimension. Möchte man dies – wie es häufig der Fall ist – verhindern, kann beim `[⟨Index⟩]` Operator als weiteres Argument `drop=FALSE` angegeben werden. Das Ergebnis ist dann eine Matrix mit nur einer Zeile oder Spalte.

```
> ageMat[ , 1, drop=FALSE]
     [,1]
[1,]   17
[2,]   30
```

Analog zum Vorgehen bei Vektoren können auch gleichzeitig mehrere Matrixelemente ausgewählt und verändert werden, indem man etwa eine Sequenz oder einen anderen Vektor als Indexvektor für eine Dimension festlegt.

[33] Für Hilfe zu diesem Thema vgl. `?Extract`.

```
> ageMat[ , 2:3]                          # 2. und 3. Spalte
     [,1]  [,2]
[1,]   30    23
[2,]   24    21

> ageMat[ , c(1, 3)]                      # 1. und 3. Spalte
     [,1]  [,2]
[1,]   17    23
[2,]   30    21

> ageMatNew       <- ageMat
> (replaceMat     <- matrix(c(11, 21, 12, 22), nrow=2, ncol=2))
     [,1]  [,2]
[1,]   11    12
[2,]   21    22

> ageMatNew[ , c(1, 3)] <- replaceMat         # ersetze Spalten 1 und 3
> ageMatNew
     [,1]  [,2]  [,3]
[1,]   11    30    12
[2,]   21    24    22
```

2.8.4 Weitere Wege, Elemente auszuwählen und zu verändern

Auf Matrixelemente kann auch zugegriffen werden, wenn nur ein einzelner Index genannt wird. Die Matrix wird dabei implizit in den Vektor der untereinander gehängten Spalten umgewandelt.

```
> idxVec <- c(1, 3, 4)
> ageMat[idxVec]
[1] 17 30 24
```

Weiter können Matrizen auch durch eine logische Matrix derselben Dimensionierung – etwa das Ergebnis eines logischen Vergleichs – indiziert werden, die für jedes Element bestimmt, ob es ausgegeben werden soll. Das Ergebnis ist ein Vektor der ausgewählten Elemente.

```
> (idxMatLog <- ageMat >= 25)
      [,1]   [,2]   [,3]
[1,]  FALSE   TRUE  FALSE
[2,]   TRUE  FALSE  FALSE

> ageMat[idxMatLog]
[1] 30 30
```

Schließlich ist es möglich, eine zweispaltige numerische Indexmatrix zu verwenden, wobei jede Zeile dieser Matrix ein Element der indizierten Matrix auswählt – der erste Eintrag einer Zeile gibt den Zeilenindex, der zweite den Spaltenindex des auszuwählenden Elements an. Eine solche Matrix entsteht etwa bei der Umwandlung einer logischen in eine numerische Indexmatrix mittels `which()`

(vgl. Abschn. 2.2.2), wenn das Argument `arr.ind=TRUE` gesetzt ist. Auch hier ist das Ergebnis ein Vektor der ausgewählten Elemente.

```
> (idxMatNum <- which(idxMatLog, arr.ind=TRUE))
     row  col
[1,]   2    1
[2,]   1    2

> ageMat[idxMatNum]
[1] 30 30
```

Die Funktion `arrayInd(⟨Indexvektor⟩, dim(⟨Matrix⟩))` konvertiert einen numerischen Indexvektor, der eine Matrix im obigen Sinn als Vektor indiziert, in eine zweispaltige numerische Indexmatrix.

```
> (idxMat <- arrayInd(idxVec, dim(ageMat)))
     [,1]  [,2]
[1,]    1     1
[2,]    1     2
[3,]    2     2
```

2.8.5 Matrizen verbinden

Eine Möglichkeit, Matrizen aus vorhandenen Daten zu erstellen, besteht mit den Funktionen `cbind(⟨Vektor1⟩, ⟨Vektor2⟩, ...)` und `rbind(⟨Vektor1⟩,↘ →⟨Vektor2⟩, ...)`, die Vektoren zu Matrizen zusammenfügen. Das `c` bei `cbind()` steht für columns (Spalten), das `r` entsprechend für rows (Zeilen). In diesem Sinn werden die Vektoren mit `cbind()` spaltenweise nebeneinander, und mit `rbind()` zeilenweise untereinander angeordnet.

```
> age       <- c(19, 19, 19, 31, 24)
> weight    <- c(95, 76, 76, 94, 76)
> height    <- c(197, 179, 186, 189, 173)
> rbind(age, weight, height)
       [,1]  [,2]  [,3]  [,4]  [,5]
age      19    19    31    19    24
weight   95    76    94    76    76
height  197   178   189   184   173

> (mat <- cbind(age, weight, height))
     age  weight  height
[1,]  19      95     197
[2,]  19      76     178
[3,]  31      94     189
[4,]  19      76     184
[5,]  24      76     173
```

2.8.6 Matrizen sortieren

Die Zeilen von Matrizen können mithilfe der `order()` Funktion entsprechend der Reihenfolge der Werte in einer ihrer Spalten sortiert werden. Die `sort()` Funktion ist hier nicht anwendbar, ihr Einsatz ist auf Vektoren beschränkt.

```
> order(⟨Vektor⟩, partial, decreasing=FALSE)
```

Für ⟨`Vektor`⟩ ist die Spalte einer Datenmatrix einzutragen, deren Werte in eine Reihenfolge gebracht werden sollen. Unter `decreasing` wird die Sortierreihenfolge eingestellt: Per Voreinstellung `FALSE` wird aufsteigend sortiert, auf `TRUE` gesetzt absteigend. Die Ausgabe ist ein Indexvektor, der die Zeilenindizes der zu ordnenden Matrix in der Reihenfolge der Werte des Sortierkriteriums enthält (vgl. Abschn. 2.5.1).[34]

```
> (rowOrder1 <- order(mat[ , "age"]))          # Kriterium: Alter
[1] 1 2 4 5 3
```

Soll die gesamte Matrix entsprechend der Reihenfolge dieser Variable angezeigt werden, ist der von `order()` ausgegebene Indexvektor zum Indizieren der Zeilen der Matrix zu benutzen. Dabei ist der Spaltenindex unter Beibehaltung des Kommas wegzulassen.

```
> mat[rowOrder1, ]
      age  weight  height
[1,]   19      95     197
[2,]   19      76     178
[3,]   19      76     184
[4,]   24      76     173
[5,]   31      94     189
```

Mit dem Argument `partial` kann noch eine weitere Matrixspalte eingetragen werden, die dann als sekundäres Sortierkriterium verwendet wird. So kann eine Matrix etwa zunächst hinsichtlich einer die Gruppenzugehörigkeit darstellenden Variable sortiert werden und dann innerhalb jeder Gruppe nach der Reihenfolge der Werte einer anderen Variable. Es können noch weitere Sortierkriterien durch Komma getrennt als Argumente vorhanden sein, es gibt also keine Beschränkung auf nur zwei solcher Kriterien.

```
# sortiere primär nach Alter und sekundär nach Gewicht
> rowOrder2 <- order(mat[ , "age"], partial=mat[ , "weight"])
> mat[rowOrder2, ]
      age  weight  height
[1,]   19      76     178
[2,]   19      76     184
[3,]   19      95     197
[4,]   24      76     173
[5,]   31      94     189
```

[34] Die Funktion sortiert *stabil*: Zeilen mit gleich großen Werten des Sortierkriteriums behalten ihre Reihenfolge relativ zueinander bei, werden also beim Sortiervorgang nicht zufällig vertauscht.

Das Argument `decreasing` legt global für alle Sortierkriterien fest, ob auf- oder absteigend sortiert wird. Soll die Sortierreihenfolge dagegen zwischen den Kriterien variieren, kann einzelnen Kriterien ein - vorangestellt werden, was als Umkehrung der mit `decreasing` eingestellten Reihenfolge zu verstehen ist.

```
# sortiere aufsteigend nach Gewicht und absteigend nach Größe
> rowOrder3 <- order(mat[ , "weight"], -mat[ , "height"])
> mat[rowOrder3, ]
      age  weight  height
[1,]   19      76     184
[2,]   19      76     178
[3,]   24      76     173
[4,]   31      94     189
[5,]   19      95     197
```

2.8.7 Randkennwerte berechnen

Die Summe aller Elemente einer Matrix lässt sich mit `sum(⟨Matrix⟩)`, die separat über jede Zeile oder jede Spalte gebildeten Summen durch die Funktionen `rowSums(⟨Matrix⟩)` bzw. `colSums(⟨Matrix⟩)` berechnen. Gleiches gilt für den Mittelwert aller Elemente, der mit `mean(⟨Matrix⟩)` ermittelt wird und die mit `rowMeans()` bzw. `colMeans()` separat über jede Zeile oder jede Spalte berechneten Mittelwerte.

```
> sum(mat)                          # Summe aller Elemente
[1] 1450

> rowSums(mat)                      # Summen jeder Zeile
[1] 311 273 314 279 273

> mean(mat)                         # Mittelwert aller Elemente
[1] 96.66667

> colMeans(mat)                     # Mittelwerte jeder Spalte
 age  weight  height
22.4    83.4   184.2
```

2.8.8 Beliebige Funktionen auf Matrizen anwenden

Wenn eine andere Funktion als die Summe oder der Mittelwert separat auf jeweils jede Zeile oder jede Spalte angewendet werden soll, ist dies mit der Funktion `apply()` zu erreichen.

```
> apply(X=⟨Matrix⟩, MARGIN=⟨Nummer⟩, FUN=⟨Funktion⟩, ...)
```

`X` erwartet die Matrix der zu verarbeitenden Daten. Unter `MARGIN` wird angegeben, ob die Funktion Kennwerte der Zeilen (`1`) oder Spalten (`2`) berechnet. Für

FUN ist die anzuwendende Funktion einzusetzen, die als Argument einen Vektor akzeptieren muss. Gibt sie mehr als einen Wert zurück, ist das Ergebnis eine Matrix mit den Rückgabewerten von FUN in den Spalten. Die drei Punkte ... stehen für optionale, ggf. durch Komma getrennte Argumente von FUN, die an diese Funktion weitergereicht werden.

```
> apply(mat, 2, sum)                    # Summen jeder Spalte
age  weight  height
112     417     921

> apply(mat, 1, max)                    # Maxima jeder Zeile
[1] 197 178 189 184 173

> apply(mat, 1, range)                  # Range jeder Zeile
      [,1]  [,2]  [,3]  [,4]  [,5]
[1,]    19    19    31    19    24
[2,]   197   178   189   184   173

> apply(mat, 2, mean, trim=0.1)         # gestutzte Mittelwerte jeder Spalte
 age  weight  height
22.4    83.4   184.2
```

Im letzten Beispiel wird das für ... eingesetzte Argument trim=0.1 an die mean() Funktion weitergereicht.

2.8.9 Matrix zeilen- oder spaltenweise mit Kennwerten verrechnen

Zeilen- und Spaltenkennwerte sind häufig Zwischenergebnisse, die für weitere Berechnungen mit einer Matrix nützlich sind. So ist es etwa zum spaltenweisen Zentrieren einer Matrix notwendig, von jedem Wert den zugehörigen Spaltenmittelwert abzuziehen. Anders gesagt soll die Matrix dergestalt mit einem Vektor verrechnet werden, dass auf jede Spalte dieselbe Operation (hier: Subtraktion), aber mit einem anderen Wert angewendet wird – nämlich mit dem Element des Vektors der Spaltenmittelwerte, das dieselbe Position im Vektor besitzt wie die Spalte in der Matrix. Die genannte Operation lässt sich mit der sweep() Funktion durchführen.

```
> sweep(x=⟨Matrix⟩, MARGIN=⟨Nummer⟩, STATS=⟨Kennwerte⟩, FUN=⟨Funktion⟩, ...)
```

Das Argument x erwartet die Matrix der zu verarbeitenden Daten. Unter MARGIN wird angegeben, ob die Funktion jeweils Zeilen (1) oder Spalten (2) mit den Kennwerten verrechnet. An STATS sind diese Kennwerte in Form eines Vektors mit so vielen Einträgen zu übergeben, wie x Zeilen (MARGIN=1) bzw. Spalten (MARGIN=2↘
→) besitzt. Für FUN ist die anzuwendende Funktion einzusetzen, Voreinstellung ist die Subtraktion "-". Die drei Punkte ... stehen für optionale, ggf. durch Komma getrennte Argumente von FUN, die an diese Funktion weitergereicht werden.

