Die Grundlagen der R-Programmiersprache

Jonathan Harrington

1. Vektoren
1.1 Elemente
Es gibt verschiedene Sorten von Objekten in R, in denen Daten gespeichert werden können. In R können Objekte beliebige Namen haben, ausgenommen dass:

- sie dürfen nicht mit einer Zahl anfangen. Daher ist ein Objektname wie `12x` verboten
- es dürfen keine Leerstellen in dem Objektnamen vorkommen. `x y` wäre als Objektname auch verboten

Das einfachste Objekt ist ein *Vektor*, der aus mehreren *Elementen* besteht. Ein Vektor wird auf diese Weise erzeugt:

```
> x = 3
```

Hier ist `x` der Objektname des Vektors und = ist ein Operator: dieser = Operator bedeutet "wird". Der Befehl bedeutet daher buchstäblich: "wir erzeugen einen Vektor mit dem Objektnamen `x` und dieser Vektor enthält ein Element, die 3".

Wenn der Objektname alleine eingegeben wird, erscheinen alle Elemente des Vektors

```
> x
[1] 3
```

Um mehrere Elemente in einem Vektor zu speichern, muss die c() Funktion verwendet werden. Jede Funktion hat einen Namen und darauf folgende ( ) Klammern. Innerhalb der Klammern kann es mehrere *Argumente* geben, die mit Kommazeichen getrennt werden müssen. In der c() Funktion sind die Argumente die Elemente, die wir speichern möchten:

```
>  x = c(3, 4, 6, 89.3, 0, -10)
```

Dies bedeutet: wir speichern in dem Vektor x die Werte 3, 4, 6 usw. Wir können wieder durch die Eingabe des Namens des Vektors bestätigen, dass diese Werte tatsächlich gespeichert worden sind:

```
> x
[1]   3.0   4.0   6.0  89.3   0.0 -10.0
```

Diese Beispiele sind alle von *numerischen* Vektoren. Wir können aber auch *alphanumerische* Vektoren erzeugen, die *Schriftzeichen-Elemente* enthalten. In diesem Fall müssen alle Elemente mit " eingetragen werden:

```
> x = c("Kiel ", "Phonetik", "Gebaeude 10")
```

```
> x
[1] "Kiel "       "Phonetik"    "Gebaeude 10"
```

Wenn numerische und alphanumerische Daten in demselben Vektor gespeichert werden sollen, dann werden alle Elemente als alphanumerisch gespeichert und erscheinen daher in "

```
> x
[1] "Kiel"       "Phonetik"    "Gebaeude 10" "20"          "30"
[6] "1"
```

Man muss darauf achten, dass Vektoren bei der Anwendung vom = Operator immer überschrieben werden:

```
> y = c("a", "d", "e")
> y
[1] "a" "d" "e"
> y = c(10, 20)
> y
[1] 10 20
```

1.2 Zugriff auf Elemente

Nehmen wir an, dass wir den obigen Vektor erzeugt haben, und wir möchten jetzt auf das zweite Element “Phonetik” zugreifen. Das wird auf diese Weise gemacht:

```
> x[2]
[1] "Phonetik"
```

R ist eine ‘unity-indexed’ Sprache, weil wir auf das erste Element mit dem Index 1 zugreifen (viele Computer-Sprachen sind ‘zero-indexed’ und fangen mit dem Index 0 an)

```
> x[1]
[1] "Kiel"
```

Wir können mit [m:n] auf Elemente *m* bis *n* zugreifen:

```
> x[3:5]
[1] "Gebaeude 10" "20"          "30"
```

oder in der anderen Reihenfolge:

```
> x[5:3]
[1] "30"          "20"          "Gebaeude 10"
```

Alle diese Eingaben können immer mit dem = Operator in einem anderen Vektor gespeichert werden:

```
> y = c(10, 20, 30, 40)
> z = y[2:4]
> z
[1] 20 30 40
```

```
> neudat = z = y[2:4]
> neudat
[1] 20 30 40
```

Wenn wir auf verschiedene Elemente zugreifen möchten, müssen wir die oben-erwähnte c() Funktion benutzen. So würden wir auf Elemente 2 und 4 zugreifen:

```
> x[c(2,4)]
[1] "Phonetik" "20"
```

Wenn einem Vektore mehrere Elemente entnommen werden sollen, ist es sinnvoll die Indexziffer in einem anderen Vektor zu speichern. Wir kommen auf diese Weise zum selben Ergebnis:

```
> a = c(2, 4)
> x[a]
[1] "Phonetik" "20"
```

Das Minuszeichen vor einem Index bedeutet: *nicht* dieses Element.

```
> x[-3]
[1] "Kiel"     "Phonetik" "20"       "30"       "1"
```

Das Minuszeichen kann mit der c() Funktion verwendet werden:

```
> x[-c(2, 4)]
[1] "Kiel"        "Gebaeude 10" "30"          "1"
```
bedeutet alle Elemente außer dem zweiten und vierten.

```
> x[-c(2:4)]
[1] "Kiel" "30"   "1"
```
bedeutet alle Elemente außer Elementen 2 bis 4.

Es ist oft nützlich zu wissen, *wie viele Elemente* ein Vektor enthält. Dazu wird die `length()` Funktion verwendet:

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

Wenn ein Index eingeben wird, der größer als die Vektorlänge ist, gibt es eine Fehlmeldung:

```
> x[20]
[1] "NA"
```

Hier bedeutet “NA” not applicable (nicht zutreffend).

