02 – Datenstrukturen

19. April 2021

Contents

1 Objekte

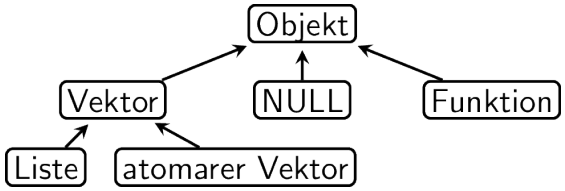
> "To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call."
>
> — *John M. Chambers*

Alles, was *existiert*, ist ein **Objekt**. Alles, was *passiert*, ist ein **Funktionsaufruf**.

Ein **R-Skript** ist eine Folge von Funktionsaufrufen getrennt durch `;` oder Zeilenumbruch.

Beispiele für Objekte sind **Listen** und **atomare Vektoren**. Sie werden zusammen als **Vektoren** bezeichnet und sind Objekte zum Speichern von Daten. Funktionen sind ebenfalls Objekte. `NULL` repräsentiert das **Null-Objekt**, dazu später mehr.

```
NULL
## NULL
```



Objekte haben (unter anderem) folgende Eigenschaften:

- `typeof(x)` ist der **Typ** von `x`
- `class(x)` ist die **Klasse** von `x`
- `length(x)` ist die **Länge** von `x`

```
object_properties <- function(x) c(typeof(x), class(x), length(x))
object_properties(NULL)
## [1] "NULL" "NULL" "0"
object_properties(list(1, "asdf"))
## [1] "list" "list" "2"
object_properties(c(1, pi, exp(1)))
## [1] "double"  "numeric" "3"
object_properties(mean)
## [1] "closure"  "function" "1"
```

Wir werden später noch genauer auf den Typ und die Klasse eingehen.

2 Vektoren

2.1 Atomare Vektoren

Alle Elemente eines atomaren Vektors haben den gleichen Typ und dieser Typ muss ein Basistyp sein.

Die **6 Basistypen** sind: `logical`, `integer`, `double`, `character`, `complex`, `raw`.

```
misc_atomic <- list(T, 1:3, pi, "asdf", 2+3i, raw(4))
sapply(misc_atomic, typeof)
## [1] "logical"   "integer"   "double"    "character" "complex"   "raw"
```

Die Typen `complex`, `raw` treten eher selten auf und werden in der Vorlesung nicht näher behandelt.

`typeof()` gibt den entsprechenden Basistyp aus. `class()` gibt für atomare Vektoren fast das Gleiche aus aber anstatt `"double"` den Wert `"numeric"`.

```
sapply(misc_atomic, class)
## [1] "logical"   "integer"   "numeric"   "character" "complex"   "raw"
```

Die Funktionen `is.logical()`, `is.integer()`, `is.double()`, `is.character()` testen den Typ eines atomaren Vektors. `is.numeric()` ist für den Test auf `integer` oder `double` geeignet. `is.atomic()`, `is.list()`, `is.vector()` testen entsprechend, ob ihr Argument atomar, eine Liste bzw. ein Vektor ist.

```
# wende is.-Funktionen auf verschiedene Objekte an
is_type <- function(x) c(is.logical(x), is.integer(x), is.double(x), is.numeric(x),
                         is.character(x), is.atomic(x), is.list(x), is.vector(x))
misc <- c(misc_atomic, list(list(), matrix(1:4, ncol=2), NULL))
res <- sapply(misc, is_type)
rownames(res) <- paste("is.", c("logi", "inte", "doub", "nume", "char", "atom",
                                "list", "vect"), sep="")
```

```
colnames(res) <- sapply(misc, function(x) class(x)[1])
print(res)
##         logical integer numeric character complex   raw  list matrix  NULL
## is.logi    TRUE   FALSE   FALSE     FALSE   FALSE FALSE FALSE  FALSE FALSE
## is.inte   FALSE    TRUE   FALSE     FALSE   FALSE FALSE FALSE   TRUE FALSE
## is.doub   FALSE   FALSE    TRUE     FALSE   FALSE FALSE FALSE  FALSE FALSE
## is.nume   FALSE    TRUE    TRUE     FALSE   FALSE FALSE FALSE   TRUE FALSE
## is.char   FALSE   FALSE   FALSE      TRUE   FALSE FALSE FALSE  FALSE FALSE
## is.atom    TRUE    TRUE    TRUE      TRUE    TRUE  TRUE FALSE   TRUE  TRUE
## is.list   FALSE   FALSE   FALSE     FALSE   FALSE FALSE  TRUE  FALSE FALSE
## is.vect    TRUE    TRUE    TRUE      TRUE    TRUE  TRUE  TRUE  FALSE FALSE
```

`NULL` ist atomar ist insofern, dass es keine Teilelemente haben kann, die nicht `NULL` sind. Allerdings ist `NULL` kein Vektor. Der Zusammenhang zwischen Matrizen und Vektoren wird später genauer erklärt.

Wir können atomare Vektoren der Länge `n` mittels Funktionen der Form `BASE_TYPE(n)` erzeugen. Diese Vektoren sind mit den entsprechenden Default-Werten gefüllt.

```
# Die Funktion str() gibt die "Struktur" ihres Argumentes aus. Diese
# Ausgabe ist oft Informativer als print().
str(list(logical(3), integer(3), double(3), character(3)))
## List of 4
##  $ : logi [1:3] FALSE FALSE FALSE
##  $ : int [1:3] 0 0 0
##  $ : num [1:3] 0 0 0
##  $ : chr [1:3] "" "" ""
# Achtung: str() kürzt 'double' mit 'num' ab...
```

Skalare sind atomare Vektoren der Länge 1. In R gibt es keine eigenen Typen oder Datenstrukturen nur für Skalare (anders als zB in C)

```
is.atomic(0)
## [1] TRUE
length(0)
## [1] 1
0[1]
## [1] 0
```

Greifen wir auf Vektoren an einer Stelle zu, die keinen definierten Wert hat, so wird `NA` zurückgegeben. Wir können Zuweisungen an Indizes größer der Länge des Vektors durchführen.

```
x <- 1:3
length(x)
## [1] 3
x[2:6]
## [1]  2  3 NA NA NA
x[5] <- 42
x
## [1]  1  2  3 NA 42
length(x)
## [1] 5
```

2.1.1 logical

Mögliche Werte eines Skalars vom Typ `logical` sind `TRUE`, `FALSE`, `NA` (not available). `T`, `F` dienen als Abkürzungen für `TRUE` und `FALSE`, sind aber keine reservierten Wörter.

```
T
## [1] TRUE
T <- FALSE # evil
T
## [1] FALSE
T <- TRUE # undo
typeof(NA)
## [1] "logical"
```

2.1.2 integer

`integer`-Werte enthalten eine endliche Teilmenge der ganzen Zahlen und `NA_integer_`. Sie werden mit dem Suffix `L` notiert, zB `42L`. Bei der Ausgabe auf der Konsole wird das Suffix nicht ausgegeben.

