V03 – Subsetting

26. April 2021

Contents

1 Intro

Mit **Subsetting** (Teilmenge bilden) wird das Auswählen von Elementen aus Vektoren (oder darauf aufbauenden Datenstrukturen) bezeichnet. Zum Teil werden dafür auch die Begriffe Indizierung (*indexing*) oder *array slicing* benutzt.

R besitzt verschiedene, sehr mächtige Subsetting-Systeme, um Teile von atomaren Vektoren, Listen, Matrizen, Arrays oder Tibbles auszuwählen und zu verändern.

2 Auswahl mehrerer Elemente

2.1 Vektoren

Es gibt 6 Möglichkeiten, einen Vektor zu subsetten.

2.1.1 Positive ganze Zahlen

Der Rückgabewert besteht aus den Elementen an den angegebenen Positionen (Indizes). (Die erste Position ist 1, nicht 0.)

```
x <- c(2.1, 4.2, 3.3, 5.4, 1.5) # Die Nachkommastelle gibt die Position an
x[c(3, 1)]
## [1] 3.3 2.1
lst <- as.list(x)
str(lst[c(5L, 1L, 3L, 2L, 4L)]) # sortiere Liste um
## List of 5
##  $ : num 1.5
##  $ : num 2.1
##  $ : num 3.3
##  $ : num 4.2
##  $ : num 5.4
str(lst[2]) # auch Liste
## List of 1
##  $ : num 4.2
```

Mehrfaches Auftreten eines Index führt zu mehrfacher Rückgabe.

```
x[c(1, 1, 2, 1)]
## [1] 2.1 2.1 4.2 2.1
```

Bei reellen Zahlen als Indizes werden die Nachkommastellen abgeschnitten, da `double` automatisch in `integer` konvertiert wird.

```
str(lst[c(2.1, 2.9)])
## List of 2
##  $ : num 4.2
##  $ : num 4.2
```

2.1.2 Negative ganze Zahlen

Die Elemente an den angegebenen Indizes werden weggelassen.

```
x[-c(3, 1)]
## [1] 4.2 5.4 1.5
```

Negative und positive ganze Zahlen dürfen nicht gemischt werden.

```
# lst[c(-1, 2)] # ERROR
```

2.1.3 0, NULL

Ein Vektor der Länge 0 wird zurückgegeben.

```
x[0]
## numeric(0)
x[NULL]
## numeric(0)
x[integer(0)]
## numeric(0)
lst[0]
## list()
```

0 kann mit negativen oder mit positiven ganzen Zahlen gemischt werden (dann wird 0 ignoriert).

```r
x[0:2]
## [1] 2.1 4.2
x[0:-2]
## [1] 3.3 5.4 1.5
```

Beachte den Unterschied zwischen `1:length(x)` und `seq_along(x)`.

```r
f <- function(x) {for (i in 1:length(x)) cat("o"); cat("\n")}
g <- function(x) {for (i in seq_along(x)) cat("o"); cat("\n")}

x
## [1] 2.1 4.2 3.3 5.4 1.5
f(x)
## ooooo
g(x)
## ooooo

y <- x[x>9]
f(y)
## oo
g(y)

y
## numeric(0)
1:length(y)
## [1] 1 0
seq_along(y)
## integer(0)
```

2.1.4 Nichts

Der ursprüngliche Vektor wird zurückgegeben. Dies ist nicht besonders nützlich für Vektoren aber für Arrays und Tibbles (wie wir bald sehen werden).

```r
x[]
## [1] 2.1 4.2 3.3 5.4 1.5
str(lst[])
## List of 5
##  $ : num 2.1
##  $ : num 4.2
##  $ : num 3.3
##  $ : num 5.4
##  $ : num 1.5
```

2.1.5 Wahrheitswerte

Jene Elemente werden ausgewählt, an deren Position der Index-Vektor `TRUE` ist.

```r
str(lst[c(T, T, F, F, T)])
## List of 3
##  $ : num 2.1
##  $ : num 4.2
##  $ : num 1.5
x[x>3]
## [1] 4.2 3.3 5.4
```

Ist der Index-Vektor vom Typ `logical` kürzer als der indizierte Vektor, so wird der Index-Vektor so lange wiederholt, bis er die selbe Länge hat (**Recycling**).

```
x[c(T,F)] # gleich x[c(T,F,T,F,T)]
## [1] 2.1 3.3 1.5
x[c(F,T,F)] # gleich x[c(F,T,F,F,T)]
## [1] 4.2 1.5
```

2.1.6 Strings

Sind die Elemente eines Vektors benannt, kann ein `character`-Vektor aus diesen Namen genutzt werden, um entsprechende Elemente auszuwählen.

```
y <- structure(x, names=letters[1:4])
y # letzter Eintrag hat keinen Namen
##    a    b    c    d <NA>
##  2.1  4.2  3.3  5.4  1.5
y[c("d", "c", "a", "x")] # kein Element mit Namen "x"
##    d    c    a <NA>
##  5.4  3.3  2.1   NA
y[c("", "NA", "<NA>", NA_character_)] # unbenannte Elemente können nicht mit einem Namen gefunden werde
## <NA> <NA> <NA> <NA>
##   NA   NA   NA   NA
y[c("a", "a", "a", "b", "b")] # Mehrfachauswahl
##   a   a   a   b   b
## 2.1 2.1 2.1 4.2 4.2
```

2.2 Arrays

Es gibt 3 Arten, wie höher-dimensionale Strukturen indiziert werden können: durch einen einzelnen atomaren Vektor, mit mehreren atomaren Vektoren, mit einer Matrix.