Im Beispiel sollen die Daten einer Matrix erst zeilenweise, dann spaltenweise zentriert werden.

```
> Mj <- rowMeans(mat)                  # Mittelwerte der Zeilen
> Mk <- colMeans(mat)                  # Mittelwerte der Spalten
> sweep(mat, 1, Mj, "-")               # zeilenweise zentrieren
            age       weight     height
[1,]   -84.66667    -8.666667   93.33333
[2,]   -72.00000   -15.000000   87.00000
[3,]   -73.66667   -10.666667   84.33333
[4,]   -74.00000   -17.000000   91.00000
[5,]   -67.00000   -15.000000   82.00000

> t(scale(t(mat), center=TRUE, scale=FALSE))       # Kontrolle mit scale() ...
> sweep(mat, 2, Mk, "-")                           # spaltenweise zentrieren
       age   weight   height
[1,]  -3.4     11.6     12.8
[2,]  -3.4     -7.4     -6.2
[3,]   8.6     10.6      4.8
[4,]  -3.4     -7.4     -0.2
[5,]   1.6     -7.4    -11.2

> scale(mat, center=TRUE, scale=FALSE)              # Kontrolle mit scale() ...
```

2.8.10 Kovarianz- und Korrelationsmatrizen

Zum Erstellen von Kovarianz- und Korrelationsmatrizen können die schon bekannten Funktionen `var(⟨Matrix⟩)`, `cov(⟨Matrix⟩)` und `cor(⟨Matrix⟩)` verwendet werden, wobei die Werte in Form einer Matrix mit Variablen in den Spalten übergeben werden müssen. `var()` und `cov()` liefern beide die Kovarianzmatrix, wobei sie wie bei Vektoren die korrigierten Kennwerte berechnen.

```
> cov(mat)                                        # Kovarianzmatrix
          age  weight  height
age     27.80   22.55     0.4
weight  22.55  102.80    82.4
height   0.40   82.40    87.7

> cor(mat)                                         # Korrelationsmatrix
                age     weight       height
age      1.000000000  0.4218204  0.008100984
weight   0.421820411  1.0000000  0.867822404
height   0.008100984  0.8678224  1.000000000
```

Die Funktion `cov2cor(⟨Kovarianzmatrix⟩)` wandelt eine Kovarianzmatrix in eine Korrelationsmatrix um. Weiterhin existiert mit `cov.wt()` eine Funktion, die direkt die unkorrigierte Kovarianzmatrix ermitteln kann.

```
> cov.wt(x=⟨Matrix⟩, method=c("unbiased", "ML"))
```

Unter `x` ist die Matrix einzutragen, von deren Spalten paarweise die Kovarianz bestimmt werden soll. Um diese Funktion auf einen einzelnen Vektor anwenden zu können, muss dieser zunächst mit `as.matrix(⟨Vektor⟩)` in eine einspaltige

Matrix konvertiert werden. Mit `method` lässt sich wählen, ob die korrigierten oder unkorrigierten Varianzen und Kovarianzen ausgerechnet werden – Voreinstellung ist `"unbiased"` für die korrigierten Kennwerte. Sind die unkorrigierten Kennwerte gewünscht, ist `method="ML"` zu setzen, da sie bei normalverteilten Variablen die Maximum-Likelihood-Schätzung der theoretischen Parameter auf Basis einer Stichprobe darstellen.

Das Ergebnis der Funktion ist eine Liste (vgl. Abschn. 3.1), die die Kovarianzmatrix als Komponente `cov`, die Mittelwerte als Komponente `center` und die Anzahl der eingegangenen Fälle als Komponente `n.obs` (number of observations) besitzt.

```
> cov.wt(mat, method="ML")
$cov
           age  weight  height
age      22.24   18.04    0.32
weight   18.04   82.24   65.92
height    0.32   65.92   70.16

$center
 age  weight  height
22.4    83.4   184.2

$n.obs
[1] 5
```

Mit `diag(`⟨`Matrix`⟩`)` lassen sich aus einer Kovarianzmatrix die in der Diagonale stehenden Varianzen extrahieren.

```
> diag(cov(mat))
 age  weight  height
27.8   102.8    87.7
```

Um gleichzeitig die Kovarianz oder Korrelation einer durch ⟨`Vektor`⟩ gegebenen Variable mit mehreren anderen, spaltenweise zu einer Matrix zusammengefassten Variablen zu berechnen, dient der Aufruf `cov(`⟨`Matrix`⟩`, `⟨`Vektor`⟩`)` bzw. `cor(`⟨`Matrix`⟩`, `⟨`Vektor`⟩`)`. Das Ergebnis ist eine Matrix mit so vielen Zeilen, wie das erste Argument Spalten besitzt.

```
> vec <- rnorm(nrow(mat))
> cor(mat, vec)
              [,1]
age      -0.1843847
weight   -0.6645798
height   -0.6503452
```

2.9 Arrays

Das Konzept der Speicherung von Daten in eindimensionalen Vektoren und zweidimensionalen Matrizen lässt sich mit der Klasse `array` auf höhere Dimensionen

verallgemeinern. In diesem Sinne sind Vektoren und Matrizen ein- bzw. zweidimensionale Spezialfälle von arrays, weshalb sich arrays in allen wesentlichen Funktionen auch wie Matrizen verhalten. So müssen die in einem array gespeicherten Werte alle denselben Datentyp aufweisen, wie es auch bei Vektoren und Matrizen der Fall ist (vgl. Abschn. 2.8.1).[35]

```
> array(data=⟨Vektor⟩, dim=length(data), dimnames=NULL)
```

Für `data` ist ein Datenvektor mit den Werten anzugeben, die das array speichern soll. Mit `dim` wird die Dimensionierung festgelegt, also die Anzahl der Dimensionen und der Werte pro Dimension. Dies geschieht mithilfe eines Vektors, der pro Dimension ein Element beinhaltet, das die Anzahl der zugehörigen Werte spezifiziert. Das Argument `dim=c(2, 3, 4)` würde etwa festlegen, dass das array zwei Zeilen, drei Spalten und vier Schichten umfassen soll. Ein dreidimensionales array lässt sich nämlich als Quader vorstellen, der aus mehreren zweidimensionalen Matrizen besteht, die in Schichten hintereinander gereiht sind. Das Argument `dimnames` dient dazu, die Dimensionen und die einzelnen Stufen in jeder Dimension gleichzeitig mit Namen zu versehen. Dies geschieht unter Verwendung einer Liste (vgl. Abschn. 3.1), die für jede Dimension eine Komponente in Form eines Vektors mit den gewünschten Bezeichnungen der einzelnen Stufen besitzt. Die Namen der Dimensionen selbst können über die Benennung der Komponenten der Liste festgelegt werden.

Als Beispiel soll die Kontingenztafel dreier kategorialer Variablen dienen: Geschlecht mit zwei und zwei weitere Variablen mit drei bzw. zwei Stufen. Ein dreidimensionales array wird durch separate zweidimensionale Matrizen für jede Stufe der dritten Dimension ausgegeben.

```
> (myArr1 <- array(1:12, c(2, 3, 2), dimnames=list(Zeile=c("f", "m"),
+                  Spalte=c("CG", "WL", "T"), Schicht=c("high", "low"))))
, , Schicht = high
       Spalte
Zeile  CG  WL  T
    f   1   3  5
    m   2   4  6

, , Schicht = low
       Spalte
Zeile  CG  WL   T
    f   7   9  11
    m   8  10  12
```

Das array wird durch die mit dem Vektor `1:12` bereitgestellten Daten in Reihenfolge der Dimensionen aufgefüllt: zunächst alle Zeilen der ersten Spalte der ersten Schicht, dann in diesem Muster alle Spalten der ersten Schicht und zuletzt in diesem Muster alle Schichten. Auf arrays lassen sich mit `apply()` wie bei Matrizen beliebige Funktionen in Richtung der einzelnen Dimensionen anwenden.

[35] So, wie sich Matrizen mit `cbind()` und `rbind()` aus Vektoren zusammenstellen lassen, ermöglicht die Funktion `abind()` aus dem gleichnamigen Paket (Plate & Heiberger, 2011) das Verbinden von Matrizen zu einem array.

Arrays lassen sich analog zu Matrizen mit dem `[⟨Index⟩]` Operator indizieren, wobei die Indizes für die zusätzlichen Dimensionen durch Komma getrennt hinzugefügt werden.

```
> myArr1[1, 3, 2]                # Element in 1. Zeile, 3. Spalte, 2. Schicht
[1] 11

> myArr2 <- myArr1*2
> myArr2[ , , "high"]            # zeige nur 1. Schicht
     Spalte
Zeile  CG  WL   T
    f   2   6  10
    m   4   8  12
```

Ähnlich wie sich bei Matrizen durch Transponieren mit `t()` Zeilen und Spalten vertauschen lassen, können mit `aperm()` auch bei arrays Dimensionen ausgetauscht werden.

```
> aperm(a=⟨array⟩, perm=⟨Vektor⟩)
```

Unter `a` ist das zu transformierende n-dimensionale array anzugeben. `perm` legt in Form eines Vektors mit den Elementen 1 bis n fest, welche Dimensionen vertauscht werden sollen. Im Fall benannter Dimensionen kann `perm` alternativ auch deren Namen enthalten. Die Position eines Elements von `perm` bezieht sich auf die alte Dimension, das Element selbst bestimmt, zu welcher neuen Dimension die alte gemacht wird. Sollen in einem dreidimensionalen array die Schichten zu Zeilen (und umgekehrt) werden, wäre `perm=c(3, 2, 1)` zu setzen. Das Vertauschen von Zeilen und Spalten wäre mit `perm=c(2, 1, 3)` zu erreichen.

2.10 Häufigkeitsauszählungen

Bei der Analyse kategorialer Variablen besteht ein typischer Auswertungsschritt darin, die Auftretenshäufigkeiten der Kategorien auszuzählen und relative sowie bedingte relative Häufigkeiten zu berechnen. Wird nur eine Variable betrachtet, ergeben sich einfache Häufigkeitstabellen, bei mehreren Variablen mehrdimensionale Kontingenztafeln der gemeinsamen Häufigkeiten.

2.10.1 Einfache Tabellen absoluter und relativer Häufigkeiten

Eine Tabelle der absoluten Häufigkeiten von Variablenwerten erstellt `table(⟨`↘
→`Faktor⟩)` und erwartet dafür als Argument ein eindimensionales Objekt, das sich als Faktor interpretieren lässt, z. B. einfach einen Vektor. Das Ergebnis ist eine Über-

sicht über die Auftretenshäufigkeit jeder vorkommenden Ausprägung, wobei fehlende Werte ignoriert werden.[36]

```
> (myLetters <- sample(LETTERS[1:5], 12, replace=TRUE))
[1] "C" "D" "A" "D" "E" "D" "C" "E" "E" "B" "E" "E"

> (tab <- table(myLetters))
myLetters
A  B  C  D  E
1  1  2  3  5
```

In der oberen Zeile der Ausgabe sind die Ausprägungen der Variable, in der unteren Zeile die jeweils zugehörigen Auftretenshäufigkeiten aufgeführt. Eindimensionale Häufigkeitstabellen verhalten sich wie Vektoren mit benannten Elementen, wobei die Benennungen den vorhandenen Ausprägungen der Variable entsprechen. Die Ausprägungen lassen sich mit dem Befehl `names(⟨Tabelle⟩)` separat ausgeben.

```
> names(tab)
[1] "A" "B" "C" "D" "E"

> tab["B"]
B
1
```

Relative Häufigkeiten ergeben sich durch Division der absoluten Häufigkeiten mit der Gesamtzahl der Beobachtungen, also mit `table(⟨Faktor⟩) / length(⟨Faktor⟩)`. Für diese Rechnung existiert auch die Funktion `prop.table(⟨Tabelle⟩)`, welche als Argument eine Tabelle der absoluten Häufigkeiten erwartet und die relativen Häufigkeiten ausgibt. Durch Anwendung von `cumsum()` auf das Ergebnis erhält man die kumulierten relativen Häufigkeiten (für eine andere Methode und die Berechnung von Prozenträngen vgl. Abschn. 2.10.5).

```
> (relFreq <- prop.table(table(myLetters)))
myLetters
         A          B          C          D          E
0.08333333 0.08333333 0.16666667 0.25000000 0.41666667

> cumsum(relFreq)                  # kumulierte relative Häufigkeiten
         A          B          C          D          E
0.08333333 0.16666667 0.33333333 0.58333333 1.00000000
```

Kommen mögliche Variablenwerte in einem Vektor nicht vor, so tauchen sie auch in einer mit `table()` erstellten Häufigkeitstabelle nicht als ausgezählte Kategorie auf. Um deutlich zu machen, dass Variablen außer den tatsächlich vorhandenen Ausprägungen potenziell auch weitere Werte annehmen, kann auf zweierlei Weise vorgegangen werden: So können die möglichen, tatsächlich aber nicht auftretenden Werte der Häufigkeitstabelle nachträglich mit der Häufigkeit 0 hinzugefügt werden.