Eine der wenigen Operatoren, die standardmäßig den Typ `integer` ausgeben ist `:`

```
sapply(list(42, 42L, NA_integer_, 1:3, 2L*2L, 2L^2L), typeof)
## [1] "double"  "integer" "integer" "integer" "integer" "double"
.Machine$integer.max # maximaler integer-Wert
## [1] 2147483647
as.integer(.Machine$integer.max+1)
## Warning: NAs introduced by coercion to integer range
## [1] NA
```

2.1.3 double

`double`-Werte enthalten eine endliche Teilmenge der reellen Zahlen und `NA_real_` (*not available*), `Inf` (unendlich), `-Inf`, `NaN` (*not a number*). Zahlen ohne Suffix `L` werden als `double` interpretiert (zB `1`, `2.3`). Double-Werte können mit der E-Notation (zB `12e-34`) eingegeben werden.

```
lst <- list(0, pi, 1e3, Inf, NaN, NA_real_)
sapply(lst, typeof)
## [1] "double" "double" "double" "double" "double" "double"
unlist(lst)
## [1]    0.000000    3.141593 1000.000000         Inf         NaN          NA
```

Beachte Maschinengenauigkeit beim Rechnen mit `double`-Werten.

```
sqrt(2)^2 == 2
## [1] FALSE
sqrt(2)^2 - 2
## [1] 4.440892e-16
```

`double`-Werte sind im IEEE 754 Standard gespeichert. Daraus ergeben sich die Genauigkeit sowie die maximalen und minimalen Werte. Diese sind in der Liste `.Machine` angegeben. Dabei bezeichnet `eps` den Abstand zwischen `1` und der nächst größeren darstellbaren Zahl.

```
str(.Machine[1:13])
## List of 13
##  $ double.eps           : num 2.22e-16
##  $ double.neg.eps       : num 1.11e-16
##  $ double.xmin          : num 2.23e-308
##  $ double.xmax          : num 1.8e+308
##  $ double.base          : int 2
##  $ double.digits        : int 53
##  $ double.rounding      : int 5
##  $ double.guard         : int 0
```

```
##  $ double.ulp.digits    : int -52
##  $ double.neg.ulp.digits: int -53
##  $ double.exponent      : int 11
##  $ double.min.exp       : int -1022
##  $ double.max.exp       : int 1024
1 == 1 + .Machine$double.eps
## [1] FALSE
1 == 1 + .Machine$double.eps/2
## [1] TRUE
.Machine$double.xmax
## [1] 1.797693e+308
.Machine$double.xmax*2
## [1] Inf
```

Statt `double`-Vektoren auf exakte Gleichheit zu testen, prüft man meist, ob der Betrag der Differenz sehr klein ist.

```
abs(c(sin(pi), sqrt(2)^2) - c(0, 2)) < 1e-14
## [1] TRUE TRUE
# Achtung: Die Größenordung des Fehlers ändert sich durch manche Operationen
sqrt(sqrt(2)^2-2)
## [1] 2.107342e-08
```

Bei der Ausgabe auf der Konsole werden standardmäßig 7 geltende Ziffern ausgegeben. Es können aber 15 bis 16 Dezimalstellen gespeichert und berechnet werden.

```
98765432109876543210
## [1] 9.876543e+19
98765432109876543210 - 98765432100000000000 # 15 Ziffern korrekt
## [1] 9876537344
1 + 1e-20 # Ausgabe als 1
## [1] 1
1 + 1e-20 - 1 # Rechnen mit 1 statt 1e-20
## [1] 0
1 + 1e-10 # Ausgabe als 1 (7 geltende Ziffern)
## [1] 1
1 + 1e-10 - 1 # Rechnen mit 1 + 1e-10
## [1] 1e-10
```

Mit `Inf`, `-Inf`, `NaN` und `NA_real_` kann gerechnet werden.

```
1 / 0
## [1] Inf
-1 / 0
## [1] -Inf
0/0
## [1] NaN
log(0)
## [1] -Inf
1 / Inf
## [1] 0
Inf - Inf
## [1] NaN
Inf + Inf
## [1] Inf
Inf / Inf
```

```
## [1] NaN
-Inf * Inf
## [1] -Inf
1 + NaN
## [1] NaN
NaN - NaN
## [1] NaN
1 + NA_real_
## [1] NA
NA_real_ - NA_real_
## [1] NA
NA_real_ - NaN
## [1] NA
NaN - NA_real_
## [1] NaN
```

2.1.4 character

`character`-Werte (Strings) sind Text, notiert in einfachen oder doppelten Anführungszeichen, zB `"blub"` oder `'bla'`. Der Text kann **Escape-Sequenzen** enthalten zB neue Zeile: `\n`, backslash: `\\`, Anführungszeichen `\"`, `\'`, Unicode `\U....`

```
"einfache ' Anführungszeichen"
## [1] "einfache ' Anführungszeichen"
'doppelte " Anführungszeichen'
## [1] "doppelte \" Anführungszeichen"
str <- "1. Zeile\n\"2. Zeile\", backslash \\"
print(str)
## [1] "1. Zeile\n\"2. Zeile\", backslash \\"
cat(str)
## 1. Zeile
## "2. Zeile", backslash \
x <- c("ä", "Hallo", "", NA_character_, "\n")
typeof(x)
## [1] "character"
length(x)
## [1] 5
nchar(x)
## [1]  1  5  0 NA  1
```

Ab R Version 4 können **raw** Strings, dh ohne Interpretation von Escape-Sequenzen, mit `r"(...)"` angegeben werden, wobei `...` eine Sequenz von Symbolen ist, die nicht die Sequenz `)"` enthält.

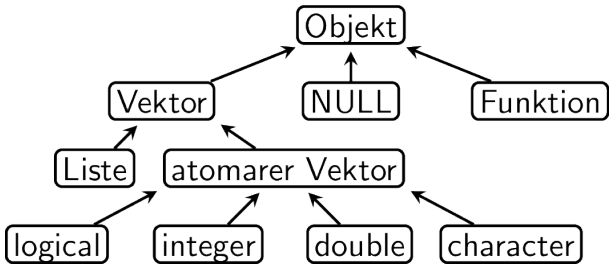
```
cat(r"(\n\\'"\'\")")
## \n\\'"\'\"
```

In R gibt es 4 `character`-Vektoren, die direkt verfügbar sind.

```
letters
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
LETTERS
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
month.name
##  [1] "January"   "February"  "March"     "April"     "May"       "June"
```

```
##  [7] "July"      "August"    "September" "October"   "November"  "December"
month.abb
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

2.1.5 Übersicht



Typ	Klasse	not available	erzeugen	Default
logical	logical	NA	logical(n)	FALSE
integer	integer	NA_integer_	integer(n)	0L
double	numeric	NA_real_	double(n)	0
character	character	NA_character_	character(n)	""

Achtung: Alle `NA`-Werte werden in der Konsole als `NA` ausgegeben.