2.2.1 Ein atomarer Vektor

Arrays (und damit Matrizen) bauen auf Vektoren auf und können genau wie Vektoren indiziert werden.

```
# erzeuge character-Matrix mit Indizes als Text:
matr <- outer(1:5, 1:4, FUN = "paste", sep = ",")
matr
##      [,1]  [,2]  [,3]  [,4]
## [1,] "1,1" "1,2" "1,3" "1,4"
## [2,] "2,1" "2,2" "2,3" "2,4"
## [3,] "3,1" "3,2" "3,3" "3,4"
## [4,] "4,1" "4,2" "4,3" "4,4"
## [5,] "5,1" "5,2" "5,3" "5,4"
vals <- matr
dim(vals) <- NULL # entferne Dimensionsattribut
vals
##  [1] "1,1" "2,1" "3,1" "4,1" "5,1" "1,2" "2,2" "3,2" "4,2" "5,2" "1,3" "2,3"
## [13] "3,3" "4,3" "5,3" "1,4" "2,4" "3,4" "4,4" "5,4"
matr[c(4, 15)] # gleich vals[c(4, 15)]
## [1] "4,1" "5,3"
matr[c(T,F)]
##  [1] "1,1" "3,1" "5,1" "2,2" "4,2" "1,3" "3,3" "5,3" "2,4" "4,4"
matr[c(-1:-10)]
##  [1] "1,3" "2,3" "3,3" "4,3" "5,3" "1,4" "2,4" "3,4" "4,4" "5,4"
```

Die Elemente von Arrays werden in der sogenannten *column-major order* gespeichert. Für höher-dimensionale Arrays bedeutet dies, dass bei Iteration über die Einträge die Dimensionen von links nach rechts durchgegangen werden.

```
# Gib alle 27 Indextupel eines 3x3x3 Arrays aus.
a <- array(1:27, rep(3,3))
index_matrix <- sapply(seq_along(dim(a)), function (i) slice.index(a,i))
index_strings <- apply(index_matrix, 1, function(x) paste0("[", paste(x, collapse=", "), "]"))
index_tbl <- tibble::tibble("Position als Vektor"=seq_along(a), Indextupel=index_strings)
print(index_tbl, n=27)
## # A tibble: 27 x 2
##    `Position als Vektor` Indextupel
##                    <int> <chr>
##  1                     1 [1, 1, 1]
##  2                     2 [2, 1, 1]
##  3                     3 [3, 1, 1]
##  4                     4 [1, 2, 1]
##  5                     5 [2, 2, 1]
##  6                     6 [3, 2, 1]
##  7                     7 [1, 3, 1]
##  8                     8 [2, 3, 1]
##  9                     9 [3, 3, 1]
## 10                    10 [1, 1, 2]
## 11                    11 [2, 1, 2]
## 12                    12 [3, 1, 2]
## 13                    13 [1, 2, 2]
## 14                    14 [2, 2, 2]
## 15                    15 [3, 2, 2]
## 16                    16 [1, 3, 2]
## 17                    17 [2, 3, 2]
## 18                    18 [3, 3, 2]
## 19                    19 [1, 1, 3]
## 20                    20 [2, 1, 3]
## 21                    21 [3, 1, 3]
## 22                    22 [1, 2, 3]
## 23                    23 [2, 2, 3]
## 24                    24 [3, 2, 3]
## 25                    25 [1, 3, 3]
## 26                    26 [2, 3, 3]
## 27                    27 [3, 3, 3]
```

2.2.2 Mehrere atomare Vektoren

Um die Dimensions-Eigenschaft ausnutzen, können wir durch `,` getrennt für jede Dimension einzeln einen Index-Vektor angeben.

Alle 6 Arten zur Indizierung von Vektoren sind erlaubt. Für die Nutzung von `character`-Subsetting muss das Attribut `dimnames` gesetzt sein.

Verschiedene Dimensionen können verschiedene Indizierungsarten nutzen.