In R können auf folgende Weise Elemente eines vorhandenen Vektors überschrieben werden. Um Elemente 3 bis 5 des Vektors `x` mit den neuen Elementen "Hamburg", "London", "Paris" zu überschreiben:

```
> x
[1] "Kiel"       "Phonetik"    "Gebaeude 10" "20"          "30"
[6] "1"
> x[3:5] = c("Hamburg", "London", "Kiel")
> x
[1] "Kiel"     "Phonetik" "Hamburg"  "London"   "Kiel"     "1"
```

1.3 Arithmetische Operator (Vektorenrechnung)

In R werden `+ - * /` für Addition, Subtraktion, Multiplikation und Division verwendet. Diese Operatoren werden auf alle Elemente eines Vektors angewendet:

```
> x = c(10, 20, 30)
> x  + 5
[1] 15 25 35
```

Es ist sehr wichtig zu bemerken, dass diese Operatoren immer auf *parallele Elemente* von zwei Vektoren angewendet werden:

```
> veca = c(10, 2, -5, 16)
> vecb = c(0, 4.5, 18, -0.35)
> veca + vecb
[1] 10.00  6.50 13.00 15.65
```

Hier werden also `veca[m]` und `vecb[m]` (`m` ist eine Zahl zwischen 1 und 4) addiert.

Daher soll man immer aufpassen, dass die Vektoren von gleicher Länge sind (siehe `length()` oben), sonst gibt es eine Warnmeldung (und das Ergebnis hat auch wenig Sinn). Zum Beispiel:

```
> vecneu = c(10, 20, 30)
> veca+vecneu
[1] 20 22 25 26
Warning message:
longer object length
        is not a multiple of shorter object length in: veca + vecneu
```

Man kann prüfen, ob die Vektoren von derselben Länge sind (diese Eingaben verwenden einen logischen Operator `==`, der ausführlicher in 3. beschrieben wird).

```
> length(veca) == length(vecb)
[1] TRUE
> length(veca) == length(vecneu)
[1] FALSE
```

Es gibt mehrere nützliche arithmetische Funktionen für die Analyse von Sprechdaten. Hier sind einige davon.

- Die Elemente eines (numerischen) Vektors hoch *n*

```
> veca^2
[1] 100   4  25 256
```

- Die Quadratwurzel der Elemente.

```
> x = c(2, 4, 6)
> sqrt(x)
[1] 1.414214 2.000000 2.449490
```

oder

```
> x^0.5
[1] 1.414214 2.000000 2.449490
```

- Die `log()` Funktion berechnet den Logarithmus von x.

```
> log(x)
[1] 0.6931472 1.3862944 1.7917595
```

bedeutet $\log_e 2$ (oder ln 2), ln 4, ln 6.

```
> log(x, base=10)
[1] 0.3010300 0.6020600 0.7781513
```

bedeutet $\log_{10}2$, $\log_{10}4$, $\log_{10}6$

- Die Funktionen `max()`, `median()`, `mean()`, `min()`, `sum()` berechnen den Höchswert, Zentralwert, Mittelwert, Minimum, und Summe eines Vektors und haben daher jeweils nur *eine Zahl* als Ergebnis. Zum Beispiel:

```
> x
[1] 2 4 6
> sum(x)
[1] 12
> mean(c(10, 20, 30, 40))
[1] 25
```

- Die Funktion `range()` berechnet den Bereich und gibt daher zwei Werte zurück: das Minimum und den Höchstwert:

```
> vec = c(0, -5, 100, 35)
> range(vec)
[1]  -5 100
```

- Die `round()` Funktion rundet die Elemente in einem Vektor auf beliebig viele Nachkommastellen. Zum Beispiel:

```
> vec = sqrt(c(5, 11, 13, 19))
> vec
[1] 2.236068 3.316625 3.605551 4.358899
> round(vec, 2)
[1] 2.24 3.32 3.61 4.36
```

1.4 Weitere nützliche Funktionen

(Diese Funktionen sind nützlich, nachdem man etwas mehr Erfahrung mit der R Programmiersprache gewonnen hat. Nicht alle Einzelheiten werden hier diskutiert – verwenden Sie dazu die `help()` Funktion – zB help(rep) usw.).

- Der : Operator ist bereits in 1.1 erwähnt worden – mit `m:n` wird eine Reihe von Integer-Zahlen (ganzen Zahlen) zwischen *m* und *n* erzeugt:

```
> 10:20
 [1] 10 11 12 13 14 15 16 17 18 19 20
> -20:11
 [1] -20 -19 -18 -17 -16 -15 -14 -13 -12 -11 -10  -9  -8  -7  -6  -5  -4  -3  -2
[20]  -1   0   1   2   3   4   5   6   7   8   9  10  11
> 5:0
[1] 5 4 3 2 1 0
```

- Die `seq()` Funktion erzeugt numerische Intervalle zwischen zwei Zahlen. Sie wird meistens mit den Argumenten `by=`, `to=`, oder `length=` verwendet:

```
> seq(10, 20, by=2)
[1] 10 12 14 16 18 20

> seq(10, 20, length=15)
 [1] 10.00000 10.71429 11.42857 12.14286 12.85714 13.57143 14.28571 15.00000
 [9] 15.71429 16.42857 17.14286 17.85714 18.57143 19.28571 20.00000
```

(15 Werte von gleichem Abstand zwischen 10 und 20)

```
> seq(10, by=5, to=100)
 [1]  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
```

(von 10 bis 100 mit einem Abstand von 5).