```
NA
## [1] NA
NA_integer_
## [1] NA
NA_real_
## [1] NA
NA_character_
## [1] NA
```

2.2 Combine

`c()` (**combine**) kombiniert Vektoren (gleichen Typs) *flach*: `c(x,c(y,z))` gleich `c(c(x,y),z)` gleich `c(x,y,z)` (Assoziativität).

```
c(1, c(2, 3:4))
## [1] 1 2 3 4
c(c(1, 2), 3:4)
## [1] 1 2 3 4
vec <- c(c(list(1), list(T)), list("asdf", list()))
str(vec)
## List of 4
##  $ : num 1
##  $ : logi TRUE
##  $ : chr "asdf"
##  $ : list()
vec <- c(list(1), list(T), list("asdf", list()))
str(vec)
## List of 4
##  $ : num 1
##  $ : logi TRUE
```

```
##  $ : chr "asdf"
##  $ : list()
```

`NULL` ist ein "neutrales Element" von `c()`.

```
c(1:3, NULL, 11:13)
## [1]  1  2  3 11 12 13
str(c(NULL, list(1), NULL, list(T), NULL))
## List of 2
##  $ : num 1
##  $ : logi TRUE
```

2.3 Typenumwandlung

Umwandlung eines Objektes von einem Typ in einen anderen wird in R mit **Coercion** (dt: Zwang) bezeichnet. Typenumwandlung kann explizit gefordert werden oder implizit geschehen. (Bei anderen Programmiersprachen ist das Wort Coercion zT nur für implizite Typenumwandlung reserviert).

Explizit kann Typenumwandlung durch `as.logical()`, `as.integer()`, `as.double()`, `as.character()` durchgeführt werden.

```
x <- as.double(c("1.2", "3.4", "1e-2"))
x
## [1] 1.20 3.40 0.01
sum(x)
## [1] 4.61
```

Bei der Nutzung von `c()` kann es zu impliziter Coercion kommen. Es werden ggf *speziellere* Typen in *allgemeinere* umgewandelt. Von speziell nach allgemein ist die Typen-Ordnung: `logical`, `integer`, `double`(, `complex`), `character`(, `raw`)

```
c(TRUE, 15)
## [1]  1 15
c(42, FALSE)
## [1] 42  0
typeof(c(1L, 2))
## [1] "double"
c(TRUE, 1L, 1, "eins")
## [1] "TRUE" "1"    "1"    "eins"
c(c(TRUE, 1L), 1, "eins") # Assoziativität durch Coercion gebrochen
## [1] "1"    "1"    "1"    "eins"
```

Auch viele Funktionen wandeln ggf die Typen ihrer Argumente um, zB kann die Anzahl gerader Elemente im atomaren Vektor `x` durch `sum(x %% 2 == 0)` berechnet werden.

```
sum(c(T,F,F,T,T))
## [1] 3
x <- 1:100
sum(x %% 7 == 0) # Anzahl Vielfacher von 7 in x
## [1] 14
```

2.4 Listen

Mit einer Liste können mehrere Objekte zu einer Einheit zusammengefasst werden. Möchte man in einer Funktion mehr als ein Objekt zurückgeben, können diese zunächst in eine Liste geschrieben und dann die Liste zurückgegeben werden.

Die Elemente von Listen können verschiedene, beliebige Typen haben.

```
lst <- list(
  1:3,
  "a",
  c(TRUE, FALSE, TRUE),
  c(2.3, 5.9),
  list(1,"b"),
  sum # Funktion
)
str(lst)
## List of 6
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.3 5.9
##  $ :List of 2
##   ..$ : num 1
##   ..$ : chr "b"
##  $ :function (..., na.rm = FALSE)
```

Listen selbst haben Typ und Klasse `list`. Dies kann mit `is.list()` getestet werden.

```
c(typeof(lst), class(lst), length(lst))
## [1] "list" "list" "6"
c(is.list(lst), is.list(1:3))
## [1]  TRUE FALSE
```

Listen sind Vektoren.

```
is.vector(lst)
## [1] TRUE
```

Der Zugriff auf Elemente erfolgt mit `[[`. Mit `[` werden Unterlisten erzeugt.

```
str(lst[c(2,4)])
## List of 2
##  $ : chr "a"
##  $ : num [1:2] 2.3 5.9
str(lst[2]) # Liste!
## List of 1
##  $ : chr "a"
str(lst[[2]]) # Element der Liste
##  chr "a"
```

Zuweisung von `NULL` löscht Listenelemente.

```
lst[c(2,5,6)] <- NULL
str(lst)
## List of 3
##  $ : int [1:3] 1 2 3
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.3 5.9
lst[1] <- list(NULL) # NULL als Listenelement
str(lst)
## List of 3
##  $ : NULL
##  $ : logi [1:3] TRUE FALSE TRUE
```

```
##  $ : num [1:2] 2.3 5.9
```

Die Funktion `list()` erstellt eine Liste ihrer Argumente, wobei `list()` die leere Liste erzeugt. Falls die Argumente von `c()` Listen sind, werden deren Elemente zu einer Liste zusammengefügt.

```
list()
## list()
str(c(list(), list(1), list(c(T, F), letters[1:3])))
## List of 3
##  $ : num 1
##  $ : logi [1:2] TRUE FALSE
##  $ : chr [1:3] "a" "b" "c"
```

Umwandlungen zwischen Listen und atomaren Vektoren sind mit `unlist()` und `as.list()` möglich.

```
str(list(1:3))
## List of 1
##  $ : int [1:3] 1 2 3
str(as.list(1:3))
## List of 3
##  $ : int 1
##  $ : int 2
##  $ : int 3
lst <- list(1:3, as.list(4:6))
str(lst)
## List of 2
##  $ : int [1:3] 1 2 3
##  $ :List of 3
##   ..$ : int 4
##   ..$ : int 5
##   ..$ : int 6
str(unlist(lst))
##  int [1:6] 1 2 3 4 5 6
```

3 Attribute

Attribute sind Meta-Daten von Objekten.

Die Attribute eines Objektes sind eine ungeordnete Sammlung aus (Name, Wert)–Paaren, wobei Namen maximal einmal vorkommen dürfen. Der Wert kann ein beliebiges Objekt außer `NULL` sein.

Jedes Objekt außer `NULL` kann Attribute haben.

Attribute können mit `attr()`, `attributes()`, `structure()`, `str()` abgefragt bzw gesetzt werden (siehe `?`). `attr(obj, "Name") <- Wert` setzt für das Objekt `obj` das Attribut namens `Name` auf den Wert `Wert`.