Subsetting mit *nichts* kann hier sinnvoll eingesetzt werden, um ganze Zeilen / Spalten / Dimensionen auszuwählen.

```
m <- matrix(1:9, nrow = 3)
colnames(m) <- c("A", "B", "C")
m
```

```
##      A B C
## [1,] 1 4 7
## [2,] 2 5 8
## [3,] 3 6 9
m[1:2, ]
##      A B C
## [1,] 1 4 7
## [2,] 2 5 8
m[c(TRUE, FALSE, TRUE), c("B", "A")]
##      B A
## [1,] 4 1
## [2,] 6 3
m[0, -2]
##      A C
dim(m[0, -2])
## [1] 0 2
```

Standardmäßig wird das Resultat von `[` auf die kleinst-mögliche Anzahl an Dimensionen vereinfacht (Dimensionen mit Wert 1 werden weggelassen / "ge*drop*t"). Ggf wird das `dim`-Attribut vollständig entfernt. Dieses Verhalten wird mit dem optionalen Argument `drop=FALSE` verhindert.

```
m[1,]
## A B C
## 1 4 7
dim(m[1,]) # Vektor, keine Matrix
## NULL
m[1,,drop=FALSE] # 1x3 Matrix
##      A B C
## [1,] 1 4 7
dim(m[1,,drop=FALSE])
## [1] 1 3
a <- array(1:16, dim=(rep(2,4)))
dim(a[,1,1,]) # aus 4D-Array wird Matrix
## [1] 2 2
dim(a[,1,1,,drop=FALSE]) # bleibt 4D-Array
## [1] 2 1 1 2
```

Bemerkung: Faktoren haben auch ein `drop`-Argument. Jedoch ist die Bedeutung etwas anders: Es gibt an, ob Levels erhalten bleiben. Der Default ist `FALSE`.

```
z <- factor(c("a", "b", "a"))
z[c(1,3)] # zwei Levels (obwohl eines nicht vorkommt)
## [1] a a
## Levels: a b
z[c(1,3), drop = TRUE] # ein Level
## [1] a a
## Levels: a
```

2.2.3 Eine Matrix

Als dritte Möglichkeit können Arrays mit n Dimensionen durch `integer`- (oder ggf `character`)-Matrizen mit n Spalten indiziert werden. Jede Zeile der Matrix gibt ein Index-Tupel an; das entsprechende Element wird ausgewählt. Das Resultat ist ein Vektor (ohne Dimension).

```
matr
##      [,1]  [,2]  [,3]  [,4]
```

```
## [1,] "1,1" "1,2" "1,3" "1,4"
## [2,] "2,1" "2,2" "2,3" "2,4"
## [3,] "3,1" "3,2" "3,3" "3,4"
## [4,] "4,1" "4,2" "4,3" "4,4"
## [5,] "5,1" "5,2" "5,3" "5,4"
select <- rbind(
  c(1, 1),
  c(3, 1),
  c(2, 4)
)
select
##      [,1] [,2]
## [1,]    1    1
## [2,]    3    1
## [3,]    2    4
matr[select]
## [1] "1,1" "3,1" "2,4"
```

Die Funktion `arrayInd()` wandelt einen Vektor aus Vektor-Indizes in eine Matrix aus Array-Indizes.

```
arrayInd(1:6, c(2,3))
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    1
## [3,]    1    2
## [4,]    2    2
## [5,]    1    3
## [6,]    2    3
arrayInd(1:6, c(3,2))
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    1
## [3,]    3    1
## [4,]    1    2
## [5,]    2    2
## [6,]    3    2
```

Für ein `logical`-Objekt gibt `which()` die Indizes der `TRUE`-Einträge zurück. Mit `arr.ind=TRUE` ist die Rückgabe bei Arrays die Matrix aus Array-Indizes, ansonsten werden Vektor-Indizes zurückgegeben.

```
x <- matrix(1:9, nrow=3)
x
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
x %% 2 == 0
##       [,1]  [,2]  [,3]
## [1,] FALSE  TRUE FALSE
## [2,]  TRUE FALSE  TRUE
## [3,] FALSE  TRUE FALSE
which(x %% 2 == 0)
## [1] 2 4 6 8
which(x %% 2 == 0, arr.ind=T)
##      row col
```

```
## [1,]   2   1
## [2,]   1   2
## [3,]   3   2
## [4,]   2   3
```

2.3 Tibbles

Tibbles haben Eigenschaften von Listen und von Matrizen. Werden sie mit einem einzelnen atomaren Vektor indiziert, verhalten sie sich wie Listen. Beim Indizieren mit zwei Vektoren verhalten sie sich wie Matrizen.

```
library(tibble)
tb <- tibble(x = 1:3, y = 3:1, z = letters[1:3])
tb
## # A tibble: 3 x 3
##       x     y z
##   <int> <int> <chr>
## 1     1     3 a
## 2     2     2 b
## 3     3     1 c
tb[tb$x == 2, ]
## # A tibble: 1 x 3
##       x     y z
##   <int> <int> <chr>
## 1     2     2 b
tb[c(1, 3), ]
## # A tibble: 2 x 3
##       x     y z
##   <int> <int> <chr>
## 1     1     3 a
## 2     3     1 c
tb[c("x", "z")]
## # A tibble: 3 x 2
##       x z
##   <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
tb[, c("x", "z")]
## # A tibble: 3 x 2
##       x z
##   <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
```