- Die `rep()` Funktion ist für die Wiederholung von Elementen nützlich. Das erste Argument ist ein Vektor; das zweite ist eine Integerzahl, mit dem die Elemente wiederholt werden sollten:

```
> y = c("a", "b", "c", "d")
> rep(y, 3)
 [1] "a" "b" "c" "d" "a" "b" "c" "d" "a" "b" "c" "d"

rep(1:5, 4)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

Das zweite Argument kann eine Integerzahl derselben Länge des Vektors sein; damit kann man die Elemente auf unterschiedliche Weisen wiederholen:

```
> y = c("a", "b", "c", "d")
> rep(y, c(2, 4, 1, 10))
[1] "a" "a" "b" "b" "b" "b" "c" "d" "d" "d" "d" "d" "d" "d" "d" "d" "d"
```

oder

```
> n = c(2, 4, 1, 10)
> rep(y, n)
 [1] "a" "a" "b" "b" "b" "b" "c" "d" "d" "d" "d" "d" "d" "d" "d" "d" "d"
```

- Die `unique()` Funktion gibt alle im Vektor vorkommenden Elemente ohne Wiederholungen aus (nicht die einzelnen Tokens sondern nur die Typen):

```
> y = c("i", "i", "a", "a", "E", "E", "E", "E", "U")
> unique(y)
[1] "i" "a" "E" "U"
```

- Die `table()` Funktion tabelliert die Anzahl der verschiedenen Typen eines Vektors:

```
> table(y)
a E i U
2 4 2 1
```

- Die `sort()` Funktion sortiert entweder alphabetisch oder numerisch:

```
> sort(y)
> sort(y)
[1] "a" "a" "E" "E" "E" "E" "i" "i" "U"
> sort(c(10, -5, 1.5, 100))
[1]  -5.0   1.5  10.0 100.0
```

Die `paste()` Funktion fügt jedem Element eines alphanumerischen Vektor beliebige Schriftzeichen hinzu. Hier ist ein Beispiel:

```
> y
[1] "i" "i" "a" "a" "E" "E" "E" "E" "U"
> paste(y, "Silbenfinal")
[1] "i Silbenfinal" "i Silbenfinal" "a Silbenfinal" "a Silbenfinal"
[5] "E Silbenfinal" "E Silbenfinal" "E Silbenfinal" "E Silbenfinal"
[9] "U Silbenfinal"
```

Man kann die Leerstelle dazwischen mit einem anderen beliebigen Schriftzeichen durch das dritte Argument `sep=` ersetzen:

```
> paste(y, "Silbenfinal", sep=".")
[1] "i.Silbenfinal" "i.Silbenfinal" "a.Silbenfinal" "a.Silbenfinal"
[5] "E.Silbenfinal" "E.Silbenfinal" "E.Silbenfinal" "E.Silbenfinal"
[9] "U.Silbenfinal"
```

Daher, wenn man keine Leerstelle haben will:

```
> paste(y, "Silbenfinal", sep="")
[1] "iSilbenfinal" "iSilbenfinal" "aSilbenfinal" "aSilbenfinal" "ESilbenfinal"
[6] "ESilbenfinal" "ESilbenfinal" "ESilbenfinal" "USilbenfinal"
```

`paste()` kann auch verwendet werden, um die Elemente von zwei Vektoren auf eine ähnliche Weise zu verbinden.

```
> silbe = c("S", "W", "W", "S", "S", "S", "W", "W", "W")
> paste(y, silbe, sep=".")
[1] "i.S" "i.W" "a.W" "a.S" "E.S" "E.S" "E.W" "E.W" "U.W"
```

2. Matrizen

2.1 Erzeugung von Matrizen

Wenn wir VOT-Messungen von verschiedenen Plosiven analysieren wollen, dann handelt es sich um einen einzigen Parameter, die Dauerwerte, die mit einem Vektor gespeichert werden können. In der gesprochenen Sprache müssen wir uns jedoch oft mit sämtlichen akustischen Parametern befassen. Ein Beispiel wäre die F1 und F2-Werte zum zeitlichen Mittelpunkt von Vokalen. Solche zwei-dimensionalen Daten werden in R in einer *Matrix* gespeichert. (Es können auch sämtliche Parameter sein: die Formantwerte, die F0-Werte, die Dauer usw.).

In R kann eine Matrix aus Vektoren mit den `rbind()` und `cbind()` Funktionen zusammengesetzt werden. Mit `rbind()` werden die Vektoren als *Reihen* der Matrix gespeichert, und mit `cbind()` als *Spalten*:

```
> a = c(10, 3, 8, 7)
> b = c(11, 45, 20, -1)
> x = rbind(a, b)
> x
     [,1] [,2] [,3] [,4]
[1,]   10    3    8    7
[2,]   11   45   20   -1
```

Hier besteht die erste Reihe von `x` aus dem Vektor `a` und die zweite Reihe aus demVektor `b`. Und hier:

```
> y = cbind(a, b)
> y
[1,] 10 11
[2,]  3 45
[3,]  8 20
[4,]  7 -1
```

bestehen die ersten zwei *Spalten* von `y` aus `a` und `b`. Die Anzahl der Reihen und der Spalten können mit den Funktionen `nrow()` und `ncol()` bestätigt werden:

```
> nrow(x)
[1] 2
> ncol(x)
[1] 4
```

`x` ist daher eine 2 x 4 Matrix, da sie 2 Reihen und 4 Spalten hat. Die Anzahl der Reihen und Spalten kann auch mit der `dim()` Funktion bestätigt werden:

```
> dim(x)
[1] 2 4
```

2.2. Arithmetische Funktionen

Die in 1.3 erwähnten Funktionen können auch auf Matrizen angewandt werden. Die +, -, *, / Funktionen werden wie bei Vektoren auf die Elemente angewendet:

```
> x
  [,1] [,2] [,3] [,4]
    10    3    8    7
    11   45   20   -1
> x -20
  [,1] [,2] [,3] [,4]
   -10  -17  -12  -13
    -9   25    0  -21
```

Wenn arithmetische Funktionen auf zwei Matrizen angewandt werden, dann werden immer Elemente derselben Reihe und Spalte addiert, subtrahiert, dividiert, oder multipliziert:

```
> x
  [,1] [,2] [,3] [,4]
    10    3    8    7
    11   45   20   -1
> y = x*2
> y
  [,1] [,2] [,3] [,4]
    20    6   16   14
    22   90   40   -2

> x + y
  [,1] [,2] [,3] [,4]
    30    9   24   21
    33  135   60   -3
```

45
+
90
=
135

Daher sollte man immer zuerst bestätigen, dass die beiden Matrizen dieselbe Reihen- und Spaltenanzahl haben, denn sonst gibt es eine Fehlmeldung:

```
> y
  [,1] [,2] [,3] [,4]
    10    3    8    7
    10    3    8    7
    10    3    8    7
> x
  [,1] [,2] [,3] [,4]
    10    3    8    7
    11   45   20   -1
> y/x
Error in y/x : non-conformable arrays
```

Die in 1.3 erwähnten Funktionen wie `mean()`, `median()`, `sum()` usw. werden auf alle Elemente einer Matrix angewendet:

```
> x
  [,1] [,2] [,3] [,4]
    10    3    8    7
    11   45   20   -1
> mean(x)
[1] 12.875
```

Hier ist `mean(x)` der Durchschnitt von allen Elementen 10, 3, 8, 7, 11, 45, 20, -1.
Wenn man solche Funktionen auf die Reihen oder die Spalten einer Matrix anwenden möchte, muss die `apply()` Funktion verwendet werden. Hier gibt es drei Argumente:

- die Matrix
- eine 1 (Reihen) oder 2 (Spalten)
- die Funktion, die angewandt werden soll: `mean()` oder `median()` oder `var()` usw.

Zum Beispiel:

```
> apply(x, 1, mean)
[1]  7.00 18.75
```

Hier wird die Funktion `mean()` auf alle Reihen der Matrix x angewendet. 7.00 ist der Durchschnitt der Elemente in Reihe 1 von `x` und 18.75 ist der Durchschnitt der Elemente in Reihe 2.

```
> apply(x, 2, median)
[1] 10.5 24.0 14.0  3.0
```

Hier wird die Funktion median() auf alle Spalten in `x` angewendet. 10.5 ist der Zentralwert der Elemente in Spalte 1 von x usw.

2.3. Zugriff auf Elemente einer Matrix

Ein Element in Reihe *r* und Spalte *s* wird einer Matrix *m* durch m[r,s] entnommen. Daher:

```
> x
  [,1] [,2] [,3] [,4]
    10    3    8    7
    11   45   20   -1

> x[2,4]
[1] -1
```

m[r,] greift auf alle Elemente der Reihe *r* zu, und m[,s] auf alle Elemente der Spalte *s*:

```
> x[2,]
[1] 11 45 20 -1
```

Reihe 2 von x

```
> x[,3]
[1] 8 20
```

Spalte 3 von x

Die selben Vorgänge, die auf den Zugriff von Elementen in Vektoren angewandt worden sind, können ebenfalls für den Zugriff von Elementen in Matrizenreihen und –spalten verwendet werden. Es ist immer wichtig zu bemerken:

Eingaben vor dem Komma beziehen sich auf die Reihen, Eingaben nach dem Komma auf die Spalten

Wir erzeugen zuerst eine 3 x 4 Matrix `neumat` aus den Vektoren `x, y, z`:

```
> x = 1:4
> y = c(-10, 0, 20, 30)
> z = c(1.1, 1.2, 1.3, 1.4)
> neumat = rbind(x, y, z)
> neumat
      [,1] [,2] [,3] [,4]
[1,]   1.0  2.0  3.0  4.0
[2,] -10.0  0.0 20.0 30.0
[3,]   1.1  1.2  1.3  1.4
```

```
> neumat[2:3,]
      [,1] [,2] [,3] [,4]
[1,] -10.0  0.0 20.0 30.0
[2,]   1.1  1.2  1.3  1.4
```

Reihen 2 und 3

```
> neumat[,2:4]
     [,1] [,2] [,3]
[1,]  2.0  3.0  4.0
[2,]  0.0 20.0 30.0
[3,]  1.2  1.3  1.4
```

Spalten 2 bis 4

```
> neumat[2:3,3:4]
     [,1] [,2]
[1,] 20.0 30.0
[2,]  1.3  1.4
```

Reihen 2 und 3 von Spalten 3 bis 4

```
> neumat[,c(2,4)]
     [,1] [,2]
[1,]  2.0  4.0
[2,]  0.0 30.0
[3,]  1.2  1.4
```

Spalten 2 und 4

```
> neumat[1,c(2,4)]
[1] 2 4
```

Spalte 1 von Reihen 2 und 4

```
> neumat[-2,1:3]
     [,1] [,2] [,3]
[1,]  1.0  2.0  3.0
[2,]  1.1  1.2  1.3
```

Spalten 1 bis 3 von allen Reihen außer Reihe 2