```
x <- 1:5
str(x)
##  int [1:5] 1 2 3 4 5
attr(x, "asdf") <- c(1, 7)
str(x)
##  int [1:5] 1 2 3 4 5
##  - attr(*, "asdf")= num [1:2] 1 7
attr(x, "blub") <- list("Hallo", c(T,F,T))
str(x)
##  int [1:5] 1 2 3 4 5
```

```
##  - attr(*, "asdf")= num [1:2] 1 7
##  - attr(*, "blub")=List of 2
##   ..$ : chr "Hallo"
##   ..$ : logi [1:3] TRUE FALSE TRUE
typeof(attributes(x))
## [1] "list"
str(attributes(x))
## List of 2
##  $ asdf: num [1:2] 1 7
##  $ blub:List of 2
##   ..$ : chr "Hallo"
##   ..$ : logi [1:3] TRUE FALSE TRUE

y <- NULL
attr(y, "a") <- "b" # ERROR
## Error in attr(y, "a") <- "b": attempt to set an attribute on NULL
```

3.1 Namen

Das Attribut `names` wird genutzt, um Einträgen eines Vektors Namen zu geben.

Dies kann direkt beim Erstellen des Vektors mit `c()` oder `list()` geschehen oder im Nachhinein mittels `attr(, "names")<-`, oder `names()<-` durchgeführt werden. Bei `c()` und `list()` kann der Name in Anführungszeichen gesetzt werden, bei speziellen Namen muss dies sogar gemacht werden. Sind Namen gesetzt, werden sie bei der Ausgabe auf der Konsole angezeigt.

```
x <- 1:4
attr(x, "names") <- letters[1:4]
x
## a b c d
## 1 2 3 4
str(x)
##  Named int [1:4] 1 2 3 4
##  - attr(*, "names")= chr [1:4] "a" "b" "c" "d"
names(x) <- LETTERS[1:4]
x
## A B C D
## 1 2 3 4

y <- c(eins=1, zwei=2, drei=3)
y
## eins zwei drei
##    1    2    3

# z <- c(a=1, 42=3) # ERROR
z <- c("a"=1, "42"=3, '"'=24) # works
z
##  a 42  "
##  1  3 24

lst <- list(wort="asdf", Zahlen=1:3)
lst
## $wort
## [1] "asdf"
```

```
##
## $Zahlen
## [1] 1 2 3
str(lst)
## List of 2
##  $ wort  : chr "asdf"
##  $ Zahlen: int [1:3] 1 2 3
```

Mittels `$` kann auf benannte Elemente von Listen (nicht von atomaren Vektoren) ohne den Index zugegriffen werden oder neue Elemente erzeugt werden.

```
lst$wort
## [1] "asdf"
lst$Zahlen
## [1] 1 2 3
lst$logisch
## NULL
lst$logisch <- TRUE
lst$"0" <- 123
str(lst)
## List of 4
##  $ wort   : chr "asdf"
##  $ Zahlen : int [1:3] 1 2 3
##  $ logisch: logi TRUE
##  $ 0      : num 123
```

Mit `attr(x, "names")` oder `names(x)` werden alle Namen angezeigt.

```
attr(x, "names")
## [1] "A" "B" "C" "D"
names(y)
## [1] "eins" "zwei" "drei"
```

Um Namen zu entfernen, nutzen wir `unname()`, `names(x) <- NULL` oder `attr(x, "names") <- NULL`.

```
unname(y)
## [1] 1 2 3
y # wurde nicht geändert, call by value!
## eins zwei drei
##    1    2    3
y <- unname(y)
y
## [1] 1 2 3

str(x)
##  Named int [1:4] 1 2 3 4
##  - attr(*, "names")= chr [1:4] "A" "B" "C" "D"
names(x) <- NULL
str(x)
##  int [1:4] 1 2 3 4
```

3.2 Weitere Bemerkungen

Viele Operationen erhalten die Attribute ihrer Inputs nicht. Die einzigen Attribute, die meist erhalten bleiben, sind `dim` und `names`.

```r
z <- structure(1:5, names=letters[1:5], something="some thing")
str(z)
##  Named int [1:5] 1 2 3 4 5
##  - attr(*, "names")= chr [1:5] "a" "b" "c" "d" ...
##  - attr(*, "something")= chr "some thing"
attributes(z[1])
## $names
## [1] "a"
attributes(sum(z))
## NULL
attributes(z) <- NULL # remove all attributes
z
## [1] 1 2 3 4 5
```

Das Attribut `dim` erlaubt es, aus Vektoren Matrizen und Arrays zu machen. Mehr dazu gleich.

Weitere Attribute ermöglichen neue Datenstrukturen auf Vektoren aufzubauen (zB Tibbles, Faktoren, Datenstrukturen für Zeit und Datum, ...). Später mehr dazu.

```r
attributes(tibble::tibble(a=c(7,8), b=c(T,F)))
## $names
## [1] "a" "b"
##
## $row.names
## [1] 1 2
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
```

Names erleichtern den Zugriff auf Elemente einer Liste und machen Code leichter verständlich: `fit$coeff` vs `fit[[1]]`. Attribute sollten jedoch nicht zum Speichern von Daten/Beobachtungen dienen (nur für Meta-Daten).

```r
age <- c(Alice=25, Bob=23) # schlechter Stil, da Namen Teil der Beobachtung sind
age
## Alice   Bob
##    25    23
tibble::tibble(name=c("Alice", "Bob"), age=c(25,23)) # besser
## # A tibble: 2 x 2
##   name    age
##   <chr> <dbl>
## 1 Alice    25
## 2 Bob      23
```

Die Funktion `identical()` überprüft, ob Objekte exakt gleich (identisch) sind. Dies schließt Gleichheit von Attributen mit ein.

```r
x <- 1:3
attr(x, "asdf") <- 42
x == 1:3
## [1] TRUE TRUE TRUE
identical(x, 1:3)
## [1] FALSE
# identical kann auch zum Testen von NA-Werten genutzt werden:
identical(NA, NA)
## [1] TRUE
```

```r
identical(NA, NaN)
## [1] FALSE
NULL == NULL
## logical(0)
identical(NULL, NULL)
## [1] TRUE
```

4 Matrizen und Arrays

Um aus einem Vektor eine **Matrix** zu machen, setzen wir das Attribut `dim` (**Dimension**) auf den atomaren Vektor aus Zeilen- und Spaltenzahl.

```r
x <- 1:6
class(x)
## [1] "integer"
attr(x, "dim") <- c(2,3)
class(x)
## [1] "matrix" "array"
x
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
identical(x, matrix(1:6, nrow=2)) # exakt gleich
## [1] TRUE
```

Matrizen haben 2 Dimensionen. Die Verallgemeinerung auf n Dimensionen wird in R als **Array** bezeichnet. Seit R Version 4 sind Matrizen auch Arrays.