Werden Tibbles mit `[` indiziert, erhält man immer ein Tibble. `drop = FALSE` ist bei Tibble der Default.

```
tb[["x"]]
## [1] 1 2 3
tb["x"]
## # A tibble: 3 x 1
##       x
##   <int>
## 1     1
## 2     2
```

```
## 3     3
tb[, "x"]
## # A tibble: 3 x 1
##       x
##   <int>
## 1     1
## 2     2
## 3     3
tb[, "x", drop=TRUE]
## [1] 1 2 3
```

3 Auswahl eines einzelnen Elementes

Neben `[` gibt es noch 2 weitere Subsetting-Operatoren: `[[` und `$`. Diese dienen der Auswahl eines einzelnen Elementes. Dabei entspricht `x$y` dem Aufruf von `x[["y"]]`.

3.1 [[

Der Subsetting-Operator `[[` hat für Listen größte Bedeutung. Mit `[` erhält man immer eine Liste, mit `[[` das Element.

```
x <- list(bla=1:3, blub="a", 4:6)
x[1]
## $bla
## [1] 1 2 3
x[[1]]
## [1] 1 2 3
```

Für das Indizieren mit `[[` werden eine einzelne natürliche Zahl oder ein String benutzt.

```
x[[2]]
## [1] "a"
x[["blub"]]
## [1] "a"
```

Wird ein atomarer Vektor der Länge > 1 übergeben, wird rekursiv indiziert, was typischerweise nur bei besonderen Datenstrukturen zum Einsatz kommt.

```
b <- list(x=1, a=list(y=2, b=list(z=3, c=list(d=5, w=4))))
str(b)
## List of 2
##  $ x: num 1
##  $ a:List of 2
##   ..$ y: num 2
##   ..$ b:List of 2
##   .. ..$ z: num 3
##   .. ..$ c:List of 2
##   .. .. ..$ d: num 5
##   .. .. ..$ w: num 4
# folgende Ausdrücke haben den selben Effekt:
b[["a"]][["b"]][["c"]][["d"]]
## [1] 5
b[[2]][[2]][[2]][[1]]
## [1] 5
b[[c("a", "b", "c", "d")]]
```

```
## [1] 5
b[[c(2, 2, 2, 1)]]
## [1] 5
```

Auch einzelne Elemente atomarer Vektoren können durch `[[` ausgewählt werden. Dann unterscheidet sich das Resultat nicht von `[`.

```
x <- 11:15
x[[2]]
## [1] 12
x[[c(2,3)]]
## Error in x[[c(2, 3)]]: attempt to select more than one element in vectorIndex
```

`[[` wird für atomare Vektoren genutzt, um zu verdeutlichen, dass genau ein Element ausgewählt wird und um Fehler zu vermeiden.

```
# Anstatt:
if (x[i] > 0 & y[j] < 0) do_stuff()
# ist es besser Stil zu schreiben:
if (x[[i]] > 0 && y[[j]] < 0) do_stuff()
```

Hinweis: `&&` und `||` sind die nicht-vektorisierten Versionen von `&` und `|`. Da `if`-Ausdrücke genau einen Wahrheitswert verlangen, wird dort bevorzugt `&&`, `||` und `[[` benutzt anstatt `&`, `|`, `[`.

Bei den selten vorkommenden Listen-Matrizen (oder Listen-Arrays) ist auch Indizierung mit einem Index pro Dimension in `[[` erlaubt.

```
lst_mat <- matrix(lapply(1:6, sample), nrow=2)
lst_mat
##      [,1]      [,2]      [,3]
## [1,] 1         Integer,3 Integer,5
## [2,] Integer,2 Integer,4 Integer,6
lst_mat[3]
## [[1]]
## [1] 3 1 2
lst_mat[1,2]
## [[1]]
## [1] 3 1 2
lst_mat[[3]]
## [1] 3 1 2
lst_mat[[1,2]]
## [1] 3 1 2
```

3.2 $

`x$y` entspricht in etwa `x[["y"]]`. Damit können benannte Listeneinträge und Spalten von Tibbles ausgewählt werden.

Wenn der Name einer Spalte in einer Variable steht, kann `$` nicht genutzt werden.

```
tbl <- tibble(number=1:5, letter=letters[1:5])
var <- "number"
tbl$var
## Warning: Unknown or uninitialised column: `var`.
## NULL
tbl[[var]]
## [1] 1 2 3 4 5
```

Im Gegensatz zu `[[` wird bei `$` **partial matching** durchgeführt wird. Dabei genügt es, den Anfang des Namens eines Eintrages zu schreiben, um diesen auszuwählen, solange dadurch das Element eindeutig identifiziert wird (**prefix condition**).

```
x <- list(abc = -5, anders = 7)
x$ab
## [1] -5
x$an
## [1] 7
x$a # nicht eindeutig
## NULL
x[["ab"]]
## NULL
x[["ander"]]
## NULL
```

Da dies sehr leicht zu Fehlern führen kann, ist empfohlen diese Funktionalität nicht auszunutzen. Um vor *partial matching* zu warnen, kann die globale Option `warnPartialMatchDollar` gesetzt werden.

```
options(warnPartialMatchDollar = TRUE)
x$an
## Warning in x$an: partial match of 'an' to 'anders'
## [1] 7
```

4 Fehlende Indizes und Indizes Out-of-Bounds

Indizes größer als die Länge des Vektors und Index-Strings, die nicht zu den Namen der Elemente des Vektors gehören, sind **out-of-bounds**.