[36] Ist das erste Argument von `table()` ein Vektor, können fehlende Werte über das Argument `exclude=NULL` mit in die Auszählung einbezogen werden. Damit in Faktoren vorkommende fehlende Werte unter einer eigenen Kategorie berücksichtigt werden, muss der Faktor `NA` als eigene Stufe enthalten und deshalb mit `factor(⟨Vektor⟩, exclude=NULL)` gebildet werden.

```
> tab["Q"] <- 0; tab
A  B  C  D  E  Q
1  1  2  3  5  0
```

Beim Hinzufügen von Werten zur Tabelle werden diese ans Ende angefügt. Geht dadurch die natürliche Reihenfolge der Ausprägungen verloren, kann die Tabelle z. B. mittels eines durch `order(names(⟨Tabelle⟩))` erstellten Vektors der geordneten Indizes sortiert werden.

```
> tabOrder <- order(names(tab))
> tab[tabOrder]                          # ...
```

Alternativ zur Veränderung der Tabelle selbst können die Daten vorab in einen Faktor umgewandelt werden. Dem Faktor lässt sich dann der nicht auftretende, aber prinzipiell mögliche Wert als weitere Stufe hinzufügen.

```
> letFac <- factor(myLetters, levels=c(LETTERS[1:5], "Q")); letFac
[1] C D A D E D C E E B E E
Levels: A B C D E Q

> table(letFac)
letFac
A  B  C  D  E  Q
1  1  2  3  5  0
```

2.10.2 Iterationen zählen

Unter einer *Iteration* innerhalb einer linearen Sequenz von Symbolen ist ein Abschnitt zu verstehen, der aus der ein- oder mehrfachen Wiederholung desselben Symbols besteht (engl. run). Iterationen werden durch Iterationen eines anderen Symbols begrenzt, oder besitzen kein vorangehendes bzw. auf sie folgendes Symbol. Die Iterationen eines Vektors zählt die `rle(⟨Vektor⟩)` Funktion, deren Ergebnis eine Liste mit zwei Komponenten ist: Die erste Komponente `lengths` ist ein Vektor, der die jeweilige Länge jeder Iteration als Elemente besitzt. Die zweite Komponente `values` ist ein Vektor mit den Symbolen, um die es sich bei den Iterationen handelt (vgl. Abschn. 3.1).

```
> (vec <- rep(rep(c("f", "m"), 3), c(1, 3, 2, 4, 1, 2)))
[1] "f" "m" "m" "m" "f" "f" "m" "m" "m" "m" "f" "m" "m"

> (res <- rle(vec))
Run Length Encoding
lengths: int [1:6] 1 3 2 4 1 2
values : chr [1:6] "f" "m" "f" "m" "f" "m"

> length(res$lengths)            # zähle Anzahl der Iterationen
[1] 6
```

Aus der jeweiligen Länge und dem wiederholten Symbol jeder Iteration lässt sich die ursprüngliche Sequenz eindeutig rekonstruieren. Dies kann durch die `inverse.rle(⟨rle-Ergebnis⟩)` Funktion geschehen, die eine Liste erwartet, wie sie `rle()` als Ergebnis besitzt.

```
> inverse.rle(res)
[1] "f" "m" "m" "m" "f" "f" "m" "m" "m" "m" "f" "m" "m"
```

2.10.3 *Absolute, relative und bedingte relative Häufigkeiten in Kreuztabellen*

Statt die Häufigkeiten der Werte nur einer einzelnen Variable zu ermitteln, können mit `table(⟨Faktor1⟩, ⟨Faktor2⟩, ...)` auch mehrdimensionale Kontingenztafeln erstellt werden. Die Elemente der Faktoren an gleicher Position werden als denselben Beobachtungsobjekten zugehörig interpretiert. Das erste Element von `⟨Faktor1⟩` bezieht sich also auf dieselbe Beobachtung wie das erste Element von `⟨Faktor2⟩`, usw. Das Ergebnis ist eine Kreuztabelle mit den gemeinsamen absoluten Häufigkeiten der Merkmale, wobei die Ausprägungen des ersten Faktors in den Zeilen stehen.

Als Beispiel sollen Personen betrachtet werden, die nach ihrem Geschlecht und dem Ort ihres Arbeitsplatzes unterschieden werden. An diesen Personen sei weiterhin eine Variable erhoben worden, die die absolute Häufigkeit eines bestimmten Ereignisses codiert.

```
> N        <- 10
> (work    <- factor(sample(c("home", "office"), N, replace=TRUE)))
[1] home office office home home office office office office office
Levels: home office

> (sex <- factor(sample(c("f", "m"), N, replace=TRUE)))
[1] f m f m m f m f m m
Levels: f m

> (counts <- sample(0:5, N, replace=TRUE))
[1] 0 3 3 1 1 1 3 5 2 4

# gemeinsame absolute Häufigkeiten von Geschlecht und Arbeitsplatz
> (absFreq <- table(sex, work))
       work
sex  home  office
  f     1       3
  m     2       4
```

Um relative Häufigkeiten auf Basis von Kreuztabellen absoluter Häufigkeiten zu ermitteln, eignet sich die `prop.table()` Funktion. Für bedingte relative Häufigkeiten besitzt sie ein zweites Argument `margin=⟨Nummer⟩`, an das etwa `1` zur Bestimmung der auf die Zeilen bezogenen bedingten relativen Häufigkeiten übergeben werden kann. Jede Zeile der Tabelle relativer Häufigkeiten wird dann durch

die zugehörige Zeilensumme dividiert, was sich manuell auch durch `sweep()` erreichen ließe. Mit `margin=2` erhält man analog die auf die Spalten bezogenen bedingten relativen Häufigkeiten.

```
> (relFreq <- prop.table(absFreq))                    # relative Häufigkeiten
       work
sex  home  office
  f   0.1     0.3
  m   0.2     0.4

> prop.table(absFreq, 1)           # auf Zeilen bedingte rel. Häufigkeiten
             work
sex       home     office
  f  0.2500000  0.7500000
  m  0.3333333  0.6666667

> prop.table(absFreq, 2)           # auf Spalten bedingte rel. Häufigkeiten
             work
sex       home     office
  f  0.3333333  0.4285714
  m  0.6666667  0.5714286

# manuelle Kontrolle
> rSums <- rowSums(relFreq)          # Zeilensummen
> cSums <- colSums(relFreq)          # Spaltensummen
> sweep(relFreq, 1, rSums, "/")      # auf Zeilen bedingte rel. Häufigkeiten ...
> sweep(relFreq, 2, cSums, "/")      # auf Spalten bedingte rel. Häufigkeiten ...
```

Um Häufigkeitsauszählungen für mehr als zwei Variablen zu berechnen, können beim Aufruf von `table()` einfach weitere Faktoren durch Komma getrennt hinzugefügt werden. Die Ausgabe verhält sich dann wie ein array. Hierbei werden etwa im Fall von drei Variablen so viele zweidimensionale Kreuztabellen ausgegeben, wie Stufen der dritten Variable vorhanden sind. Soll dagegen auch in diesem Fall eine einzelne Kreuztabelle mit verschachteltem Aufbau erzeugt werden, ist die `ftable()` Funktion (flat table) zu nutzen.

```
> ftable(x, row.vars=NULL, col.vars=NULL)
```

Unter `x` kann entweder eine bereits mit `table()` erzeugte Kreuztabelle eingetragen werden, oder aber eine durch Komma getrennte Reihe von Faktoren bzw. von Objekten, die sich als Faktor interpretieren lassen. Die Argumente `row.vars` und `col.vars` kontrollieren, welche Variablen in den Zeilen und welche in den Spalten angeordnet werden. Beide Argumente akzeptieren numerische Vektoren mit den Nummern der entsprechenden Variablen, oder aber Vektoren aus Zeichenketten, die den Namen der Faktoren entsprechen.

```
> (group <- factor(sample(c("A", "B"), 10, replace=TRUE)))
[1] B B A B A B A B A A
Levels: A B

> ftable(work, sex, group, row.vars="work", col.vars=c("sex", "group"))
         sex    f      m
        group  A  B   A  B
```

```
work
home          0  1   1  1
office        1  2   3  1
```

Beim Erstellen von Kreuztabellen kann auch auf die Funktion `xtabs()` zurückgegriffen werden, insbesondere wenn sich die Variablen in Datensätzen befinden (vgl. Abschn. 3.2).

```
> xtabs(formula=~ ., data=⟨Datensatz⟩)
```

Im ersten Argument `formula` erwartet `xtabs()` eine sog. *Modellformel* (vgl. Abschn. 5.2). Hier ist dies eine besondere Art der Aufzählung der in der Kontingenztafel zu berücksichtigenden Faktoren, die durch ein + verknüpft rechts von der Tilde ~ aufgeführt werden. In der Voreinstellung `~ .` werden alle Variablen des unter `data` angegebenen Datensatzes einbezogen, d. h. alle möglichen Kombinationen von Faktorstufen gebildet. Eine links der ~ genannte Variable wird als Vektor von Häufigkeiten interpretiert, die pro Stufenkombination der rechts von der ~ genannten Faktoren zu addieren sind. Stammen die in der Modellformel genannten Variablen aus einem Datensatz, muss dieser unter `data` aufgeführt werden. Die Ausprägungen des rechts der ~ zuerst genannten Faktors bilden die Zeilen der ausgegebenen Kontingenztafel.

```
> (persons <- data.frame(sex, work, counts))        # Variablen als Datensatz
    sex    work  counts
1     f    home       0
2     m  office       3
3     f  office       3
4     m    home       1
5     m    home       1
6     f  office       1
7     m  office       3
8     f  office       5
9     m  office       2
10    m  office       4

# gemeinsame Häufigkeiten der Stufen von sex und work
> xtabs(~ sex + work, data=persons)
         work
sex  home  office
  f     1       3
  m     2       4

# Summe von counts pro Zelle
> xtabs(counts ~ sex + work, data=persons)
        work
sex  home  office
  f     0       9
  m     2      12
```

Einen Überblick über die Zahl der in einer Häufigkeitstabelle ausgewerteten Faktoren sowie die Anzahl der zugrunde liegenden Beobachtungsobjekte erhält man mit `summary(⟨Tabelle⟩)`. Im Fall von Kreuztabellen wird hierbei zusätzlich ein

χ^2-Test auf Unabhängigkeit bzw. auf Gleichheit von Verteilungen berechnet (vgl. Abschn. 8.2.1, 8.2.2).

```
> summary(table(sex, work))
Number of cases in table: 10
Number of factors: 2
Test for independence of all factors:
Chisq = 4.444, df = 1, p-value = 0.03501
Chi-squared approximation may be incorrect
```

2.10.4 Randkennwerte von Kreuztabellen

Um Randsummen, Randmittelwerte oder ähnliche Kennwerte für eine Kreuztabelle zu berechnen, können alle für Matrizen vorgestellten Funktionen verwendet werden, insbesondere `apply()`, aber etwa auch `rowSums()` und `colSums()` sowie `rowMeans()` und `colMeans()`. Hier nimmt die Tabelle die Rolle der Matrix ein.

```
# Zeilensummen
> apply(xtabs(~ sex + work, data=persons), MARGIN=1, FUN=sum)
f m
4 6

# Spaltenmittel
> colMeans(xtabs(~ sex + work, data=persons))
home office
 1.5    3.5
```

`addmargins()` berechnet beliebige Randkennwerte für eine Kreuztabelle `A` entsprechend der mit dem Argument `FUN` bezeichneten Funktion. Die Funktion operiert separat über jeder der mit dem Vektor `margin` bezeichneten Dimensionen. Die Ergebnisse der Anwendung von `FUN` werden `A` in der Ausgabe als weitere Zeile und Spalte hinzugefügt.

```
> addmargins(A=⟨Tabelle⟩, margin=⟨Vektor⟩, FUN=⟨Funktion⟩)

# Randmittel
> addmargins(xtabs(~ sex + work, data=persons), c(1, 2), mean)
          work
 sex  home  office  mean
   f   1.0     3.0   2.0
   m   2.0     4.0   3.0
mean   1.5     3.5   2.5
```

2.10.5 Kumulierte relative Häufigkeiten und Prozentrang

Die `ecdf(x=⟨Vektor⟩)` Funktion (empirical cumulative distribution function) ermittelt die kumulierten relativen Häufigkeiten kategorialer Daten. Diese geben für einen

Wert an, welcher Anteil der Daten nicht größer als dieser ist. Das Ergebnis ist analog zur Verteilungsfunktion quantitativer Zufallsvariablen.