```r
x1 <- structure(1:24, dim=24) # 1D-Array
x1
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
class(x1)
## [1] "array"
x2 <- structure(1:24, dim=c(4,6)) # Matrix (2D-Array)
x2
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    5    9   13   17   21
## [2,]    2    6   10   14   18   22
## [3,]    3    7   11   15   19   23
## [4,]    4    8   12   16   20   24
class(x2)
## [1] "matrix" "array"
x3 <- structure(1:24, dim=2:4) # 3D-Array
x3
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
```

```
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
class(x3)
## [1] "array"
x4 <- structure(1:24, dim=1:4) # 4D-Array
x4
## , , 1, 1
##
##      [,1] [,2]
## [1,]    1    2
##
## , , 2, 1
##
##      [,1] [,2]
## [1,]    3    4
##
## , , 3, 1
##
##      [,1] [,2]
## [1,]    5    6
##
## , , 1, 2
##
##      [,1] [,2]
## [1,]    7    8
##
## , , 2, 2
##
##      [,1] [,2]
## [1,]    9   10
##
## , , 3, 2
##
##      [,1] [,2]
## [1,]   11   12
##
## , , 1, 3
##
##      [,1] [,2]
## [1,]   13   14
##
## , , 2, 3
```

```
##
##      [,1] [,2]
## [1,]   15   16
##
## , , 3, 3
##
##      [,1] [,2]
## [1,]   17   18
##
## , , 1, 4
##
##      [,1] [,2]
## [1,]   19   20
##
## , , 2, 4
##
##      [,1] [,2]
## [1,]   21   22
##
## , , 3, 4
##
##      [,1] [,2]
## [1,]   23   24
class(x4)
## [1] "array"
```

Wir können `dim` setzen mittels `attr(, "dim")<-`, `dim()<-` oder `array()`. Den Wert von `dim` erhalten wir mit `attr(, "dim")` oder `dim()`.

Ein Array `x` mit `length(dim(x))` gleich `2` ist eine Matrix. Matrizen können wir auch mit `matrix()` erzeugen.

```
x
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
dim(x)
## [1] 2 3
dim(x) <- c(3,2)
x
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Der Wert von `dim` ist ein `integer`-Vektor der Länge ≥ 1, sodass `prod(dim(x))` (Produkt aller Elemente) gleich `length(x)` ist.

Die Einträge des zugrunde liegenden Vektors werden in einer Rechteckstruktur (bei Matrizen) bzw "Hyperrechteckstruktur" (bei Arrays) angeordnet. Für Matrizen geschieht dies in der *column-major order* (außer man setzt das Argument `byrow` der Funktion `matrix` entsprechend).

```
A <- matrix(1:9, nrow=3)
A
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
```

```
## [3,]    3    6    9
dim(A) <- NULL # entferne Dimensions-Attribut
A # Der Matrix A lag der Vektor 1:9 zugrunde
## [1] 1 2 3 4 5 6 7 8 9
B <- matrix(1:9, nrow=3, byrow=T)
B
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
dim(B) <- NULL
B # Beim Erstellen der Matrix B wurde 1:9 umgeordnet
## [1] 1 4 7 2 5 8 3 6 9
```

Arrays können sowohl auf einem atomaren Vektor (beliebigen Basis-Typs) als auch auf einer Liste aufbauen (Die mit Abstand häufigste Anwendung sind jedoch Matrizen aufbauend auf atomaren Vektoren vom Typ `double`, `integer` oder `logical`.)

```
lst <- list("a", 1, T, list())
matrix(lst, ncol=2)
##      [,1] [,2]
## [1,] "a"  TRUE
## [2,] 1    List,0
```

Die Ausgabe von `typeof()` für ein Array entspricht der Ausgabe für den zugrunde liegenden Vektor.

```
mat_types <- list(
     matrix(c(T, F, T, F), ncol=2),
     matrix(1:4, ncol=2),
     matrix(as.double(1:4), ncol=2),
     matrix(letters[1:4], ncol=2),
     matrix(as.list(1:4), ncol=2)
  )
sapply(mat_types, typeof)
## [1] "logical"   "integer"   "double"    "character" "list"
```

Nutze `is.array()`, `is.matrix()` oder `class()` um herauszufinden, ob ein Objekt eine Matrix oder ein Array ist.

```
classify_array <- function(x) c(is.atomic(x), is.list(x), is.vector(x), is.matrix(x), is.array(x))
misc <- list(1:4, matrix(c(T,T), nrow=2), matrix(lst, ncol=2), array(1:4, dim = c(2,2,1)))
res <- sapply(misc, classify_array)
rownames(res) <- paste("is.",c("atom", "list", "vect", "matr", "arry"), sep="")
colnames(res) <- sapply(misc, function(x) paste(class(x)[1], typeof(x)))
print(res)
##         integer integer matrix logical matrix list array integer
## is.atom            TRUE           TRUE       FALSE          TRUE
## is.list           FALSE          FALSE        TRUE         FALSE
## is.vect            TRUE          FALSE       FALSE         FALSE
## is.matr           FALSE           TRUE        TRUE         FALSE
## is.arry           FALSE           TRUE        TRUE          TRUE
```

Die Zeilen und Spalten (alle Dimensionen) von Matrizen (Arrays) können benannt werden. Dies geschieht mittels des Attributes `dimnames`. Der Wert von `dimnames` ist eine Liste (ggf mit benannten Werten) der selben Länge wie `dim`, deren Einträge `character`-Vektoren sind. Abkürzungen für `attr(, "dimnames")` sind `dimnames()` bzw `rownames()`, `colnames()` für die ersten beiden Einträge der Liste.

```r
x <- matrix(1:4, ncol=2)
rownames(x) <- letters[1:2]
colnames(x) <- letters[11:12]
x
##   k l
## a 1 3
## b 2 4
str(attributes(x))
## List of 2
##  $ dim     : int [1:2] 2 2
##  $ dimnames:List of 2
##   ..$ : chr [1:2] "a" "b"
##   ..$ : chr [1:2] "k" "l"
dimnames(x) <- list("Zeilen" = c("z1", "z2"), "Spalten" = c("s1", "s2"))
x
##       Spalten
## Zeilen s1 s2
##     z1  1  3
##     z2  2  4
```

Weitere nützliche Funktionen im Zusammenhang mit Matrizen und Arrays sind (siehe `?`):

- `nrow()`, `ncol()`, `NROW()`, `NCOL()`

```r
y <- cbind(1:3, 7:9)
y
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
dim(y)
## [1] 3 2
c(ncol(y), NCOL(y), nrow(y), NROW(y), length(y)) # hier sind nrow, NROW gleich
## [1] 2 2 3 3 6
z <- 1:3
nrow(z)
## NULL
ncol(z)
## NULL
c(NROW(z), NCOL(z)) # hier unterschiedlich
## [1] 3 1
```