Die Subsetting-Operatoren verhalten sich bei *out-of-bounds*-Werten unterschiedlich.

```
x <- 1:3
x[4]
## [1] NA
x[[4]]
## Error in x[[4]]: subscript out of bounds
lst <- as.list(x)
lst[4]
## [[1]]
## NULL
lst[[4]]
## Error in lst[[4]]: subscript out of bounds
names(x) <- letters[1:3]
names(lst) <- letters[1:3]
x["d"]
## <NA>
##   NA
x[["d"]]
## Error in x[["d"]]: subscript out of bounds
lst["d"]
## $<NA>
## NULL
lst[["d"]]
## NULL
lst$d
## NULL
```

Vektor-Typ	`[.integer`	`[[.integer`	`[.character`	`[[.character`	`$`
atomar	`NA`	*ERROR*	`NA`	*ERROR*	-
list	`list(NULL)`	*ERROR*	`list(NULL)`	`NULL`	`NULL`

Für benannte Vektoren wird der Name von fehlenden Elementen als `"<NA>"` angezeigt.

```
x <- c(a=1, 2, b=3)
x[1:4]
##    a          b <NA>
##    1    2    3   NA
```

Indizierung mit `NA`-Werten (`NA_integer_`, `NA`) ergibt `NA`-Werte.

```
x <- 1:3
x[NA_integer_]
## [1] NA
x[NA]  # NA ist logical -> recycling zu x[c(NA, NA, NA)]
## [1] NA NA NA
```

5 Subsetting und Zuweisung

Alle Subsetting-Operatoren können mit dem Zuweisungsoperator `<-` kombiniert werden.

```
x <- 1:5
x[c(1, 2)] <- 11:12
x
## [1] 11 12  3  4  5

x[[3]] <- 23
x
## [1] 11 12 23  4  5

x[-1] <- 32:35
x
## [1] 11 32 33 34 35

x[c(1, 1, 1)] <- 41:43 # Effekt schlecht vorhersehbar -> Vermeiden!
x
## [1] 43 32 33 34 35

x[c(1, NA)] <- c(1, 2) # ERROR
## Error in x[c(1, NA)] <- c(1, 2): NAs are not allowed in subscripted assignments

x[c(T, F, NA)] <- 50 # NA wird bei Zuweisung wie FALSE behandelt
x
## [1] 50 32 33 50 35

# obiges Verhalten ist nützlich in Kombination mit bedingter Zuweisung
tb <- tibble(a = c(1, 10, NA))
tb$a < 5
## [1]  TRUE FALSE    NA
tb$a[tb$a < 5] <- 0
tb$a
## [1]  0 10 NA
```

Partial matching bei `$` wird nicht bei der Zuweisung durchgeführt.

```
x <- list(abc = 1)
x$ab
## Warning in x$ab: partial match of 'ab' to 'abc'
## [1] 1
x$ab <- 2
str(x)
## List of 2
##  $ abc: num 1
##  $ ab : num 2
```

Bei der Zuweisung kommt es ggf zur Typenumwandlung (*Coercion*).

```
x <- 1:4
x[3:4] <- c(T,F)
x
## [1] 1 2 1 0
x[1] <- "0"
x
## [1] "0" "2" "1" "0"
```

Betrachte die Zuweisung `x[i] <- y` für Vektoren `x` und `y` gleichen Typs mit `length(x[i])` gleich $n > 0$ und `length(y)` gleich $m > 0$. Sei $k = \lceil n/m \rceil \in \mathbb{N}$ die kleinste natürliche Zahl sodass $mk \geq n$. Dann wird der Vektor `y` k-mal wiederholt und die ersten n Elemente `x[i]` zugewiesen. Dies entspricht `x[i] <- rep(y, k)[1:n]`. Ist $mk = n$ vereinfacht sich dies zu `x[i] <- rep(y, k)`. Ist $mk \neq n$ wird eine Warnung ausgegeben. Dieses Verhalten wird als **Recycling** bezeichnet.

```
x <- 1:9
x[1:6] <- 11:13 # n=6, m=3, k=2
x
## [1] 11 12 13 11 12 13  7  8  9
x[7:10] <- 22 # n=4, m=1, k=4
x
##  [1] 11 12 13 11 12 13 22 22 22 22
x[1:6] <- 31:34 # n=6, m=4, k=2
## Warning in x[1:6] <- 31:34: number of items to replace is not a multiple of
## replacement length
x
##  [1] 31 32 33 34 31 32 22 22 22 22
x[1:2] <- 41:43 # n=2, m=3, k=1
## Warning in x[1:2] <- 41:43: number of items to replace is not a multiple of
## replacement length
x
##  [1] 41 42 33 34 31 32 22 22 22 22
lst <- as.list(1:8)
lst[2:7] <- list(T, "up") # n=6, m=2, k=3
str(lst)
## List of 8
##  $ : int 1
##  $ : logi TRUE
##  $ : chr "up"
##  $ : logi TRUE
##  $ : chr "up"
##  $ : logi TRUE
##  $ : chr "up"
```

```
##  $ : int 8
```