Das Ergebnis von `ecdf()` ist eine Stufenfunktion mit so vielen Sprungstellen, wie es unterschiedliche Werte in `x` gibt, also `length(unique(x))`. Die Höhe jedes Sprungs entspricht der relativen Häufigkeit des Wertes an der Sprungstelle. Enthält `x` also keine mehrfach vorkommenden Werte, erzeugt `ecdf()` eine Stufenfunktion mit so vielen Sprungstellen, wie `x` Elemente besitzt. Dabei weist jeder Sprung dieselbe Höhe auf – die relative Häufigkeit `1/length(x)` jedes Elements von `x`. Tauchen in `x` Werte mehrfach auf, unterscheiden sich die Sprunghöhen dagegen entsprechend den relativen Häufigkeiten.

Die Ausgabe von `ecdf()` ist ihrerseits eine Funktion, die zunächst einem eigenen Objekt zugewiesen werden muss, ehe sie zur Ermittlung kumulierter relativer Häufigkeiten Verwendung finden kann. Ist `Fn()` diese Stufenfunktion, und möchte man die kumulierten relativen Häufigkeiten der in `x` gespeicherten Werte erhalten, ist `x` selbst als Argument für den Aufruf von `Fn()` einzusetzen. Andere Werte als Argument von `Fn()` sind aber ebenfalls möglich. Indem `Fn()` für beliebige Werte ausgewertet wird, lassen sich empirische und interpolierte Prozentränge ermitteln. Mit `ecdf()` erstellte Funktionen lassen sich über `plot()` direkt grafisch abbilden (Abb. 8.1, 11.23, vgl. Abschn. 11.6.6).

```
> (vec <- round(rnorm(10), 2))
[1] -1.57 2.21 -1.01 0.21 -0.29 -0.61 -0.17 1.90 0.17 0.55

# kumulierte relative Häufigkeiten der vorhandenen Werte
> Fn <- ecdf(vec)
> Fn(vec)
[1] 0.1 1.0 0.2 0.7 0.4 0.3 0.5 0.9 0.6 0.8

> 100 * Fn(0.1)                                    # Prozentrang von 0.1
[1] 50

> 100 * (sum(vec <= 0.1) / length(vec))            # Kontrolle
[1] 50
```

Soll die Ausgabe der kumulierten relativen Häufigkeiten in der richtigen Reihenfolge erfolgen, müssen die Werte mit `sort()` geordnet werden.

```
> Fn(sort(vec))
[1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Die Sprungstellen einer von `ecdf()` erstellten Funktion lassen sich mit der `knots()` Funktion extrahieren. Das Ergebnis sind gerade die in `x` enthaltenen unterschiedlichen sortierten Werte.

```
> knots(Fn)
[1] -1.57 -1.01 -0.61 -0.29 -0.17 0.17 0.21 0.55 1.90 2.21
```

2.10.6 Diversität kategorialer Daten

Der Diversitätsindex H ist ein Streuungsmaß für kategoriale Daten, der mit Werten im Intervall $[0, 1]$ angibt, wie sehr sich die Daten auf die Kategorien verteilen. Sind f_j die relativen Häufigkeiten der p Kategorien, ist $H = -\frac{1}{\ln p} \cdot \sum (f_j \cdot \ln f_j)$. Für $f_j = 0$ ist $f_j \cdot \ln f_j$ dabei als 0 definiert. H ist genau dann 0, wenn die Daten konstant sind, also nur eine von mehreren Kategorien auftritt. Für eine Gleichverteilung, in der alle Kategorien dieselbe Häufigkeit besitzen, die Daten also maximal verteilt sind, ist $H = 1$. R verfügt über keine spezialisierte Funktion für die Berechnung von H. Es lässt sich aber der Umstand nutzen, dass H bis auf den Faktor $\frac{1}{\ln p}$ mit dem Shannon-Index aus der Informationstheorie übereinstimmt, der sich mit der aus dem Paket `vegan` (Oksanen et al., 2011) stammenden Funktion `diversity()` berechnen lässt.[37] Die Funktion erwartet als Argument die absoluten oder relativen Häufigkeiten der Kategorien als Zeilenvektor.

```
> P    <- nlevels(letFac)                    # Anzahl Kategorien
> Fj   <- prop.table(table(letFac))          # relative Häufigkeiten
> library(vegan)                             # für diversity()
> shannonIdx <- diversity(t(Fj))             # Shannon Index
> (H <- (1/log(P)) * shannonIdx)             # Diversität
[1] 0.794822

> keep <- Fj > 0                             # Indizes der Häufigkeiten > 0
> -(1/log(P)) * sum(Fj[keep] * log(Fj[keep]))          # Kontrolle ...
```

2.11 Codierung, Identifikation und Behandlung fehlender Werte

Empirische Datensätze besitzen häufig zunächst keine zufriedenstellende Qualität, etwa durch Fehler in der Eingabe von Werten, Ausreißer (vgl. Abschn. 6.6.1, 11.6) oder durch unvollständige Daten – wenn also nicht für alle Beobachtungsobjekte Werte von allen erhobenen Variablen vorliegen. So können etwa VPn die Auskunft bzgl. bestimmter Fragen verweigern, Aufgaben übersehen oder in adaptiven Tests aufgrund von Verzweigungen nicht vorgelegt bekommen.

Fehlende Werte bergen aus versuchsplanerischer Perspektive die Gefahr, dass sie womöglich nicht zufällig, sondern systematisch entstanden sind und so zu verzerrten Ergebnissen führen. Aber auch bei der statistischen Auswertung werfen sie Fragen auf: Zum einen sind verschiedene Strategien des Umgangs mit ihnen denkbar, die nicht unbedingt zu gleichen Ergebnissen führen. Zum anderen können fehlende Werte bewirken, dass nicht wie beabsichtigt in allen Experimentalbedingungen dieselbe Anzahl von Beobachtungen vorliegt. Dies ist jedoch für die Anwendung und

[37] Zudem gilt folgende Beziehung zur diskreten Kullback-Leibler-Divergenz KL_{eq} der beobachteten Häufigkeiten zur Gleichverteilung: $H = -\frac{1}{\ln p} \cdot KL_{eq} + 1$.

Interpretation vieler üblicher Verfahren relevant, deren Einsatz durch das Vorliegen fehlender Werte problematisch werden könnte.[38]

Für die Behandlung fehlender Werte in statistischen Tests vgl. Abschn. 5.4. Um fehlende Werte in Indexvektoren zu vermeiden, wo sie meist zu nicht intendierten Konsequenzen führen, sollten logische Indexvektoren mit `which()` in numerische konvertiert werden (vgl. Abschn. 2.2.2).

2.11.1 Fehlende Werte codieren und ihr Vorhandensein prüfen

Wenn ein Datensatz eingegeben wird und fehlende Werte vorliegen, so dürfen diese nicht einfach weggelassen, sondern müssen mit der Konstante `NA` (not available) codiert werden – auch bei `character` Vektoren ist sie nicht in Anführungszeichen zu setzen.[39]

```
> (vec1 <- c(10, 20, NA, 40, 50, NA))
[1] 10 20 NA 40 50 NA

> length(vec1)
[1] 6
```

In manchen Situationen werden Zeichenketten prinzipiell ohne Anführungszeichen ausgegeben, etwa bei Faktoren oder Datensätzen (vgl. Abschn. 3.2). Zur leichteren Unterscheidung von gültigen Zeichenketten erscheinen fehlende Werte deshalb dann als `<NA>`.[40]

```
# Ausgabe von Zeichenketten mit Anführungszeichen -> nur NA
> LETTERS[c(1, NA, 3)]
[1] "A" NA "C"

# Ausgabe von Zeichenketten ohne Anführungszeichen -> <NA>
> factor(LETTERS[c(1, NA, 3)])
[1] A <NA> C
Levels: A C
```

---

[38] Neben den hier beschriebenen Methoden zur Behandlung fehlender Werte existieren auch andere Versuche, mit diesem Problem umzugehen. Das als *multiple Imputation* bezeichnete Verfahren ersetzt fehlende Werte durch solche Zahlen, die unter Berücksichtigung bestimmter Rahmenbedingungen generiert wurden und dabei Eigenschaften der tatsächlich vorhandenen Daten berücksichtigen sollen. Multiple Imputation wird in R u. a. durch die Pakete `Hmisc`, `Amelia II` (Honaker, King & Blackwell, 2011) und `mice` (van Buuren & Groothuis-Oudshoorn, 2011) unterstützt.

[39] Für jeden Datentyp existiert jeweils eine passende Konstante, all diese Konstanten tragen aber den Namen `NA`. Der mit `typeof(NA)` ausgegebene Basis-Datentyp ist `logical`. Damit ist die Ausgabe etwa von `c(1, 2, 3)[NA]` gleich `NA NA NA`, da logische Indexvektoren zyklisch verlängert werden (vgl. Abschn. 2.1.2, 2.2.2).

[40] In solchen Situationen ist also `NA` die Ausgabe eines gültigen Elements und von einem fehlenden Wert `<NA>` zu unterscheiden. So erzeugt `factor(c("A", "NA", "C"))[c(NA, 2, 3)]` die Ausgabe `<NA> NA C`. Einzig die gültige Zeichenkette `"<NA>"` lässt sich in der Ausgabe nicht von einem fehlenden Wert unterscheiden. In diesem Fall kann nur mithilfe der `is.na()` Funktion festgestellt werden, ob es sich tatsächlich um einen fehlenden Wert handelt oder nicht.

Ob in einem Vektor fehlende Werte vorhanden sind, wird mit der Funktion `is.na(⟨Vektor⟩)` ermittelt.[41] Sie gibt einen logischen Vektor aus, der für jede Position angibt, ob das Element ein fehlender Wert ist. Im Fall einer Datenmatrix liefert `is.na()` eine Matrix aus Wahrheitswerten, die für jedes Matrixelement angibt, ob es sich um einen fehlenden Wert handelt.

```
> is.na(vec1)
[1] FALSE FALSE TRUE FALSE FALSE TRUE

> vec2      <- c(NA, 7, 9, 10, 1, 8)
> (matNA    <- rbind(vec1, vec2))
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
vec1    10    20    NA    40    50    NA
vec2    NA     7     9    10     1     8

> is.na(matNA)
       [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
vec1  FALSE  FALSE   TRUE  FALSE  FALSE   TRUE
vec2   TRUE  FALSE  FALSE  FALSE  FALSE  FALSE
```

Bei einem großen Datensatz ist es mühselig, die Ausgabe von `is.na()` manuell nach `TRUE` Werten zu durchsuchen. Daher bietet sich `any()` an, um zu erfahren, ob überhaupt fehlende Werte vorliegen, `sum()`, um deren Anzahl und `which()`, um deren Position zu ermitteln.