- transponierte Matrix `t()`, "transponieren" für Arrays `aperm()`
- Matrizen und Arrays verbinden: `rbind()`, `cbind()`, `abind::abind()`

```r
x <- cbind(1:3)
x
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
x <- cbind(x , 11:13)
x
##      [,1] [,2]
## [1,]    1   11
```

```
## [2,]    2   12
## [3,]    3   13
y <- t(x)
y
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   11   12   13
abind::abind(y, y+500, along=3)
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   11   12   13
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]  501  502  503
## [2,]  511  512  513
```

- Matrixoperationen: `%*%` (Matrixmultiplikation), `det()` (Determinante), `solve()` (Lineares Gleichungssystem lösen und Matrix invertieren), `qr()` (QR-Zerlegung), `chol()` (Cholesky-Zerlegung), `svd()` (Singulärwertzerlegung), `eigen()` (Eigenwerte und -vektoren)

Bemerkung: Einige Attribute werden beim Setzen auf Korrektheit (zB richtiger Typ) getestet (`class`, `comment`, `dim`, `dimnames`, `names`, `row.names`, `tsp`; siehe `?attr`). Die meisten Attribute werden jedoch nicht geprüft.

```
x <- 1:3
attr(x, "dim") <- 4 # ERROR
## Error in attr(x, "dim") <- 4: dims [product 4] do not match the length of object [3]
attr(x, "names") <- letters # ERROR
## Error in attr(x, "names") <- letters: 'names' attribute [26] must be the same length as the vector [
```

5 S3 Objekte

Durch das Setzen des `class`-Attributs machen wir Objekte zu sogenannten **S3 Objekten**. Diese werden im Kapitel über objektorientierte Programmierung ausführlicher behandelt. Hier gibt es einen Überblick.

Beispiele für S3 Klassen sind `tibble`, `factor`, `Date`, `POSIXct` (Zeit), (gleich ausführlicher).

Einige Funktionen (`print()`, `plot()`, `summary()`) behandeln Objekte je nach Klasse unterschiedlich.

Ist `class` gesetzt, wird der Wert dieses Attributes von `class()` zurückgegeben (**explizite Klasse**).

```
attr(x, "class") <- "Spitze"
class(x)
## [1] "Spitze"
```

Ist `class` nicht gesetzt, gibt `class()` die **implizite Klasse** aus: `"matrix"`, `"array"`, `"function"`, `"numeric"` oder den Wert von `typeof(x)`.

```
type_and_class <- function(x) c(typeof(x), class(x)[1], {y <- attr(x, "class")[1];if(is.null(y))"NULL"el
misc <- c(misc_atomic, list(list(), matrix(1:4, ncol=2), array(1, dim=1), sum, mean,
                            NULL, structure(1:3, class="asdf"), factor(1:3)))
res <- sapply(misc, type_and_class)
rownames(res) <- c("typeof()", "class()", "attr class")
```

```
colnames(res) <- c("logi", "inte", "doub", "char", "comp", "raw", "list", "matr",
                   "arra", "sum", "mean", "NULL", "S3", "factor")
print(res)
##            logi      inte      doub      char         comp      raw    list
## typeof()   "logical" "integer" "double"  "character" "complex" "raw"  "list"
## class()    "logical" "integer" "numeric" "character" "complex" "raw"  "list"
## attr class "NULL"    "NULL"    "NULL"    "NULL"       "NULL"    "NULL" "NULL"
##            matr      arra     sum       mean       NULL   S3        factor
## typeof()   "integer" "double" "builtin"  "closure"  "NULL" "integer" "integer"
## class()    "matrix"  "array"  "function" "function" "NULL" "asdf"    "factor"
## attr class "NULL"    "NULL"   "NULL"     "NULL"     "NULL" "asdf"    "factor"
```

Das Klassen-Attribut kann mit `unclass()` oder `attr(, "class") <- NULL` entfernt werden.

```
str(x)
##  'Spitze' int [1:3] 1 2 3
x <- unclass(x)
str(x)
##  int [1:3] 1 2 3
```

5.1 Faktoren

Faktoren sind `integer`-Vektoren mit `attr(x ,"class")` gleich `"factor"` und dem Attribut `levels`.

Sie speichern kategorische Daten. Die Menge der möglichen Werte gibt das Attribut `levels` an.

Die Funktion `factor()` erzeugt eine Faktor. `table()` listet wie oft welcher Wert auftritt.

```
x <- c("m", "f", "f", "f", "m", "f")
x_factor <- factor(x)
x_factor
## [1] m f f f m f
## Levels: f m
str(attributes(x_factor))
## List of 2
##  $ levels: chr [1:2] "f" "m"
##  $ class : chr "factor"
table(x_factor)
## x_factor
## f m
## 4 2
unclass(x_factor)
## [1] 2 1 1 1 2 1
## attr(,"levels")
## [1] "f" "m"

# create factor "by hand":
y <- rep(1:3,3)
y
## [1] 1 2 3 1 2 3 1 2 3
is.factor(y)
## [1] FALSE
attr(y, "class") <- "factor"
attr(y, "levels") <- letters[26:24]
is.factor(y)
## [1] TRUE
```

```
y
## [1] z y x z y x z y x
## Levels: z y x
table(y)
## y
## z y x
## 3 3 3
```

Siehe auch `?factor`.

5.2 Date

Objekte der Klasse `Date` speichern Kalenderdaten.

Sie sind `double`-Vektoren, deren `class`-Attribut den Wert `"Date"` hat.

`Sys.Date()` gibt ein `Date`-Objekt zurück, das das aktuelle Datum enthält.

```
Sys.Date()
## [1] "2021-04-17"
attributes(Sys.Date())
## $class
## [1] "Date"
```

Der zugrunde liegende `double`-Wert ist die Anzahl Tage seit 1.1.1970.

```
unclass(Sys.Date())
## [1] 18734
structure(c(0,1), class="Date")
## [1] "1970-01-01" "1970-01-02"
```

5.3 POSIXct

Möchte man Datum und Uhrzeit speichern, kann man auf die Klasse `POSIXct` (*Portable Operating System Interface*, `c()`, time) zurückgreifen.

Dies ist ein `double` Vektor, dessen `class`-Attribut den Wert `c("POSIXct", "POSIXt")` hat.

Dies bedeutet, dass `POSIXct` eine Unterklasse von `POSIXt` ist (später mehr).

`Sys.time()` gibt ein `POSIXct`-Objekt zurück, das das aktuelle Datum und die Uhrzeit enthält.

```
x <- Sys.time()
x
## [1] "2021-04-17 16:47:18 CEST"
attributes(x)
## $class
## [1] "POSIXct" "POSIXt"
unclass(x)
## [1] 1618670839
```

Der zugrunde liegende `double`-Wert ist die Anzahl Sekunden seit 1.1.1970

```
unclass(Sys.time())
## [1] 1618670839
structure(c(0,1), class=c("POSIXct", "POSIXt"))
## [1] "1970-01-01 01:00:00 CET" "1970-01-01 01:00:01 CET"
```

5.4 proc_time

`proc.time()` gibt die Laufdauer des R-Prozesses als S3 Objekt der Klasse `proc_time` an.

```
x <- proc.time()
x
##    user  system elapsed
##    1.93    0.43    3.09
class(x)
## [1] "proc_time"
str(attributes(x))
## List of 2
##  $ names: chr [1:5] "user.self" "sys.self" "elapsed" "user.child" ...
##  $ class: chr "proc_time"
str(unclass(x))
##  Named num [1:5] 1.93 0.43 3.09 NA NA
##  - attr(*, "names")= chr [1:5] "user.self" "sys.self" "elapsed" "user.child" ...
```

Zugrunde liegt ein `double`-Vektor der Länge 5.