```
> neumat[2:3,-c(2,4)]
      [,1] [,2]
[1,] -10.0 20.0
[2,]   1.1  1.3
```

Reihen 2 und 3 von allen Spalten außer Spalten 2 und 4

Wie bei Vektoren können Elemente einer Matrix ganz einfach mit dem = Operator überschrieben werden. Zum Beispiel so könnte man Spalten 2 und 3 der ersten Reihe mit 100 und 200 ersetzen:

```
> neumat[1,c(2:3)] = c(100, 200)
> neumat
      [,1]  [,2]  [,3] [,4]
[1,]   1.0 100.0 200.0  4.0
[2,] -10.0   0.0  20.0 30.0
```

```
[3,]   1.1   1.2   1.3  1.4
```

2.4 Weitere nützliche Funktionen

Die `c()` Funktion wandelt eine Matrix wieder in einen Vektor um. In der Umwandlung werden die Elemente *der Spalte nach* verkettet:

```
> vec = c(neumat)
> vec

[1]   1.0 -10.0   1.1 100.0   0.0   1.2 200.0  20.0   1.3   4.0  30.0   1.4
```

Die `matrix()` Funktion wird verwendet, um ein Vektor in eine Matrix umzuwandeln. Die ersten drei Argumente dieser Funktion sind:

- Der Name vom Vektor
- Die Anzahl der Reihen
- Die Anzahl der Spalten

Die Elemente eines Vektors werden der Spalte nach in eine Matrix aufgebaut. Daher können wir die ursprüngliche Matrix `neumat` aus `vec` mit diesem Befehl rekonstruieren:

```
> matrix(vec, 3, 4)
```

Wenn eine Zahl anstatt eines Vektors für das erste Argument eingegeben wird, dann ist das Ergebnis eine Matrix die aus dieser Zahl besteht:

```
> matrix(5, 3, 3)
     [,1] [,2] [,3]
[1,]    5    5    5
[2,]    5    5    5
[3,]    5    5    5
```

Die `t()` Funktion (‘transpose’) transponiert die Reihen und Spalten:

```
> t(neumat)
     [,1] [,2] [,3]
[1,]    1  -10  1.1
[2,]  100    0  1.2
[3,]  200   20  1.3
[4,]    4   30  1.4
```

Die `dimnames()` Funktion wird verwendet, um den Dimensionen einer Matrix Namen zu geben. In der Analyse von Sprechdaten entsprechen die Reihen oft Etikettierungen und die Spalten den verschiedenen Parametern. Zum Beispiel könnte neumat Werte für drei Vokale /i e a/ von vier Parametern A, B, C, D enthalten. So könnte man mit dimnames() diese Informationen der Matrix hinzufügen (die Dimensionen-Namen müssen immer aus einem Vektor von Schriftzeichen bestehen)

```
> vowlab = c("i", "a", "u")
> param = c("A", "B", "C", "D")
> dimnames(neumat) = list(vowlab, param)
> neumat
```

```
      A     B     C    D
i   1.0 100.0 200.0  4.0
a -10.0   0.0  20.0 30.0
u   1.1   1.2   1.3  1.4
```

Zwar sieht das aus als ob die Matrix jetzt eine zusätzliche Reihe und Spalte bekommen hätte – in Wirklichkeit ändert sich nichts an der Anzahl der Reihen und Spalten:

```
> dim(neumat)
[1] 3 4
```

Diese Eingaben zeigen die Dimensionen-Namen der Reihen und der Spalten:

```
> dimnames(neumat)[[1]]
[1] "i" "a" "u"
> dimnames(neumat)[[2]]
[1] "A" "B" "C" "D"
```

Die Dimensionen-Namen können auf diese Weise gelöscht werden:

```
> dimnames(neumat) = list(NULL, NULL)
>neumat
> neumat
      [,1]  [,2]  [,3] [,4]
[1,]   1.0 100.0 200.0  4.0
[2,] -10.0   0.0  20.0 30.0
[3,]   1.1   1.2   1.3  1.4
```

3. Logische Vektoren

3.1 Einführung

Nehmen wir an, wir haben ein Objekt von Vokaletikettierungen und ein zweites Objekt, das deren Dauer- und F1-Werte enthält. Diese Objekte heißen `labs` und `daten`: `labs` ist ein Schriftzeichen-Vektor (der Vokal-Etikettierungen) und `daten` ist eine Matrix der Dauerwerte (in der ersten Spalte) und der F1-Werte (in der zweiten Spalte).

```
> labs
 [1] "a" "e" "i" "e" "i" "a" "e" "a" "a" "i"

> daten
      Dauer  F1
 [1,]    33 979
 [2,]    56 592
 [3,]    37 224
 [4,]    50 597
 [5,]    49 281
 [6,]    21 737
 [7,]    38 520
 [8,]    32 887
```

```
 [9,]    21 755
[10,]    60 343
```

`labs` und `daten` sind zueinander parallel: dies bedeutet, dass die Werte für Element *n* in `labs` in Reihe *n* von `daten` ist (daher hat der dritte Vokal eine Dauer von 37 ms und einen F1-Wert von 224 Hz, usw.).

Die Etikettierungen sind – wie wir oben sehen können – von verschiedenen Vokalen und jetzt möchten wir die Durchschnittsdauer der /a/ Vokale berechnen. Man könnte die entsprechenden Reihen der Matrix `daten` entnehmen, und dann die `mean()` Funktion anwenden:

```
> neudat = daten[c(1, 6, 8, 9),]
> mean(neudat[,1])
[1] 26.75
```

Dieser Vorgang ist nur möglich, wenn die Anzahl der Reihen klein ist, wodurch die Matrix überschaubar ist. Es wäre jedoch eine sehr langwierige Arbeit, in einer Matrix von über 2000 Vokalen alle Reihen auf diese Weise zu finden. Stattdessen verwendet man *logische Vektoren*, um auf Elemente in einem Objekt (wie `daten`) zu zugreifen, wenn gewisse Bedingungen in einem anderen Objekt (wie `labs`) erfüllt sind.

3.2 Was ist ein logischer Vektor?

Ein logischer Vektor besteht aus TRUE und FALSE Elementen und kann auf die übliche Weise mit dem = Operator erzeugt werden. Es genügt T und F einzugeben:

```
> temp = c(T, F, T)
> temp
> temp
[1]  TRUE FALSE  TRUE
```

Logische Vektoren folgen einer sogenannten *Boolean-Logik* mit diesen Prinzipien:

T und T gleicht T	F und F gleicht F
T und F gleicht F	
T oder T gleicht T	F oder F gleicht F
T oder F gleicht T	

Solche Befehle werden auf diese Weise in R eingetragen:

```
> T & F
[1] FALSE
> T | F
[1] TRUE
> F & F
[1] FALSE
```

& bedeutet 'und'
| bedeutet 'oder'

Klammern können auch verwendet werden: das Material innerhalb der Klammern wird zuerst berechnet:

```
> (T & F) | T
[1] TRUE
```

Die sum() Funktion summiert die Anzahl der T Elemente

```
> vec = c(T, T, F, T, F)
> sum(vec)
[1] 3
```

Man kann die Anzahl der F Elemente berechnen, indem ein ! Zeichen verwendet wird. Das ! Zeichen bedeutet 'nicht' oder das Entgegengesetzte:

```
> sum(!vec)
[1] 2
```

Die `any()` Funktion berechnet, ob mindestens ein T Element vorhanden ist:

```
> any(vec)
[1] TRUE
```

Mit any(!vec) stellt man die Frage: gibt es mindestens ein F Element?

```
> any(!vec)
[1] TRUE
```

3.3 Wie entstehen logische Vektoren?

Logische Vektoren entstehen durch die folgenden Operatoren in Tabelle I.

`x == y`	x gleicht y
x != y	x gleicht nicht y
x < y	x ist weniger als y
x <= y	x ist weniger als oder gleicht y
x > y	x ist grösser als y
x >= y	x ist grösser als oder gleicht y

Tabelle I: Logische Operatoren

In diesem Fall:

- sind `x` und `y` beide Vektoren
- besteht `x` aus einem oder mehreren Elementen

Wir müssen jetzt zwei Fälle berücksichtigen: erstens wenn `y` aus einem Element besteht; und zweitens wenn `y` aus derselben Anzahl von Elementen wie `x` besteht.

3.3.1 Vergleiche mit einem Element

Zuerst erzeugen wir einen Vektor:

> x = c(10, 20, 30)

Der Befehl `x == 20` bedeutet buchstäblich: welche Elemente von `x` gleichen 20? Ein T wird für jedes Element erzeugt, das 20 gleicht, sonst ein F:

```
> x == 20
[1] FALSE  TRUE FALSE
```

`x` kann auch alphanumerisch sein:

```
> x = c("Kiel", "Phonetik", 2002, "Signale")
> x == "Phonetik"
[1] FALSE  TRUE FALSE FALSE
```

Hier sind drei weitere Beispiele:

```
> x != "Phonetik"
[1]  TRUE FALSE  TRUE  TRUE
```

```
> daten
     Dauer  F1
 [1,]    33 979
 [2,]    56 592
 [3,]    37 224
 [4,]    50 597
 [5,]    49 281
 [6,]    21 737
 [7,]    38 520
 [8,]    32 887
 [9,]    21 755
[10,]    60 343
```

Dies bedeutet: welche Elemente der ersten Spalte in `daten` sind höher als 30?

```
> daten[,1] > 30
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE
```

Welche Elemente der Reihen 2 bis 8 der zweiten Spalte sind größer als, oder gleichen, 520?

```
> daten[2:8,2] <= 520
[1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

3.3.2 Parallele Vergleiche

`x` und `y` die gleiche Anzahl von Elementen. Die logischen Operators vergleichen in diesem Fall Element *m* von `x` mit Element *m* von `y`. Hier ist ein Beispiel:

```
> x = c(10, 20, 30)
> y = c(9, 50, 30)
> x == y
[1] FALSE FALSE  TRUE
```

Das Ergebnis ist F F T, weil nur die dritten Elemente von `x` und `y` identisch sind. Man kann mit den anderen logischen Operatoren ähnliche Vergleiche durchführen:

```
> x != y
[1]  TRUE  TRUE FALSE
```

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

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

3.3.3 Der %in% Operator

Der %in% Operator wird benutzt, um einem Vektor mehrere Elemente zu entnehmen. Dies kann auch mit sämtlichen Teilfragen und dem | Operator durchgeführt werden. Hier erzeugen wir einen logischen Vektor, der T ist, wenn labs `"I"` oder `"E"` enthält:

```
> labs
 [1] "I" "E" "I" "E" "I" "O" "O" "O" "I" "E"
> lvec= (labs == "I") | (labs == "E")
> lvec
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

Man kann aber das gleiche Ergebnis mit dem `%in%` Operator erreichen:

Dies bedeutet: ist `"I"` oder `"E"` in `labs` enthalten? (T für ja, F für nein).