```
> any(is.na(vec1))                  # gibt es fehlende Werte?
[1] TRUE

> sum(is.na(vec1))                  # wie viele?
[1] 2

> which(is.na(vec1))                # an welcher Position im Vektor?
[1] 3 6
```

2.11.2 Fehlende Werte ersetzen und umcodieren

Fehlende Werte werden bei der Dateneingabe in anderen Programmen oft mit Zahlen codiert, die keine mögliche Ausprägung einer Variable sind, z. B. mit 999. Bisweilen ist diese Codierung auch nicht einheitlich, sondern verwendet verschiedene Zahlen, etwa wenn Daten aus unterschiedlichen Quellen zusammengeführt werden. Bei der Verarbeitung von aus anderen Programmen übernommenen Datensätzen in R muss die Codierung fehlender Werte also ggf. angepasst werden (vgl. Abschn. 4.2).

Die Identifikation der zu ersetzenden Werte kann über `⟨Vektor⟩ %in% ⟨Menge⟩` erfolgen, wobei `⟨Menge⟩` ein Vektor mit allen Werten ist, die als fehlend gelten

[41] Der == Operator eignet sich nicht zur Prüfung auf fehlende Werte, da das Ergebnis von `⟨Wert⟩ == NA` selbst `NA` ist (vgl. Abschn. 2.11.3).

sollen (vgl. Abschn. 2.3.2). Der damit erzeugte Indexvektor lässt sich direkt an das Ergebnis von `is.na()` zuweisen, wodurch die zugehörigen Elemente auf `NA` gesetzt werden. Das Vorgehen bei Matrizen ist analog.

```
# fehlende Werte sind zunächst mit -999 und 999 codiert
> vec <- c(30, 25, 23, 21, -999, 999)       # Vektor mit fehlenden Werten
> is.na(vec) <- vec %in% c(-999, 999)       # ersetze missings durch NA
> vec
[1] 30 25 23 21 NA NA

# Matrix mit fehlenden Werten
> (mat <- matrix(c(30, 25, 23, 21, -999, 999), nrow=2, ncol=3))
     [,1]  [,2]  [,3]
[1,]   30    23    99
[2,]   25    21   999

> is.na(mat) <- mat %in% c(-999, 999)       # ersetze missings durch NA
> mat
     [,1]  [,2]  [,3]
[1,]   30    23    NA
[2,]   25    21    NA
```

2.11.3 Behandlung fehlender Werte bei der Berechnung einfacher Kennwerte

Wenn in einem Vektor oder einer Matrix fehlende Werte vorhanden sind, muss Funktionen zur Berechnung statistischer Kennwerte über ein Argument angegeben werden, wie mit ihnen zu verfahren ist. Andernfalls kann der Kennwert nicht berechnet werden, und das Ergebnis der Funktion ist seinerseits `NA`.[42] Allerdings können `NA` Einträge zunächst manuell entfernt werden, ehe die Daten an eine Funktion übergeben werden. Zu diesem Zweck existiert die Funktion `na.omit(⟨Vektor⟩)`, die das übergebene Objekt um fehlende Werte bereinigt ausgibt.

```
> sd(na.omit(vecNA))                # um NA bereinigten Vektor übergeben
[1] 5.125102

# fehlende Werte manuell entfernen
> goodIdx <- !is.na(vecNA)          # Indizes der nicht fehlenden Werte
> mean(vecNA[goodIdx])              # um NA bereinigten Vektor übergeben
[1] 24.33333
```

Die Behandlung fehlender Werte lässt sich in vielen Funktionen auch direkt über das Argument `na.rm` steuern. In der Voreinstellung `FALSE` sorgt es dafür, dass fehlende Werte nicht stillschweigend bei der Berechnung des Kennwertes ausgelassen

[42] Allgemein ist das Ergebnis aller Rechnungen `NA`, sofern der fehlende Wert für das Ergebnis relevant ist. Ist das Ergebnis auch ohne den fehlenden Wert eindeutig bestimmt, wird es ausgegeben – so erzeugt `TRUE | NA` die Ausgabe `TRUE`, da sich bei einem logischen ODER das zweite Argument nicht auf das Ergebnis auswirkt, wenn das erste WAHR ist.

werden, sondern das Ergebnis NA ist. Soll der Kennwert dagegen auf Basis der vorhandenen Werte berechnet werden, muss das Argument na.rm=TRUE gesetzt werden.

```
> sum(vecNA)
[1] NA

> sum(vecNA, na.rm=TRUE)
[1] 146

> apply(matNA, 1, mean)
[1] NA NA

> apply(matNA, 1, mean, na.rm=TRUE)
[1] 23.33333 25.33333
```

Auf die dargestellte Weise lassen sich fehlende Werte u. a. in den Funktionen sum(), prod(), range(), mean(), median(), quantile(), var(), sd(), cov() und cor() behandeln.

2.11.4 Behandlung fehlender Werte in Matrizen

Bei den Funktionen cov(⟨Vektor1⟩, ⟨Vektor2⟩) und cor(⟨Vektor1⟩,↘ →⟨Vektor2⟩) bewirkt das Argument na.rm=TRUE, dass ein aus zugehörigen Werten von ⟨Vektor1⟩ und ⟨Vektor2⟩ gebildetes Wertepaar nicht in die Berechnung von Kovarianz oder Korrelation eingeht, wenn wenigstens einer der beiden Werte NA ist.

Zur Behandlung fehlender Werte stehen bei cov() und cor() außer dem Argument na.rm=TRUE mit dem *zeilenweisen* bzw. *paarweisen* Fallausschluss weitere Möglichkeiten zur Verfügung. Sie sind aber erst relevant, wenn Kovarianz- bzw. Korrelationsmatrizen auf Basis von Matrizen mit mehr als zwei Spalten erstellt werden.

2.11.4.1 Zeilenweiser (fallweiser) Fallausschluss

Beim zeilenweisen Fallausschluss werden die Matrixzeilen komplett entfernt, in denen NA Werte auftauchen, ehe die Matrix für Berechnungen herangezogen wird. Weil eine Zeile oft allen an einem Beobachtungsobjekt erhobenen Daten entspricht, wird dies auch als fallweiser Fallausschluss bezeichnet. na.omit(⟨Matrix⟩) bereinigt eine Matrix mit fehlenden Werten um Zeilen, in denen NA Einträge auftauchen. Die Indizes der dabei ausgeschlossenen Zeilen werden der ausgegebenen Matrix als Attribut hinzugefügt. Mit der so gebildeten Matrix fließen im Beispiel die Zeilen 1 und 2 nicht mit in Berechnungen ein.

```
> ageNA     <- c(18, NA, 27, 22)
> DV1       <- c(NA, 1, 5, -3)
```

```
> DV2       <- c(9, 4, 2, 7)
> (matNA    <- cbind(ageNA, DV1, DV2))
     ageNA  DV1  DV2
[1,]    18   NA    9
[2,]    NA    1    4
[3,]    27    5    2
[4,]    22   -3    7

> na.omit(matNA)                        # Zeilen mit NA entfernen
     ageNA  DV1  DV2
[1,]    27    5    2
[2,]    22   -3    7

attr(,"na.action")
[1] 2 1

attr(,"class")
[1] "omit"

> colMeans(na.omit(matNA))              # Berechnung ohne NAs
ageNA  DV1  DV2
 24.5  1.0  4.5
```

Alternativ besteht die Möglichkeit, alle Zeilen mit fehlenden Werten anhand von `apply(is.na(⟨Matrix⟩), 1, any)` selbst festzustellen. Mit dem resultierenden logischen Indexvektor lassen sich die betroffenen Zeilen von weiteren Berechnungen ausschließen.

```
> (rowNAidx <- apply(is.na(matNA), 1, any))         # Zeilen mit NA feststellen
[1] TRUE TRUE FALSE FALSE

> matNA[!rowNAidx, ]                                # Zeilen mit NA entfernen
     ageNA  DV1  DV2
[1,]    27    5    2
[2,]    22   -3    7
```

Bei den Funktionen `cov()` und `cor()` bewirkt bei der Berechnung von Kovarianz- und Korrelationsmatrizen mit mehr als zwei Variablen das Argument `use="complete.obs"` den fallweisen Ausschluss fehlender Werte. Seine Verwendung hat denselben Effekt wie die vorherige Reduktion der Matrix um Zeilen, in denen fehlende Werte auftauchen.

```
> cov(matNA, use="complete.obs")
       age  DV1    DV2
age   12.5   20  -12.5
DV1   20.0   32  -20.0
DV2  -12.5  -20   12.5

# beide Arten des fallweisen Ausschlusses erzielen dasselbe Ergebnis
> all.equal(cov(matNA, use="complete.obs"), cov(na.omit(matNA)))
[1] TRUE
```

2.11.4.2 Paarweiser Fallausschluss

Beim paarweisen Fallausschluss werden die Werte einer auch NA beinhaltenden Zeile soweit als möglich in Berechnungen berücksichtigt, die Zeile wird also nicht vollständig ausgeschlossen. Welche Werte einer Zeile Verwendung finden, hängt von der konkreten Auswertung ab. Der paarweise Fallausschluss wird im Fall der Berechnung der Summe oder des Mittelwertes über Zeilen oder Spalten mit dem Argument na.rm=TRUE realisiert, das alle Werte außer NA einfließen lässt.

```
> rowMeans(matNA)
[1] NA NA 11.333333 8.666667

> rowMeans(mat, na.rm=TRUE)
[1] 13.500000 2.500000 11.333333 8.666667
```

Bei der Berechnung von Kovarianz- und Korrelationsmatrizen für mehr als zwei Variablen mit cov() und cor() bewirkt das Argument use="pairwise.complete.obs" den paarweisen Ausschluss fehlender Werte. Es wird dann bei der Berechnung jeder Kovarianz geprüft, ob pro Zeile in den zugehörigen beiden Spalten ein gültiges Wertepaar existiert und dieses ggf. verwendet. Anders als beim fallweisen Ausschluss geschieht dies also auch dann, wenn in derselben Zeile Werte anderer Variablen fehlen, dies für die zu berechnende Kovarianz aber irrelevant ist.

Angewendet auf die Daten in matNA bedeutet das beispielsweise, dass beim fallweisen Ausschluss das von VP 1 gelieferte Wertepaar nicht in die Berechnung der Kovarianz von ageNA und DV2 einfließt, weil der Wert für DV1 bei dieser VP fehlt. Beim paarweisen Ausschluss werden diese Werte dagegen berücksichtigt. Lediglich bei der Berechnung der Kovarianz von DV1 und DV2 werden keine Daten der ersten VP verwendet, weil ein Wert für DV1 von ihr fehlt.

```
> cov(matNA, use="pairwise.complete.obs")
              ageNA     DV1          DV2
ageNA      20.33333      20   -16.000000
  DV1      20.00000      16   -10.000000
  DV2     -16.00000     -10     9.666667
```

Ob fehlende Werte fall- oder paarweise ausgeschlossen werden sollten, hängt u. a. von den Ursachen ab, warum manche Untersuchungseinheiten unvollständige Daten geliefert haben und andere nicht. Insbesondere stellt sich die Frage, ob Untersuchungseinheiten mit fehlenden Werten systematisch andere Eigenschaften haben, sodass von ihren Daten generell kein Gebrauch gemacht werden sollte.

2.11.5 Behandlung fehlender Werte beim Sortieren von Daten

Beim Sortieren von Daten mit sort() und order() wird die Behandlung fehlender Werte mit dem Argument na.last kontrolliert, das auf NA, TRUE oder FALSE gesetzt werden kann. Bei sort() ist na.last per Voreinstellung auf NA gesetzt und

sorgt so dafür, dass fehlende Werte entfernt werden. Bei `order()` ist die Voreinstellung `TRUE`, wodurch fehlende Werte ans Ende platziert werden. Auf `FALSE` gesetzt bewirkt `na.last` die Platzierung fehlender Werte am Anfang.

2.12 Zeichenketten verarbeiten

Obwohl Zeichenketten bei der numerischen Auswertung von Daten oft eine Nebenrolle spielen und zuvorderst in Form von Bezeichnungen für Variablen oder Gruppen in Erscheinung treten, ist es bisweilen hilfreich, sie flexibel erstellen, manipulieren und ausgeben zu können.[43]

2.12.1 Objekte in Zeichenketten umwandeln

Mit der `toString(⟨Objekt⟩)` Funktion lassen sich Ergebnisse beliebiger Berechnungen in Zeichenketten umwandeln. Als Argument wird ein Objekt erwartet – typischerweise die Ausgabe einer Funktion. Das Ergebnis ist eine einzelne Zeichenkette, die als Inhalt die normalerweise auf der Konsole erscheinende Ausgabe von `⟨Objekt⟩` hat. Dabei werden einzelne Elemente der Ausgabe innerhalb der Zeichenkette durch Komma mit folgendem Leerzeichen getrennt. Komplexe Objekte (z. B. Matrizen) werden dabei wie Vektoren verarbeitet.