`user` gibt an, wie lange der R-Prozess die CPU beansprucht hat (in Sekunden); `system` ist die Dauer der Belastung der CPU durch das Betriebssystems im Auftrag des R-Prozesses (zB für Daten laden, speichern, Ausgabe); `elapsed` ist die gesamt vergangene Zeit.

Mit `proc_time`-Objekten kann man rechnen.

```
pt <- proc.time()
x <- 0
for (i in 1:1e7) x <- x + 1
proc.time() - pt
##    user  system elapsed
##    0.21    0.00    0.20
```

`system.time(expression)` misst die Rechenzeit zum Auswerten des Ausdrucks `expression` und gibt diese als `proc_time`-Objekt zurück.

```
tm <- system.time({
  data <- runif(1e6)
  stats <- c(mean(data), var(data))
})
class(tm)
## [1] "proc_time"
tm
##    user  system elapsed
##    0.04    0.00    0.03
```

6 Tibbles

Tibbles sind ebenfalls S3-Objekte (der Klasse `tbl_df`).

```
library(tibble)
tib1 <- tibble(x = 1:3, y = letters[3:1])
tib1
## # A tibble: 3 x 2
##       x y
##   <int> <chr>
## 1     1 c
```

```
## 2     2 b
## 3     3 a
class(tib1)
## [1] "tbl_df"     "tbl"        "data.frame"
str(attributes(tib1))
## List of 3
##  $ names    : chr [1:2] "x" "y"
##  $ row.names: int [1:3] 1 2 3
##  $ class    : chr [1:3] "tbl_df" "tbl" "data.frame"
```

Tibbles basieren auf **Data-Frames**. Data-Frames sind in Base-R ohne das Laden von Paketen verfügbar.

```
df <- data.frame(x = 1:3, y = letters[3:1])
df
##   x y
## 1 1 c
## 2 2 b
## 3 3 a
```

Data-Frames (und damit Tibbles) sind S3-Objekte, welche auf Listen aufbauen. Jeder Eintrag der Liste muss die gleiche Länge haben.

```
typeof(tib1)
## [1] "list"
str(unclass(tib1))
## List of 2
##  $ x: int [1:3] 1 2 3
##  $ y: chr [1:3] "c" "b" "a"
##  - attr(*, "row.names")= int [1:3] 1 2 3
tib1$x # wie bei Liste
## [1] 1 2 3
```

Typischerweise werden Datensätze in Tibbles geladen und ausgehend von dieser Datenstruktur ausgewertet. Jede Zeile entspricht einer Beobachtung, jede Spalte einer beobachteten Variable / einem Feature.

Mit `nrow()` erhalten wir die Anzahl der Beobachtungen, mit `ncol()` Anzahl der Variablen. `length()` entspricht `ncol()`, da ein Tibble eine Liste seiner Spalten ist und `length()` die Länge der Liste ausgibt.

```
c(ncol(tib1), nrow(tib1), length(tib1))
## [1] 2 3 2
```

Ein Tibble kann Listen als Spalten haben.

```
lst <- list(T, 1:5, "asdf", 0, list(0))
tib2 <- tibble(asdf=LETTERS[1:5], Y=lst)
tib2 # Y ist Listen-Spalte
## # A tibble: 5 x 2
##   asdf  Y
##   <chr> <list>
## 1 A     <lgl [1]>
## 2 B     <int [5]>
## 3 C     <chr [1]>
## 4 D     <dbl [1]>
## 5 E     <list [1]>
```

Ein Tibble kann Matrizen oder Tibbles als Spalten haben.

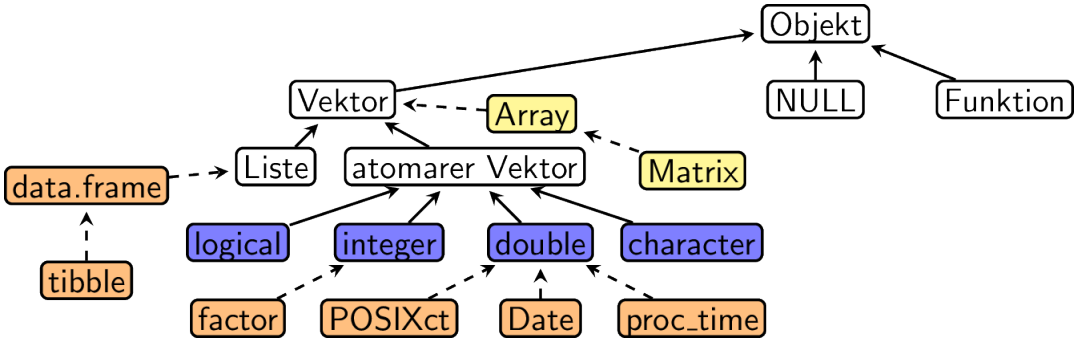
```
tib3 <- tibble(x=1:5)
tib3$y <- matrix(1:20, nrow=5)
tib3$z <- tib2
str(tib3)
## tibble [5 x 3] (S3: tbl_df/tbl/data.frame)
##  $ x: int [1:5] 1 2 3 4 5
##  $ y: int [1:5, 1:4] 1 2 3 4 5 6 7 8 9 10 ...
##  $ z: tibble [5 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ asdf: chr [1:5] "A" "B" "C" "D" ...
##   ..$ Y   :List of 5
##   .. ..$ : logi TRUE
##   .. ..$ : int [1:5] 1 2 3 4 5
##   .. ..$ : chr "asdf"
##   .. ..$ : num 0
##   .. ..$ :List of 1
##   .. .. ..$ : num 0
tib3
## # A tibble: 5 x 3
##       x y[,1]  [,2]  [,3]  [,4] z$asdf $Y
##   <int> <int> <int> <int> <int> <chr>  <list>
## 1     1     1     6    11    16 A      <lgl [1]>
## 2     2     2     7    12    17 B      <int [5]>
## 3     3     3     8    13    18 C      <chr [1]>
## 4     4     4     9    14    19 D      <dbl [1]>
## 5     5     5    10    15    20 E      <list [1]>
```

Die Bedingung für Validität eines Tibbles ist, dass alle Einträge der zugrundeliegenden Liste auf einem Vektor-Typ basieren und die Ausgabe von `NROW()` übereinstimmt.