Es ist möglich *out-of-bounds* Zuweisungen durchzuführen, wodurch der Vektor automatisch vergrößert wird. Dies ist besonders für Listen praktisch, kann aber auch zu Performance-Problemen führen.

```
lst <- list()
lst$a <- 1
lst[["b"]] <- 2
lst[[3]] <- 3
str(lst)
## List of 3
##  $ a: num 1
##  $ b: num 2
##  $  : num 3

N <- 1e7
system.time({
  x <- integer(0)
  for (i in 1:N) x[i] <- 42L # out of bounds assignment
})
##    user  system elapsed
##    2.04    0.14    2.19
system.time({
  x <- integer(N)
  for (i in 1:N) x[i] <- 42L # in bounds assignment
})
##    user  system elapsed
##    0.33    0.00    0.33
```

Subsetting mit *nichts* kann in Verbindung mit Zuweisung nützlich sein, da Attribute des ursprünglichen Objektes erhalten bleiben.

```
tb <- tibble(x=2^(1:4), y=c("1.2", "2.3", "3.4", "4.5"))
tb <- lapply(tb, as.integer)
class(tb) # explizite Klasse wurde nicht übertragen
## [1] "list"
tb <- tibble(x=2^(1:4), y=c("1.2", "2.3", "3.4", "4.5"))
tb[] <- lapply(tb, as.integer)
class(tb) # Tibble-Klasse bleibt erhalten
## [1] "tbl_df"     "tbl"        "data.frame"
```

Die Kombination aus Subsetting und Zuweisungsoperator notieren wir hier `[<-`, `[[<-`, bzw `$<-`.

Listenelemente können mittels `[[<-` und `NULL` aus der Liste entfernt werden. Um `NULL` einer Liste hinzuzufügen, kann `[<-` mit `list(NULL)` genutzt werden.

```
x <- list(a = 1, b = 2)
x[["b"]] <- NULL
str(x)
## List of 1
##  $ a: num 1
y <- list(a = 1)
y["b"] <- list(NULL)
str(y)
## List of 2
##  $ a: num 1
```

```
##  $ b: NULL
```

6 Anwendungen

6.1 Lookup-Tabellen

Um für Abkürzungen den vollständigen Ausdruck nachzuschlagen, erstellen wir einen benannten `character`-Vektor. Die Namen sind die Abkürzungen, die Werte entsprechen den vollständigen Begriffen.

```
x <- c("m", "w", "u", "w", "m")
lookup <- c(m = "männlich", w = "weiblich", u = NA)
lookup[x]
##          m          w          u          w          m
## "männlich" "weiblich"         NA "weiblich" "männlich"
unname(lookup[x])
## [1] "männlich" "weiblich" NA          "weiblich" "männlich"
```

6.2 Matching and Merging

Angenommen wir haben eine Lookup-Tabelle mit mehreren Spalten. Mit der Funktion `match()` finden wir die richtigen Indizes in der Lookup-Tabelle und können damit unseren Datensatz erweitern.

```
students <- tibble(
  name = c("Alice", "Bob", "Carl", "Dave", "Eve"),
  grade = c(1, 2, 2, 3, 1)
)
students
## # A tibble: 5 x 2
##   name  grade
##   <chr> <dbl>
## 1 Alice     1
## 2 Bob       2
## 3 Carl      2
## 4 Dave      3
## 5 Eve       1
info <- tibble(
  grade = 3:1,
  desc = c("Poor", "Good", "Excellent"),
  fail = c(T, F, F)
)
info
## # A tibble: 3 x 3
##   grade desc      fail
##   <int> <chr>     <lgl>
## 1     3 Poor      TRUE
## 2     2 Good      FALSE
## 3     1 Excellent FALSE
id <- match(students$grade, info$grade)
id
## [1] 3 2 2 1 3
info[id, ]
## # A tibble: 5 x 3
##   grade desc      fail
##   <int> <chr>     <lgl>
```

```
## 1     1 Excellent FALSE
## 2     2 Good      FALSE
## 3     2 Good      FALSE
## 4     3 Poor      TRUE
## 5     1 Excellent FALSE
cbind(students, info[id, -1])
##    name grade      desc  fail
## 1 Alice     1 Excellent FALSE
## 2   Bob     2      Good FALSE
## 3  Carl     2      Good FALSE
## 4  Dave     3      Poor  TRUE
## 5   Eve     1 Excellent FALSE
```