```
> randVals <- round(rnorm(5), 2)
> toString(randVals)
[1] "-0.03, 1.01, -0.52, -1.03, 0.18"
```

Die `formatC()` Funktion ist auf die Umwandlung von Zahlen in Zeichenketten spezialisiert und bietet sich vor allem für die formatierte Ausgabe von Dezimalzahlen an.

```
> formatC(x=⟨Zahl⟩, digits=⟨Dezimalstellen⟩, width=⟨Breite⟩,
+         flag="⟨Modifikation⟩", format="⟨Zahlentyp⟩")
```

Ist `x` eine Dezimalzahl, wird sie mit `digits` vielen Dezimalstellen ausgegeben. Die Angabe von `digits` fügt ganzen Zahlen keine Dezimalstellen hinzu, allerdings verbreitert sich die ausgegebene Zeichenkette auf `digits` viele Zeichen, indem `x` entsprechend viele Leerzeichen vorangestellt werden. Sollen der Zahl stattdessen Nullen vorangestellt werden, ist `flag="0"` zu setzen. Linksbündig ausgerichtete Zeichenketten sind mit `flag="-"` zu erreichen. Die Länge der Zeichenkette lässt sich auch unabhängig von der Zahl der Dezimalstellen mit dem Argument `width` kontrollieren. Schließlich ermöglicht `format` die Angabe, was für ein Zahlentyp bei `x` vorliegt, insbesondere ob es eine ganze Zahl (`"d"`) oder eine Dezimalzahl ist.

[43] Das Paket `stringr` (Wickham, 2010) stellt für viele der im Folgenden aufgeführten Funktionen – und für einige weitere Aufgaben – Alternativen bereit, die den Umgang mit Zeichenketten erleichtern und konsistenter gestalten sollen.

Im letztgenannten Fall kann die Ausgabeform etwa mit `"f"` wie gewohnt erfolgen (z. B. `"1.234"`) oder mit `"e"` in wissenschaftlicher Notation (z. B. `"1.23e+03"`) – für weitere Möglichkeiten vgl. `?formatC`.

```
> formatC(3, digits=5, format="d")
[1] "       3"

> formatC(c(1, 2.345), width=5, format="f")
[1] "1.0000" "2.3450"
```

2.12.2 Zeichenketten erstellen und ausgeben

Die einfachste Möglichkeit zum Erstellen eigener Zeichenketten ist ihre manuelle Eingabe auf der Konsole oder im Editor. Für Vektoren von Zeichenketten ist dabei zu beachten, dass der `length()` Befehl jede Zeichenkette als ein Element betrachtet. Dagegen gibt die `nchar("⟨Zeichenkette⟩")` Funktion die Wortlänge jedes Elements an, aus wie vielen einzelnen Zeichen jede Zeichenkette des Vektors also besteht.

```
> length("ABCDEF")
[1] 1

> nchar("ABCDEF")
[1] 6

> nchar(c("A", "BC", "DEF"))
[1] 1 2 3
```

Die Methode, Zeichenketten manuell in Vektoren zu erstellen, stößt jedoch dort schnell an ihre Grenzen, wo sie von Berechnungen abhängen sollen oder viele Zeichenketten nach demselben Muster erzeugt werden müssen. Die Funktionen `paste()` und `sprintf()` sind hier geeignete Alternativen.

Mit `paste()` lassen sich Zeichenketten mit einem bestimmten Aufbau erzeugen, indem verschiedene Komponenten aneinandergehängt werden, die etwa aus einem gemeinsamen Präfix und unterschiedlicher laufender Nummer bestehen können.

```
> paste(⟨Objekt1⟩, ⟨Objekt2⟩, ..., sep=" ", collapse=NULL)
```

Die ersten Argumente von `paste()` sind Objekte, deren Elemente jeweils die Bestandteile der zu erstellenden Zeichenketten ausmachen und zu diesem Zweck aneinandergefügt werden. Das erste Element des ersten Objekts wird dazu mit den ersten Elementen der weiteren Objekte verbunden, ebenso die jeweils zweiten und folgenden Elemente. Das Argument `sep` kontrolliert, welche Zeichen jeweils zwischen Elementen aufeinander folgender Objekte einzufügen sind – in der Voreinstellung ist dies das Leerzeichen. In der Voreinstellung `collapse=NULL` ist das Ergebnis ein Vektor aus Zeichenketten, wobei jedes seiner Elemente aus der Kombination jeweils eines Elements aus jedem übergebenen Objekt besteht. Hierbei werden unterschiedlich lange Vektoren ggf. zyklisch verlängert (vgl. Abschn. 2.5.4.1). Wird

stattdessen für `collapse` eine Zeichenfolge übergeben, ist das Ergebnis eine einzelne Zeichenkette, deren Bestandteile durch diese Zeichenfolge getrennt sind.

```
> paste("group", LETTERS[1:5], sep="_")
[1] "group_A" "group_B" "group_C" "group_D" "group_E"

# Farben der Default-Farbpalette
> paste(1:5, palette()[1:5], sep=": ")
[1] "1: black" "2: red" "3: green3" "4: blue" "5: cyan"

> paste(1:5, letters[1:5], sep=".", collapse=" ")
[1] "1.a 2.b 3.c 4.d 5.e"
```

Die an die gleichnamige Funktion der Programmiersprache C angelehnte Funktion `sprintf()` erzeugt Zeichenketten, deren Aufbau durch zwei Komponenten bestimmt wird: Einerseits durch einen die Formatierung und feste Elemente definierenden Teil (den sog. *format string*), andererseits durch eine Reihe von Objekten, deren Werte an festgelegten Stellen des format strings einzufügen sind.

```
> sprintf(fmt="⟨format string⟩", ⟨Objekt1⟩, ⟨Objekt2⟩, ...)
```

Das Argument `fmt` erwartet eine Zeichenkette aus festen und variablen Elementen. Gewöhnliche Zeichen werden als feste Elemente interpretiert und tauchen unverändert in der erzeugten Zeichenkette auf. Variable Elemente werden durch das Prozentzeichen `%` eingeleitet, auf das ein Buchstabe folgen muss, der die Art des hier einzufügenden Wertes definiert. So gibt etwa `%d` an, dass hier ein ganzzahliger Wert einzufügen ist, `%f` dagegen weist auf eine Dezimalzahl hin und `%s` auf eine Zeichenfolge. Das Prozentzeichen selbst wird durch `%%` ausgegeben, doppelte Anführungszeichen durch `\"`,[44] Tabulatoren durch `\t` und Zeilenumbrüche durch `\n` (vgl. `?Quotes`).

Für jedes durch ein Prozentzeichen definierte Feld muss nach `fmt` ein passendes Objekt genannt werden, dessen Wert an der durch `%` bezeichneten Stelle eingefügt wird. Die Entsprechung zwischen variablen Feldern und Objekten wird über deren hergestellt, der Wert des ersten Objekts wird also an der Stelle des ersten variablen Elements eingefügt, etc.

```
> N     <- 20
> grp   <- "A"
> M     <- 14.2
> sprintf("For %d particpants in group %s, the mean was %f", N, grp, M)
[1] "For 20 particpants in group A, the mean was 14.200000"
```

Format strings erlauben eine weitergehende Formatierung der Ausgabe, indem zwischen dem `%` und dem folgenden Buchstaben Angaben gemacht werden, die sich z. B. auf die Anzahl der auszugebenden Dezimalstellen beziehen können. Für detailliertere Informationen vgl. `?sprintf`.

```
> sprintf("%.3f", 1.23456)          # begrenze Ausgabe auf 3 Dezimalstellen
[1] "1.234"
```

[44] Alternativ kann `fmt` in einfache Anführungszeichen `'⟨format string⟩'` gesetzt werden, innerhalb derer sich dann auch doppelte Anführungszeichen ohne voranstehendes \ Symbol befinden können (vgl. Abschn. 1.3.5, Tabelle 1.2, Anmerkung b).

Um mehrere Zeichenketten kombiniert als eine einzige Zeichenkette lediglich auf der R Konsole auszugeben (und nicht in einem Vektor zu speichern), kann die `cat()` (concatenate) Funktion verwendet werden. Sie erlaubt auch eine gewisse Formatierung – etwa in Form von Zeilenumbrüchen durch die Escape-Sequenz `\n` oder `\t` für Tabulatoren.

```
> cat("⟨Zeichenkette 1⟩", "⟨Zeichenkette 2⟩", ..., sep=" ")
```

`cat()` kombiniert die übergebenen Zeichenketten durch Verkettung zunächst zu einer einzelnen, wobei zwischen den Zeichenketten das unter `sep` genannte Trennzeichen eingefügt wird. Numerische Variablen werden hierbei automatisch in Zeichenketten konvertiert. Die Ausgabe von `cat()` unterscheidet sich in zwei Punkten von der üblichen Ausgabe einer Zeichenkette: Zum einen wird die Zeichenkette nicht in Anführungszeichen gesetzt. Zum anderen wird am Anfang jeder Zeile auf die Ausgabe der Position des zu Zeilenbeginn stehenden Wertes, etwa `[1]`, verzichtet.

```
> cVar <- "A string"
> cat(cVar, "with\n", 4, "\nwords\n", sep="+")
A string+with
+4+
words
```

In der Voreinstellung setzen die meisten Ausgabefunktionen von R Zeichenketten in Anführungszeichen. In `print()` lässt sich dies mit dem Argument `quote=`↘
→`FALSE` verhindern, allgemein hat die `noquote("⟨Zeichenkette⟩")` Funktion denselben Effekt.

```
> print(cVar, quote=FALSE)
[1] A string

> noquote(cVar)
[1] A string
```

2.12.3 Zeichenketten manipulieren

Die Funktionen `tolower("⟨Zeichenkette⟩")` und `toupper("⟨Zeichenkette⟩`↘
→`")` konvertieren die Buchstaben in den übergebenen (Vektoren von) Zeichenketten in Klein- bzw. Großbuchstaben.

```
> tolower(c("A", "BC", "DEF"))
[1] "a" "bc" "def"

> toupper(c("ghi", "jk", "i"))
[1] "GHI" "JK" "I"
```

Aus Zeichenketten lassen sich mit `substring()` konsekutive Teilfolgen von Zeichen extrahieren.

```
> substring(text="⟨Zeichenkette⟩", first=⟨Beginn⟩, last=⟨Ende⟩)
```

Aus den Elementen des für `text` angegebenen Vektors von Zeichenketten wird jeweils jene Zeichenfolge extrahiert, die beim Buchstaben an der Stelle `first` beginnt und mit dem Buchstaben an der Stelle `last` endet. Sollte eine Zeichenkette weniger als `first` oder `last` Buchstaben umfassen, werden nur so viele ausgeben, wie tatsächlich vorhanden sind – ggf. eine leere Zeichenkette.

```
> substring(c("ABCDEF", "GHIJK", "LMNO", "PQR"), first=4, last=5)
[1] "DE" "JK" "O" ""
```

Mit der `strsplit()` Funktion (string split) ist es möglich, eine einzelne Zeichenkette in mehrere Teile zu zerlegen.

```
> strsplit(x="⟨Zeichenkette⟩", split="⟨Zeichenkette⟩", fixed=FALSE)
```

Die Elemente des für `x` übergebenen Vektors werden dafür nach Vorkommen der unter `split` genannten Zeichenkette durchsucht, die als Trennzeichen interpretiert wird. Die Zeichenfolgen links und rechts von `split` machen die Komponenten der Ausgabe aus, die aus einer Liste von Vektoren von Zeichenketten besteht – eine Komponente für jedes Element des Vektors von Zeichenketten `x`. In der Voreinstellung `split=NULL` werden die Elemente von `x` in einzelne Zeichen zerlegt.[45] Die `strsplit()` Funktion ist damit die Umkehrung von `paste()`. Das Argument `fixed` bestimmt, ob `split` i. S. eines sog. *regulären Ausdrucks* interpretiert werden soll (Voreinstellung `FALSE`, vgl. Abschn. 2.12.4) oder als exakt die übergebene Zeichenfolge selbst (`TRUE`).