Um Konvertierungen zwischen Matrizen und Tibbles durchzuführen, kann `as_tibble()` bzw `as.matrix()` benutzt werden.

```
x <- matrix(1:12, ncol=3, dimnames=list(NULL, LETTERS[1:3]))
x
##      A B  C
## [1,] 1 5  9
## [2,] 2 6 10
## [3,] 3 7 11
## [4,] 4 8 12
tib <- as_tibble(x)
tib
## # A tibble: 4 x 3
##       A     B     C
##   <int> <int> <int>
## 1     1     5     9
## 2     2     6    10
## 3     3     7    11
## 4     4     8    12
as.matrix(tib)
##      A B  C
## [1,] 1 5  9
## [2,] 2 6 10
## [3,] 3 7 11
## [4,] 4 8 12
```

7 Überblick Datenstrukturen



8 Numerische Verwirrung

Wir listen ein paar Dinge, die im Umgang mit numerischen Typen Schwierigkeiten bereiten können.

Beachte die unterschiedlichen Typen von `0L` vs `0`, `1:1` vs `1`, `2L*2L` vs `2L^2L`.

```
sapply(list(0L, 0, 1:1, 1, 2L*2L, 2L^2L), typeof)
## [1] "integer" "double"  "integer" "double"  "integer" "double"
```

Der Begriff `numeric` wird inkonsistent verwendet.

In den Funktionen `class()`, `as.numeric()`, `numeric(n)`, `print()`, `str()` ist `numeric` synonym zu `double`.

```
c(class(1L), class(pi))
## [1] "integer" "numeric"
typeof(numeric(3))
## [1] "double"
double(0)
## numeric(0)
str(double(3))
##  num [1:3] 0 0 0
c(typeof(numeric(3)), typeof(as.numeric(1L)))
## [1] "double" "double"
```

Die Funktion `is.numeric()` behandelt `numeric` als Überbegriff für `integer` und `double`.

```
c(is.numeric(1L), is.numeric(pi))
## [1] TRUE TRUE
```

Die *not available*-Wert vom Typ `double` heißt `NA_real_`

Welche der folgenden Ausdrücke ist wahr? `0 == 0L`, `identical(0, 0L)`, `sin(pi) == 0`

Auch `integer`-Werte leiden unter der Endlichkeit des Computers (*integer overflow*).

```
c(as.integer(2^30), as.integer(2^31), as.integer(2^30)*2L)
## Warning: NAs introduced by coercion to integer range
## Warning in as.integer(2^30) * 2L: NAs produced by integer overflow
## [1] 1073741824         NA         NA
```

9 Much Ado About Nothing

Es gibt verschiedene Objekte in R, die auf das Fehlen eines Wertes hinweisen.

`NULL` ist Rückgabewert von `c()` und hat Länge `0` und Typ `NULL`.

```
c()
## NULL
c(typeof(NULL), length(NULL))
## [1] "NULL" "0"
```

`BASE_TYPE(0)` ergibt einen atomaren Vektor der Länge 0 vom Typ `BASE_TYPE`.

```
c(typeof(integer(0)), length(integer(0)))
## [1] "integer" "0"
identical(NULL, integer(0))
## [1] FALSE
```

Die Werte `NA`, `NA_integer_`, `NA_real_`, `NA_character_` haben Länge 1 und den entsprechenden Basistyp.

```
c(length(NA), typeof(NA_integer_))
## [1] "1"       "integer"
```

`matrix(nrow=0, ncol=0)` und `array(dim=c(0))` haben Länge 0 und einen Basistyp (`logical`), aber im Unterschied zu `logical(0)` das Attribut `dim`.

```
m <- matrix(nrow=0, ncol=0)
c(typeof(m), length(m))
## [1] "logical" "0"
attributes(m)
## $dim
## [1] 0 0
identical(m, logical(0))
## [1] FALSE
```

Für einen atomaren Vektor sind `x[0]`, `x[NULL]` gleich `BASE_TYPE(0)`, wohingegen `x[]` dem ganzen Vektor `x` entspricht. `x[NA_integer_]`, `x[NA]` geben Vektoren mit `NA`-Werten zurück.

```
x <- 1:3
x[0]
## integer(0)
x[NULL]
## integer(0)
x[]
## [1] 1 2 3
identical(x, x[])
## [1] TRUE
x[NA_integer_]
## [1] NA
x[NA]
## [1] NA NA NA
```

`NaN` verhält sich größtenteils wie `NA_real_`. Die beiden Werte sind jedoch nicht identisch.

```
c(typeof(NaN), typeof(NA_real_))
## [1] "double" "double"
c(length(NaN), length(NA_real_))
## [1] 1 1
identical(NA_real_, NaN)
## [1] FALSE
```

Teste fehlende Werte mit `is.null()`, `is.nan()` oder `is.na()`.

```r
classify_missing <- function(x) c(is.null(x), is.na(x), is.nan(x))
misc <- list(NA, NaN, 0, NA_character_, "")
res <- sapply(misc, classify_missing)
rownames(res) <- paste("is.",c("null", "na", "nan"), sep="")
colnames(res) <- c("NA", "NaN", "0", "NA_character_", "\"\"")
print(res)
##            NA   NaN     0 NA_character_     ""
## is.null FALSE FALSE FALSE         FALSE FALSE
## is.na    TRUE  TRUE FALSE          TRUE FALSE
## is.nan  FALSE  TRUE FALSE         FALSE FALSE
is.null(NULL)
## [1] TRUE
is.nan(NULL)
## logical(0)
is.na(NULL)
## logical(0)
is.null(logical(0))
## [1] FALSE
is.nan(logical(0))
## logical(0)
is.na(logical(0))
## logical(0)
x <- c(1, NA, 3, NaN, 5)
is.null(x)
## [1] FALSE
is.nan(x)
## [1] FALSE FALSE FALSE  TRUE FALSE
is.na(x)
## [1] FALSE  TRUE FALSE  TRUE FALSE
```

Gleichheit: Welche Werte haben folgende Ausdrücke? `1:3 == NA`, `identical(NA, NA)`, `NULL == NULL`, `identical(NULL, NULL)`

```r
1:3 == NA
## [1] NA NA NA
NA == NA
## [1] NA
identical(NA, NA)
## [1] TRUE
NULL == NULL
## logical(0)
identical(NULL, NULL)
## [1] TRUE
```

Die vier häufigen Basistypen haben Default-Werte (`FALSE`, `0L`, `0`, `""`). Default-Werte sind nicht nichts. Insbesondere ist `length("")` gleich 1, aber `nchar("")` gleich `0` und `nchar(NA_character_)` ist `NA_integer_`.

```r
c(length(""), nchar(""), length(NA_character_), nchar(NA_character_))
## [1]  1  0  1 NA
typeof(nchar(NA_character_))
## [1] "integer"
```