```
> labs %in% c("I", "E")
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

3.4 Zugriff auf Elemente

3.4.1 Elemente in Vektoren

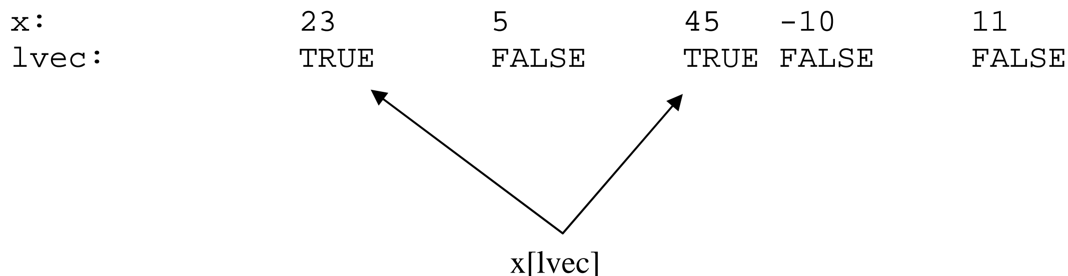
Mit logischen Vektoren kann man auf Elemente eines Vektors oder einer Matrix auf eine ähnliche Weise wie in 1.2 and 2.3 zugreifen. Wir erzeugen zuerst einen logischen Vektor `lvec`, der T ist, wenn ein Element in `x` größer als 20 ist:

```
> x = c(23, 5, 45, -10, 11)

> lvec = x > 20
[1]  TRUE FALSE  TRUE FALSE FALSE
```

Die Bedeutung von `x[lvec]` ist: alle Elemente von `x`, für die `lvec` TRUE ist.

```
x:          23      5       45   -10      11
lvec:       TRUE    FALSE   TRUE FALSE    FALSE
```



Daher besteht `x[lvec]` aus den Elementen 23 45. Die Bedeutung von `x[!lvec]` ist: alle Elemente in `x`, für die lvec FALSE ist:

```
> x[!lvec]
```

```
[1]   5 -10  11
```

In diesem Beispiel haben wir zwei Objekte:

- `freunde`: die Namen von fünf Freunden
- `zeit`: die Dauer, die diese Freunde brauchen, um in die Arbeit zu fahren

```
> freunde = c("Paul", "Karin", "Elke", "Georg", "Peter")
> zeit = c(50, 11, 35, 41, 12)
```

Mit diesen Objekten und logischen Vektoren können wir sämtliche Fragen beantworten wie:

- Welche Freunde brauchen mehr als 40 Minuten?

```
> temp = zeit > 40
```

Diese Zeile erzeugt einen logischen Vektor, der True ist, wenn ein Element von `zeit` größer als 40 ist

```
> freunde[temp]
[1] "Paul"  "Georg"
```

Diese Zeile bedeutet: die Elemente in `freunde`, für die `temp` T ist.

- Welche Freunde brauchen mehr als die Durchschnittszeit?

```
> temp = zeit > mean(zeit)
> freunde[temp]
[1] "Paul"  "Elke"  "Georg"
```

Welcher Freund braucht am längsten? Hier muss die `max()` Funktion aus 1.3 verwendet werden:

```
> temp = zeit == max(zeit)
> freunde[temp]
[1] "Paul"
```

- Wieviele Freunde brauchen weniger als 40 Minuten?

```
> temp = zeit < 40
> sum(temp)
[1] 3
```

Oder in einer Zeile:

```
> sum(zeit < 40)
[1] 3
```

- Gibt es Freunde, die mehr als 45 Minuten brauchen?

```
> temp = zeit > 45
> any(temp)
[1] TRUE
```

oder in einer Zeile:

```
> any(zeit > 45)
[1] TRUE
```

Eine Frage wie

‘welcher Freund braucht zwischen 25 und 45 Minuten?’

muss in zwei Fragen zerlegt werden, die dann mit dem & Operator verbunden werden. Die Teilfragen sind:

- die Freunde, die mehr als 25 Minuten brauchen

```
> temp = zeit > 25
```

und (&)

- die Freunde, die weniger als 45 Minuten brauchen

```
> temp = zeit < 45
```

Die Teilfragen müssen auch in ( ) Klammern gesetzt werden:

```
> temp = (zeit > 25) & (zeit < 45)
> freunde[temp]
[1] "Elke"  "Georg"
```

Hier ist etwas Komplizierteres. Wir wollen wissen, welche Freunde länger als Elke brauchen, um in die Arbeit zu kommen. Man muss immer versuchen, solche Fragen in kleinere Teilfragen zu zerlegen:

- Wir benötigen die Zeit, die Elke braucht, um in die Arbeit zu kommen (1)
- Wir müssen diese Zeit mit den anderen Zeiten vergleichen (2)
- Dafür müssen wir die entsprechenden Freunde finden (3)

Die Befehle sind:

```
> lvec = freunde=="Elke"
> zeit[lvec]
[1] 35
```

1. Zuerst brauchen wir einen logischen Vektor, der T ist, für die Elemente in `freunde`, die `Elke` sind. Damit können wir dann Elkes Zeit finden

```
>  temp = zeit > zeit[lvec]
```

2. Hier vergleichen wir alle Zeiten mit Elkes Zeit.

```
> freunde[temp]
[1] "Paul"  "Georg"
```

3. Und dann finden wir die entsprechenden Freunde.

3.4.2 Elemente in Matrizen

Mit logischen Vektoren kann man ebenfalls auf Reihen oder Spalten in Matrizen zugreifen. Die Grundregel ist wie in 2.3:

> Eingaben vor dem Komma beziehen sich auf Reihen.
> Eingaben nach dem Komma beziehen sich auf Spalten.