```
> strsplit(c("abc_def_ghi", "jkl_mno"), split="_")
[[1]]
[1] "abc" "def" "ghi"

[[2]]
[1] "jkl" "mno"

> strsplit("Xylophon", split=NULL)
[[1]]
[1] "X" "y" "l" "o" "p" "h" "o" "n"
```

Mit der in der Hilfe-Seite von `strsplit()` definierten `strReverse("⟨Zeichen↘`
`→kette⟩")` Funktion (string reverse, vgl. Abschn. 12.2) wird die Reihenfolge der Zeichen innerhalb einer Zeichenkette umgekehrt. Dies ist deswegen nicht mit `rev↘`
`→()` möglich, weil eine Zeichenkette ein einzelnes Element eines Vektors darstellt.

```
# Definition der strReverse() Funktion
> strReverse <- function(x) {
+               sapply(lapply(strsplit(x, NULL), rev), paste, collapse="") }

> strReverse(c("Lorem", "ipsum", "dolor", "sit"))
[1] "meroL" "muspi" "rolod" "tis"
```

[45] Die Ausgabe ist ggf. mit `unlist()` (vgl. Abschn. 3.1.2) in einen Vektor, oder mit `do.call("↘`
`→cbind", ⟨Liste⟩)` bzw. `do.call("rbind", ⟨Liste⟩)` in eine Matrix umzuwandeln, wenn die Listenkomponenten dieselbe Länge besitzen (vgl. Abschn. 3.4.1).

2.12.4 Zeichenfolgen finden

Die Suche nach bestimmten Zeichenfolgen innerhalb von Zeichenketten ist mit den Funktionen `match()`, `pmatch()` und `grep()` möglich. Soll geprüft werden, ob die in einem Vektor `x` enthaltenen Elemente jeweils eine exakte Übereinstimmung in den Elementen eines Vektors `table` besitzen, ist `match()` anzuwenden. Beide Objekte müssen nicht unbedingt Zeichenketten sein, werden aber intern zu solchen konvertiert.

```
> match(x=⟨gesuchte Werte⟩, table=⟨Objekt⟩)
```

Die Ausgabe gibt für jedes Element von `x` die erste Position im Objekt `table` an, an der es dort ebenfalls vorhanden ist. Enthält `table` kein mit `x` übereinstimmendes Element, ist die Ausgabe an dieser Stelle `NA`.

Die fast identische `pmatch()` Funktion unterscheidet sich darin, dass die Elemente von `table` nicht nur auf exakte Übereinstimmung getestet werden: Findet sich für ein Element von `x` ein identisches Element in `table`, ist der Index das Ergebnis, an dem dieses Element zum ersten Mal vorkommt. Andernfalls wird in `table` nach teilweisen Übereinstimmungen in dem Sinne gesucht, dass auch eine Zeichenkette zu einem Treffer führt, wenn sie mit jener aus `x` beginnt, sofern es nur eine einzige solche Zeichenkette in `table` gibt.

```
> match(c("abc", "de", "f", "h"), c("abcde", "abc", "de", "fg", "ih"))
[1] 2 3 NA NA

> pmatch(c("abc", "de", "f", "h"), c("abcde", "abc", "de", "fg", "ih"))
[1] 2 3 4 NA
```

Die `grep()` Funktion ähnelt dem gleichlautenden POSIX-Befehl Unix-artiger Betriebssysteme und bietet stark erweiterte Suchmöglichkeiten.

```
> grep(pattern="⟨Suchmuster⟩", x="⟨Zeichenkette⟩")
```

Unter `pattern` ist ein Muster anzugeben, das die zu suchende Zeichenfolge definiert. Obwohl hier auch einfach eine bestimmte Zeichenfolge übergeben werden kann, liegt die Besonderheit darin, dass `pattern` reguläre Ausdrücke akzeptiert. Mit regulären Ausdrücken lassen sich auch Muster von Zeichenfolgen charakterisieren wie „ein A gefolgt von einem B oder C und einem Leerzeichen“: `"A[BC][[:blank:]]"` (vgl. `?regex`, Friedl, 2006 und speziell für die Anwendung in R Spector, 2008). Der zu durchsuchende Vektor von Zeichenketten wird unter `x` genannt.

Die Ausgabe besteht in einem Vektor von Indizes derjenigen Elemente von `x`, die das gesuchte Muster enthalten. Alternativ gibt die ansonsten genauso zu verwendende Funktion `grepl()` einen logischen Indexvektor aus, der für jedes Element von `x` angibt, ob es `pattern` enthält.

```
> grep("A[BC][[:blank:]]", c("AB ", "AB", "AC ", "A "))
[1] 1 3

> grepl("A[BC][[:blank:]]", c("AB ", "AB", "AC ", "A "))
[1] TRUE FALSE TRUE FALSE
```

Um mithilfe von regulären Ausdrücken definierte Zeichenfolgen nicht nur finden, sondern später auch aus Zeichenketten entfernen zu können (s. u.), ist neben der Information, *ob* eine Zeichenkette die gesuchte Zeichenfolge enthält, auch ggf. die Information notwendig, an welcher Stelle sie auftaucht. Dies lässt sich mit `regexpr()` ermitteln.

```
> regexpr(pattern="⟨Suchmuster⟩", text="⟨Zeichenkette⟩")
```

Die Argumente `pattern` und `text` haben jeweils dieselbe Bedeutung wie `pattern` und `x` von `grep()`. Das Ergebnis ist ein numerischer Vektor mit so vielen Elementen wie jene von `text`. Enthält ein Element von `text` das Suchmuster nicht, ist das Ergebnis an dieser Stelle `-1`. Andernfalls ist das Ergebnis die erste Stelle des zugehörigen Elements von `text`, an der das gefundene Suchmuster dort beginnt. Der ausgegebene numerische Vektor besitzt weiterhin das Attribut `match.length`, das seinerseits ein numerischer Vektor ist und codiert, wie viele Zeichen die Zeichenfolge umfasst, auf die das Suchmuster zutrifft. Auch hier steht die `-1` für den Fall, dass sich das Suchmuster nicht in der Zeichenkette findet. Das Ergebnis eignet sich besonders, um mit der `substr()` Funktion weiterverarbeitet zu werden, da sich aus ihm die Informationen für deren Argumente `first` und `last` leicht bestimmen lassen.

```
> pat       <- "[[:upper:]]+"                # suche nach Großbuchstaben
> txt       <- c("abcDEFG", "ABCdefg", "abcdefg")
> (start    <- regexpr(pat, txt))            # Start und Länge der Fundstellen
[1] 4 1 -1

attr(,"match.length")
[1] 4 3 -1

> len <- attr(start, "match.length")           # nur Länge der Fundstellen
> end <- start + len - 1                       # letzte Zeichen der Fundstellen
> substring(txt, start, end)                   # extrahiere Fundstellen
[1] "DEFG" "ABC" ""
```

Im Unterschied zu `regexpr()` berücksichtigt die ansonsten gleich zu verwendende `gregexpr()` Funktion nicht nur das erste Auftreten von `pattern` in `text`, sondern auch ggf. spätere. Die Ausgabe ist eine Liste mit so vielen Komponenten, wie `text` Elemente besitzt.

Da die Syntax regulärer Ausdrücke recht komplex ist, bereitet dem nicht mit der Materie vertrauten Anwender u. U. die Suche schon nach einfachen Mustern Schwierigkeiten. Eine Vereinfachung bietet die Funktion `glob2rx()`, mit der Muster von Zeichenfolgen mithilfe gebräuchlicherer Platzhalter (sog. *wildcards* bzw. *Globbing*-Muster) beschrieben und in einen regulären Ausdruck umgewandelt werden können. So steht z. B. der Platzhalter `?` für ein beliebiges einzelnes Zeichen, `*` für eine beliebige Zeichenkette.

```
> glob2rx(pattern="⟨Muster mit Platzhaltern⟩")
```

Das Argument `pattern` akzeptiert einen Vektor, dessen Elemente Zeichenfolgen aus Buchstaben und Platzhaltern sind. Die Ausgabe besteht aus einem Vektor mit

regulären Ausdrücken, wie sie z. B. in der `grep()` Funktion angewendet werden können.

```
> glob2rx("asdf*.txt")          # Namen, die mit asdf beginnen und .txt enden
[1] "^asdf.*\\.txt$"
```

2.12.5 Zeichenfolgen ersetzen

Wenn in Zeichenketten nach bestimmten Zeichenfolgen gesucht wird, dann häufig, um sie durch andere zu ersetzen. Dies ist etwa möglich, indem dem Ergebnis von `substr()` ein passender Vektor von Zeichenketten zugewiesen wird – dessen Elemente ersetzen dann die durch `first` und `last` begrenzten Zeichenfolgen in den Elementen von `text`. Dabei ist es notwendig, dass für `text` ein bereits bestehendes Objekt übergeben wird, das dann der Änderung unterliegt.

```
> charVec <- c("ABCDEF", "GHIJK", "LMNO", "PQR")
> substring(charVec, 4, 5) <- c("..", "xx", "++", "**"); charVec
[1] "ABC..F" "GHIxx" "LMN+" "PQR"
```

Auch die `sub()` (substitute) und `gsub()` Funktionen dienen dem Zweck, durch ein Muster definierte Zeichenfolgen innerhalb von Zeichenketten auszutauschen.

```
> sub(pattern="⟨Suchmuster⟩", replacement="⟨Ersatz⟩",
+       x="⟨Zeichenkette⟩")
```

Für `pattern` kann ein regulärer Ausdruck übergeben werden, dessen Vorkommen in den Elementen von `x` durch die unter `replacement` genannte Zeichenfolge ersetzt werden. Wenn `pattern` in einem Element von `x` mehrfach vorkommt, wird es nur beim ersten Auftreten ersetzt.

```
> sub("em", "XX", "Lorem ipsum dolor sit Lorem ipsum")
[1] "LorXX ipsum dolor sit Lorem ipsum"
```

Im Unterschied zu `sub()` ersetzt die ansonsten gleich zu verwendende `gsub()` Funktion `pattern` nicht nur beim ersten Auftreten in `x` durch `replacement`, sondern überall.

```
> gsub("em", "XX", "Lorem ipsum dolor sit Lorem ipsum")
[1] "LorXX ipsum dolor sit LorXX ipsum"
```

2.12.6 Zeichenketten als Befehl ausführen

Durch die Kombination der Funktionen `parse()` und `eval()` lassen sich Zeichenketten als Befehle interpretieren und wie direkt eingegebene Befehle ausführen. Dieses Zusammenspiel ermöglicht es, in Abhängigkeit von vorherigen Auswertungen einen nachfolgend benötigten Befehl zunächst als Zeichenkette zu erstellen und dann auszuführen.

```
> parse(file="⟨Pfad und Dateiname⟩", text="⟨Zeichenkette⟩")
```

Hierfür ist zunächst mit `parse()` eine für das Argument `text` zu übergebende Zeichenkette in ein weiter interpretierbares Objekt umzuwandeln.[46] Ist `text` ein Vektor von Zeichenketten, wird jedes Element als ein Befehl verstanden. Alternativ kann mit `file` eine Datei oder sonstige Quelle genannt werden, die eine solche Zeichenkette enthält (vgl. Abschn. 4.2.3).

```
> obj1 <- parse(text="3 + 4")
> obj2 <- parse(text=c("vec <- c(1, 2, 3)", "vec^2"))
```

Das Ausführen eines mit `parse()` erstellten Objekts geschieht mit `eval(⟨expression⟩)`.