Etwas kürzer ist dies mit der Funktion `merge()` zu bewerkstelligen.

```
merge(x=students, y=info, by.x="grade", by.y="grade")
##   grade  name      desc  fail
## 1     1 Alice Excellent FALSE
## 2     1   Eve Excellent FALSE
## 3     2   Bob      Good FALSE
## 4     2  Carl      Good FALSE
## 5     3  Dave      Poor  TRUE
# Da grade die einzge Spalte ist, die in beiden Tibbles vorkommt, geht es noch kürzer:
stu <- merge(students, info)
stu
##   grade  name      desc  fail
## 1     1 Alice Excellent FALSE
## 2     1   Eve Excellent FALSE
## 3     2   Bob      Good FALSE
## 4     2  Carl      Good FALSE
## 5     3  Dave      Poor  TRUE
```

6.3 Random samples / bootstrap

`sample(x, n)` wählt `n` Elemente des Vektors `x` zufällig ohne zurücklegen (bei `replace=TRUE` mit zurücklegen) aus. Siehe `?sample`. Um aus gegebenen Daten neue, 'zufällige' Daten zu generieren, wählen wir zufällige Zeilen-Indizes aus.

```
stu[sample(nrow(tb)), ] # Randomly reorder
##   grade  name      desc  fail
## 3     2   Bob      Good FALSE
## 1     1 Alice Excellent FALSE
## 4     2  Carl      Good FALSE
## 2     1   Eve Excellent FALSE
stu[sample(nrow(tb), 3), ] # Select 3 random rows
##   grade  name      desc  fail
## 1     1 Alice Excellent FALSE
## 2     1   Eve Excellent FALSE
## 3     2   Bob      Good FALSE
stu[sample(nrow(tb), 6, replace = TRUE), ] # Select 6 bootstrap replicates
##     grade name      desc  fail
## 3       2  Bob      Good FALSE
## 3.1     2  Bob      Good FALSE
## 4       2 Carl      Good FALSE
## 4.1     2 Carl      Good FALSE
```

```
## 2       1  Eve Excellent FALSE
## 3.2     2  Bob       Good FALSE
```

6.4 Sortieren

Sortiere nach Zeilen oder Spalten eines Tibbles mittels `order()`.

```
x <- c(20, 30, 10)
order(x)
## [1] 3 1 2
x[order(x)]
## [1] 10 20 30

stu2 <- stu[sample(nrow(stu)), ] # randomly perturb
stu2
##   grade  name      desc  fail
## 1     1 Alice Excellent FALSE
## 2     1   Eve Excellent FALSE
## 3     2   Bob      Good FALSE
## 5     3  Dave      Poor  TRUE
## 4     2  Carl      Good FALSE
stu2[order(stu2$grade), ] # sort rows by grade
##   grade  name      desc  fail
## 1     1 Alice Excellent FALSE
## 2     1   Eve Excellent FALSE
## 3     2   Bob      Good FALSE
## 4     2  Carl      Good FALSE
## 5     3  Dave      Poor  TRUE
stu2[, order(names(stu2))] # sort columns by name
##        desc  fail grade  name
## 1 Excellent FALSE     1 Alice
## 2 Excellent FALSE     1   Eve
## 3      Good FALSE     2   Bob
## 5      Poor  TRUE     3  Dave
## 4      Good FALSE     2  Carl
```

Sortieren basiert auf dem Vergleichsoperatoren. Diese sind auch für `character`-Werte definiert und folgen der **lexikographischen Ordnung**. (siehe auch `?"<"`)

```
"a" < "A"
## [1] TRUE
"A" < "ä"
## [1] TRUE
"ä" < "aa"
## [1] TRUE
"aa" < "ab"
## [1] TRUE
"ab" < "b"
## [1] TRUE
```

Beachte den Unterschied zwischen `sort()` (sortierter atomarer Vektor), `order()` (Indizes zur Sortierung), `rank()` (Rang: "1. Platz", "2. Platz", ...)

```
x <- c(40, 20, 10, 30)
sort(x)
## [1] 10 20 30 40
```

```
rank(x)
## [1] 4 2 1 3
order(x)
## [1] 3 2 4 1
tb <- tibble(num = x, rank=rank(x))
tb
## # A tibble: 4 x 2
##     num  rank
##   <dbl> <dbl>
## 1    40     4
## 2    20     2
## 3    10     1
## 4    30     3
tb[order(x),]
## # A tibble: 4 x 2
##     num  rank
##   <dbl> <dbl>
## 1    10     1
## 2    20     2
## 3    30     3
## 4    40     4
```