Hier sind wieder die Daten von Vokalen und deren Dauer- und Formantwerte:

```
> labs
 [1] "a" "e" "i" "e" "i" "a" "e" "a" "a" "i"

> daten
      Dauer  F1
 [1,]    33 979
 [2,]    56 592
 [3,]    37 224
 [4,]    50 597
 [5,]    49 281
 [6,]    21 737
 [7,]    38 520
 [8,]    32 887
 [9,]    21 755
[10,]    60 343
```

Eine neue Matrix der Dauer- und F1-Werte für die /a/ Vokale soll erzeugt werden. Zuerst muss /a/ mit einem logischen Vektor identifiziert werden:

```
> temp = labs=="a"
```

Die benötigte Matrix der /a/-Daten enthält dann die Reihen, für die `temp` TRUE ist:

```
> daten[temp,]
     Dauer  F1
[1,]    33 979
[2,]    21 737
[3,]    32 887
[4,]    21 755
```

Hier muss der logische Vektor **vor dem Komma** eingetragen werden, da wir alle **Reihen**, die Daten zum /a/ enthalten aus `daten` entnehmen möchten. Wenn wir nur die Dauerwerte von /a/ brauchen, dann muss eine 1 nach dem Komma eingetragen werden, um die erste Spalte zu identifizieren:

```
> daten[temp,1]
[1] 33 21 32 21
```

`daten[temp,2]` wäre die Eingabe für die F1-Werte von /a/, usw.

Grundlagen der R Programmiersprache
Fragen

Um diese Fragen zu beantworten, R starten und dann:
```
> load("/data/teach/Modul_F/R1uebung")
```

1. Die Matrix `flights` hat Informationen von der Anzahl der Abflüge von 5 Luftlinien (A-E) an 5 Reiseziele:

	Auckland	Hong Kong	London	Manilla	Tokyo
A	0	10	5	3	2
B	2	0	7	1	0
C	8	0	5	0	7
D	0	9	10	0	0
E	2	4	0	5	4

Beispiel.
Schreiben Sie R-Ausdrücke für:
- die Abflüge nach Hong Kong

Antwort: `flights[,2]`

- die Abflüge der Luftlinie C

Antwort: `flights[3,]`

Schreiben Sie R-Ausdrücke für:
(1) die Abflüge von A und E
(2) die Abflüge von B, D, und E nach London und Tokyo
(3) die Summe der Abflüge von A und D nach London und Manilla.
(4) das Produkt der Flüge nach Auckland und Hong Kong
(5) die Standardabweichung der Flüge nach Tokyo

Erzeugen Sie zwei Vektoren,
`luftlinien`, der die Namen der Luftlinien (A-E) enthält
`destinations`, mit den Namen der Reiseziele (Auckland, Hong Kong usw).

Beispiel
Verwenden Sie diese zwei Vektoren und logische Vektoren (wenn nötig), um diese Fragen zu beantworten:

Welche Luftlinien haben mehr als 5 Flüge nach London?

<u>Antwort</u>
```
temp = flights[,3] > 5
luftlinien[temp]
```

Verwenden Sie ggf. logische Vektoren, um diese Fragen zu beantworten.

7. Welche Luftlinien fliegen nach Manilla?
8. Welche Luflinien haben mehr Abflüge nach Hong Kong als nach Manilla?
9. Welche Luftlinien haben mehr Flüge nach London als nach Auckland und Tokyo zusammen?
10. Welche Zielorte haben weniger Flüge von der Luftlinie B als von der Luftlinie C?
11. Welche Luftlinien haben mehr Flüge als die Durchschnittszahl (von allen Fluglinien) nach Manilla?
12. Bei welchen Luftlinien gibt es einen Unterschied zwischen der Anzahl der Flüge nach Tokyo und nach Hong Kong?

B. Die Matrix `vdata.emu` enthält diese Daten, die in den Reihen für 812 Vokale gespeichert sind:

Dauer	Spalte 1
F1-Werte	Spalte 2
F2-Werte	Spalte 3
f0-Werte (Grundfrequenz)	Spalte 4

(die Formant- und Grundfrequenzwerte sind zum zeitlichen Mittelpunkt der Vokale entnommen worden).

Die Etikettierungen dieser 812 Vokale sind als der Vektor `vlabs.emu` gespeichert. Schreiben Sie R-Ausdrücke in den folgenden Fällen.

1. Der Durchschnitt der f0-Werte aller Vokale
2. F2 - F1 für die ersten 60 Vokale
3. Dividieren Sie die Summe der F1-Werte durch die Summe der f0-Werte
4. Gibt es Vokale, die einen F1-Wert von mehr als 900 Hz haben?
5. Wieviele Vokale gibt es mit einer Dauer von weniger als 100 ms?
6. Bei welchem Prozentteil der Vokale ist F2-F1 größer als 500 Hz?
7. Verwenden Sie logische Vektoren und die `sum()` Funktion, um festzustellen wieviele "E" Vokale es in `vlabs.emu` gibt.
8. Verwenden Sie logische Vektoren und die `any()` Funktion, um festzustellen ob es "U" Vokale in `vlabs.emu` gibt.
9. Verwenden Sie logische Vektoren und die `%in%` Funktion, um festzustellen, wieviele "E", "O" und "I" Vokale es in `vlabs.emu` gibt.
10. Was ist der F1-Durchschnitt aller "E" Vokale?
11. Was ist die Durchschnittsdauer aller "U" Vokale?
12. Bei welchem Prozentteil von "I" Vokalen ist F1 weniger als 350 Hz?