```
> eval(obj1)
[1] 7

> eval(obj2)
[1] 1 4 9
```

2.13 Datum und Uhrzeit

Insbesondere bei der Analyse von Zeitreihen[47] ist es sinnvoll, Zeit- und Datumsangaben in einer Form zu speichern, die es erlaubt, solche Werte in natürlicher Art für Berechnungen zu nutzen – etwa um über die Differenz zweier Uhrzeiten die zwischen ihnen verstrichene Zeit ebenso zu ermitteln wie die zwischen zwei Datumsangaben liegende Anzahl von Tagen. R bietet solche Möglichkeiten mithilfe besonderer Klassen.[48]

2.13.1 Datumsangaben erstellen und formatieren

Objekte der Klasse `Date` codieren ein Datum mittels der seit einem Stichtag (meist der 1. Januar 1970) verstrichenen Anzahl von Tagen und können Tag, Monat und Jahr eines Zeitpunkts ausgeben. Diese numerische Repräsentation wird sichtbar, wenn `Date` Objekte mit `as.numeric()` umgewandelt werden. Das aktuelle Datum

[46] Solcherart erstellte Objekte können mit `deparse()` wieder in Zeichenketten umgewandelt werden.

[47] Für die Auswertung von Zeitreihen vgl. Shumway und Stoffer (2011) sowie den Abschnitt `Time Series Analysis` der Task Views (R Development Core Team, 2011a).

[48] Für eine einführende Behandlung der vielen für Zeitangaben existierenden Subtilitäten vgl. Grothendieck und Petzoldt (2004) sowie `?DateTimeClasses`. Der Umgang mit Zeit- und Datumsangaben wird durch Funktionen des Pakets `lubridate` (Grolemund & Wickham, 2011) erleichtert. Das Paket `timeDate` (Würtz & Chalabi, 2011) enthält viele weiterführende Funktionen zur Verarbeitung solcher Daten.

unter Beachtung der Zeitzone nennt `Sys.Date()` in Form eines solchen Objekts. Um selbst ein Datum zu erstellen, ist `as.Date()` zu verwenden.

```
> as.Date(x="⟨Datumsangabe⟩", format="⟨format string⟩")
```

Die Datumsangabe für `x` ist eine Zeichenkette, die ein Datum in einem Format nennt, das unter `format` als format string zu spezifizieren ist. In einer solchen Zeichenkette stehen `%⟨Buchstabe⟩` Kombinationen als Platzhalter für den einzusetzenden Teil einer Datumsangabe, sonstige Zeichen i. d. R. für sich selbst. Voreinstellung ist `"%Y-%m-%d"`, wobei `%Y` für die vierstellige Jahreszahl, `%m` für die zweistellige Zahl des Monats und `%d` für die zweistellige Zahl der Tage steht.[49] In diesem Format erfolgt auch die Ausgabe, die sich jedoch mit `format(⟨Date-Objekt⟩, format`↘
`→="⟨format string⟩")` kontrollieren lässt.

```
> Sys.Date()
[1] "2009-02-09"

> (myDate <- as.Date("01.11.1974", format="%d.%m.%Y"))
[1] "1974-11-01"

> format(myDate, format="%d.%m.%Y")
[1] "01.11.1974"
```

2.13.2 Uhrzeit

Objekte der Klasse `POSIXct` (calendar time) repräsentieren neben dem Datum gleichzeitig die Uhrzeit eines Zeitpunkts als Anzahl der Sekunden, die seit einem Stichtag (meist der 1. Januar 1970) verstrichen ist, besitzen also eine Genauigkeit von einer Sekunde. Sie berücksichtigen dabei die Zeitzone sowie die Unterscheidung von Sommer- und Winterzeit. Die Funktion `Sys.time()` gibt das aktuelle Datum nebst Uhrzeit in Form eines solchen Objekts aus, allerdings für eine Standard-Zeitzone. Das Ergebnis muss deshalb mit den u. g. Funktionen in die aktuelle Zeitzone konvertiert werden. Alternativ gibt `date()` Datum und Uhrzeit mit englischen Abkürzungen für Wochentag und Monat als Zeichenkette aus, wobei die aktuelle Zeitzone berücksichtigt wird.

```
> Sys.time()
[1] "2009-02-07 09:23:02 CEST"

> date()
[1] "Sat Feb 7 09:23:02 2009"
```

Objekte der Klasse `POSIXlt` (local time) speichern dieselbe Information, allerdings nicht in Form der seit einem Stichtag verstrichenen Sekunden, sondern als

[49] Vergleiche Abschn. 2.12.2 sowie `?strptime` für weitere mögliche Elemente des format strings. Diese Hilfe-Seite erläutert auch, wie mit Namen für Wochentage und Monate in unterschiedlichen Sprachen umzugehen ist.

Liste mit benannten Komponenten: Dies sind numerische Vektoren u. a. für die Sekunden (`sec`), Minuten (`min`) und Stunden (`hour`) der Uhrzeit sowie für Tag (`mday`), Monat (`mon`) und Jahr (`year`) des Datums. Zeichenketten lassen sich analog zu `as.Date()` mit `as.POSIXct()` bzw. mit `as.POSIXlt()` in entsprechende Objekte konvertieren, `strptime()` erzeugt ebenfalls ein `POSIXlt` Objekt.

```
> as.POSIXct(   x="⟨Datum und Uhrzeit⟩", format="⟨format string⟩")
> as.POSIXlt(   x="⟨Datum und Uhrzeit⟩", format="⟨format string⟩")
>   strptime(   x="⟨Datum und Uhrzeit⟩", format="⟨format string⟩")
```

Voreinstellung für den format string bei `as.POSIXlt()` und bei `as.POSIXct()` ist `"%Y-%m-%d %H:%M:%S"`, wobei `%H` für die zweistellige Zahl der Stunden im 24 h-Format, `%M` für die zweistellige Zahl der Minuten und `%S` für die zweistellige Zahl der Sekunden des Datums stehen (vgl. Fußnote 49).

```
> (myTime <- as.POSIXct("2009-02-07 09:23:02"))
[1] "2009-02-07 09:23:02 CET"

> charDates     <- c("05.08.1972, 03:37", "31.03.1981, 12:44")
> (lDates       <- strptime(charDates, format="%d.%m.%Y, %H:%M"))
[1] "1972-08-05 03:37:00" "1981-03-31 12:44:00"

> lDates$mday                                   # Tag isoliert
[1] 5 31

> lDates$hour                                   # Stunde isoliert
[1] 3 12
```

`POSIXct` Objekte können besonders einfach mit der Funktion `ISOdate()` erstellt werden, die intern auf `strptime()` basiert, aber keinen format string benötigt.

```
> ISOdate(year=⟨Jahr⟩, month=⟨Monat⟩, day=⟨Tag⟩,
+          hour=⟨Stunde⟩, min=⟨Minute⟩, sec=⟨Sekunde⟩, tz="⟨Zeitzone⟩")
```

Für die sich auf Datum und Uhrzeit beziehenden Argumente können Zahlen im 24 h-Format angegeben werden, wobei 12:00:00 h Voreinstellung für die Uhrzeit ist. Mit `tz` lässt sich die Zeitzone im Form der standardisierten Akronyme festlegen, Voreinstellung ist hier `"GMT"`.

```
# Zeitzone: Central European Time
> ISOdate(2010, 6, 30, 17, 32, 10, tz="CET")
[1] "2010-06-30 17:32:10 CEST"
```

Auch Objekte der Klassen `POSIXct` und `POSIXlt` können mit `format()` in der gewünschten Formatierung ausgegeben werden.

```
> format(myTime, "%H:%M:%S")                    # nur Stunden, Minuten, Sekunden
[1] "09:23:02"

> format(lDates, "%d.%m.%Y")                    # nur Tag, Monat, Jahr
[1] "05.08.1972" "31.03.1981"
```

2.13.3 Berechnungen mit Datum und Uhrzeit

Aus Datumsangaben der Klasse `Date`, `POSIXct` und `POSIXlt` lassen sich bestimmte weitere Informationen in Form von Zeichenketten extrahieren, etwa der Wochentag mit `weekdays(⟨Datum⟩)`, der Monat mit `months(⟨Datum⟩)` oder das Quartal mit `quarters(⟨Datum⟩)`.

```
> weekdays(lDates)
[1] "Samstag"  "Dienstag"

> months(lDates)
[1] "August" "März"
```

Objekte der Klasse `Date`, `POSIXct` und `POSIXlt` verhalten sich in vielen arithmetischen Kontexten in natürlicher Weise, da die sinnvoll für Daten interpretierbaren Rechenfunktionen besondere Methoden für sie besitzen (vgl. Abschn. 12.2.5): So werden zu `Date` Objekten addierte Zahlen als Anzahl von Tagen interpretiert; das Ergebnis ist ein Datum, das entsprechend viele Tage vom `Date` Objekt abweicht. Die Differenz zweier `Date` Objekte besitzt die Klasse `difftime` und wird als Anzahl der Tage ausgegeben, die vom zweiten zum ersten Datum vergehen. Hierbei ergeben sich negative Zahlen, wenn das erste Datum zeitlich vor dem zweiten liegt.[50] Ebenso wie Zahlen lassen sich auch `difftime` Objekte zu `Date` Objekten addieren.

```
> myDate + 365
[1] "1975-11-01"

> (diffDate <- as.Date("1976-06-19") - myDate)
Time difference of 596 days

> myDate + diffDate
[1] "1976-06-19"
```

Zu Objekten der Klasse `POSIXlt` oder `POSIXct` addierte Zahlen werden als Sekunden interpretiert. Aus der Differenz zweier solcher Objekte entsteht ein Objekt der Klasse `difftime`. Die Addition von `difftime` und `POSIXlt` oder `POSIXct` Objekten ist ebenfalls definiert.

```
> lDates + c(60, 120)                    # 1 und 2 Minuten später
[1] "1972-08-05 03:38:00 CET" "1981-03-31 12:46:00 CEST"

> (diff21 <- lDates[2] - lDates[1])
Time difference of 3160.338 days

> lDates[1] + diff21
[1] "1981-03-31 12:44:00 CEST"
```

[50] Die Zeiteinheit der im `difftime` Objekt gespeicherten Werte (etwa Tage oder Minuten), hängt davon ab, aus welchen Datumsangaben das Objekt entstanden ist. Alternativ bestimmt das Argument `units` von `difftime()`, um welche Einheit es sich handeln soll.

In der `seq()` Funktion (vgl. Abschn. 2.4.1) ändert sich die Bedeutung des Arguments `by` hin zu Zeitangaben, wenn für `from` oder `to` Datumsangaben übergeben werden. Für die Schrittweite werden dann etwa die Werte "⟨Anzahl⟩ `years`" oder "⟨Anzahl⟩ `days`" akzeptiert (vgl. `?seq.POSIXt`). Dies gilt analog auch für das Argument `breaks` der `cut()` Funktion (vgl. Abschn. 2.6.7), die kontinuierliche Daten in Kategorien einteilt, die etwa durch Stunden (`breaks="hour"`) oder Kalenderwochen (`breaks="week"`) definiert sind (vgl. `?cut.POSIXt`). Für weitere geeignete arithmetische Funktionen vgl. `methods(class="POSIXt")` und `methods(class="Date")`.

```
# jährliche Schritte vom 01.05.2010 bis zum 01.05.2013
> seq(ISOdate(2010, 5, 1), ISOdate(2015, 5, 1), by="years")
[1] "2010-05-01 12:00:00 GMT" "2011-05-01 12:00:00 GMT"
[3] "2012-05-01 12:00:00 GMT" "2013-05-01 12:00:00 GMT"

# 4 zweiwöchentliche Schritte vom 22.10.1997
> seq(ISOdate(1997, 10, 22), by="2 weeks", length.out=4)
[1] "1997-10-22 12:00:00 GMT" "1997-11-05 12:00:00 GMT"
[3] "1997-11-19 12:00:00 GMT" "1997-12-03 12:00:00 GMT"

# 100 zufällige Daten zwischen 13.06.1995 und 4 Wochen später
> secsPerDay   <- 60 * 60 * 24                 # Sekunden pro Tag
> randDates    <- ISOdate(1995, 6, 13)
+               + sample(0:(28*secsPerDay), 100, replace=TRUE)

# teile Daten in Kalenderwochen ein
> randWeeks <- cut(randDates, breaks="week")
> summary(randWeeks)                          # Häufigkeiten
1995-06-12     1995-06-19     1995-06-26     1995-07-03     1995-07-10
        15             26             20             37              2
```