6.5 Aggregierte Zeilen expandieren

Angenommen mehrfach vorkommende Beobachtungen werden nur einmal gelistet. Dafür wird in einer zusätzlichen Spalte (`n`) ihre Häufigkeit notiert. Um den expandierten Datensatz zu erhalten kann ein entsprechende Index-Vektor mit `rep()` erzeugt werden.

```
tb <- tibble(x = c(2, 4, 1), y = c(9, 11, 6), n = c(3, 5, 1))
rep(1:nrow(tb), tb$n)
## [1] 1 1 1 2 2 2 2 2 3
tb[rep(1:nrow(tb), tb$n), ]
## # A tibble: 9 x 3
##       x     y     n
##   <dbl> <dbl> <dbl>
## 1     2     9     3
## 2     2     9     3
## 3     2     9     3
## 4     4    11     5
## 5     4    11     5
## 6     4    11     5
## 7     4    11     5
## 8     4    11     5
## 9     1     6     1
```

6.6 Spalten aus Tibbles entfernen

Um Spalten zu entfernen, kann man sie auf `NULL` setzten.

```
tib <- tibble(x = 1:3, y = 3:1, z = letters[1:3])
tib$z <- NULL
```

Alternativ können nur alle übrigen Spalten ausgewählt werden.

```
tib$z <- letters[1:3]
tib[c("x", "y")]
## # A tibble: 3 x 2
##       x     y
##   <int> <int>
## 1     1     3
## 2     2     2
## 3     3     1
```

Falls dabei nur die zu entfernenden Spalten beschrieben werden sollen, ist `setdiff()` hilfreich.

```
names(tib)
## [1] "x" "y" "z"
setdiff(names(tib), "z")
## [1] "x" "y"
tb[setdiff(names(tib), "z")]
## # A tibble: 3 x 2
##       x     y
##   <dbl> <dbl>
## 1     2     9
## 2     4    11
## 3     1     6
```

6.7 Konditionales Subsetting

Eine der nützlichsten Subsetting-Funktionalitäten in R ist das Subsetting mit `logical`-Vektoren in Kombination mit logischen Operatoren, um nur Daten auszuwählen die bestimmte Bedingungen erfüllen.

```
head(mtcars) # betrachte die obersten Zeilen des Datensatzes mtcars
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
# siehe ?mtcars
mtcars[mtcars$gear == 5, ]
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
mtcars[mtcars$gear == 5 & mtcars$cyl == 4, ]
##               mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa  30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
```

Achtung: Nutze hierbei `&` und `|` (vektorisiert) und nicht `&&` und `||` (skalare Operatoren).

6.8 Boolesche Algebra vs Mengen

Es gibt eine natürliche Äquivalenz zwischen `integer`-Subsetting (Mengenoperationen) und `logical` Subsetting (Boolesche Algebra).

Mengenoperationen sind effektiver, falls das erste oder letzte `TRUE` gefunden werden soll oder falls die meisten Einträge des `logical`-Vektors `FALSE` sind.

`which()` konvertiert einen `logical` Index-Vektor in die entsprechende `integer`-Repräsentation.

```
x <- sample(7)
b <- x < 4
x
## [1] 5 6 1 2 3 4 7
b
## [1] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
which(b)
## [1] 3 4 5
x[which(b)[1]] # erster Wert < 4
## [1] 1
```

Die Umkehrfunktion ist in Base-R nicht implementiert. Wir können sie jedoch einfach selbst schreiben.

```
unwhich <- function(b, n) {
  out <- rep(FALSE, n)
  out[b] <- TRUE
  out
}
unwhich(which(b), length(b))
## [1] FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
```

Achtung: `x[-which(b)]` ist nicht äquivalent zu `x[!b]`: Falls alle Einträge von `b` gleich `FALSE` sind, ist `which(b)` gleich `integer(0)` und `-integer(0)` ist immer noch `integer(0)`.

```
x <- 1:3
b <- c(F, T, F)
x[!b]
## [1] 1 3
x[-which(b)] # gleiches Resultat
## [1] 1 3
b <- c(F, F, F)
x[!b]
## [1] 1 2 3
x[-which(b)] # verschiedene Resultate
## integer(0)
```

Es gibt Äquivalenzen auch zwischen logischen und den Mengenoperatoren. Seien `bx` und `by` `logical`-Vektoren mit den zugehörigen Index-Vektoren `ix <- which(bx)` und `iy <- which(by)`.

- `bx & by` entspricht `intersect(ix, iy)`,
- `bx | by` entspricht `union(ix, iy)`,
- `bx & !by` entspricht `setdiff(ix, iy)`,
- `xor(bx, by)` entspricht `setdiff(union(ix, iy), intersect(ix, iy))`.

```
bx <- 1:10 %% 2 == 0
ix <- which(bx)
bx
##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
ix
## [1]  2  4  6  8 10
by <- 1:10 %% 5 == 0
iy <- which(by)
by
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
iy
## [1]  5 10

bx & by
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
intersect(ix, iy)
## [1] 10

bx | by
##  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
union(ix, iy)
## [1]  2  4  6  8 10  5

bx & !by
##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE
setdiff(ix, iy)
## [1] 2 4 6 8

xor(bx, by)
##  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
setdiff(union(ix, iy), intersect(ix, iy))
## [1] 2 4 6 8 5
```