# V04 – Funktionen

26. April 2021

## Contents



# 1 Intro

> "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 etwas tut, ist ein *Funktionsaufruf*. Funktionen selbst sind *Objekte*.

Funktionen in R können

- anderen Funktionen als Argument dienen,
- Rückgabewert einer Funktion sein,
- in einer Datenstruktur (Liste) gespeichert werden

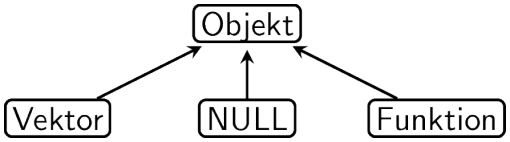
Damit sind sie sogenannte **First-Class-Funktionen**.

```
# Funktion als Argument
apply(matrix(1:4, nrow=2), 2, sum)  # Zeilensumme
## [1] 3 7

# Funktion als Rückgabewert
create_fun <- function (x) {
```

```
  fun <- function() {
    print(x)
  }
  return(fun)
}
f <- create_fun("blub")
f()
## [1] "blub"

# Funktion in Liste
funs <- list(
  half = function(x) x / 2,
  twice = function(x) x * 2
)
funs$twice(10)
## [1] 20
```




Mit `function(arg_list) expression` können wir ein Funktions-Objekt erzeugen.

Damit wir die Funktion später aufrufen können, weisen wir ihr typischerweise einen Variablennamen zu: `function_name <- function(arg_list) expression`.

Wird kein Name zugewiesen sprechen wir von **anonymen Funktionen**.

```
head(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
sapply(mtcars, function(x) length(unique(x))) # anonyme Funktion
##  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
##   25    3   27   22   22   29   30    2    2    3    6
```

Für `function_name` gelten die selben Regeln für mögliche Namen wie bei anderen Objekten.

Erinnerung: Gültige Namen für Variablen bestehen aus Buchstaben, Ziffern, `.`, oder `_` und beginnen mit einem Buchstaben oder dem Punkt nicht gefolgt von einer Ziffern und sind keine **reservierte Wörter**.

Weitere gültige Namen sind beliebige Zeichenketten in Backticks (“rückwärts geneigtes Hochkomma”, `` ` ``).

```
`a b` <- 1
# a b # ERROR
`a b`
## [1] 1
# § <- 2 # ERROR
`§` <- 2
# § # ERROR
`§`
```

```
## [1] 2
`x` <- 42 # ist das Gleiche wie x
x
## [1] 42
```

Diese Regeln, inklusive Backticks, gelten auch für die Namen von Funktionsargumenten.

```
f <- function(`_`, `***`, `42`) {
  c(`_`, `***`, `42`)
}
f(1,2,3)
## [1] 1 2 3
```

Beim Funktionsaufruf können statt Backticks auch Anführungsszeichen genutzt werden.

```
f(`***`=1, '42'=2, "_"=3)
## [1] 3 1 2
c(`***`=1, '42'=2, "_"=3)
##  ***  42   _
##    1   2   3
```

## 2 Formen des Funktionsaufrufs

Schreiben wir keine Klammern hinter eine Funktion, greifen wir auf das Funktionsobjekt zu statt die Funktion aufzurufen. `print()` gibt für eine Funktion ihren R-Code aus.

```
f <- function(x,y) x+y
f # auf oberster Ebene gleich print(f)
## function(x,y) x+y
```

Bekannt ist der Funktionsaufruf mit `()`, notiert als `function_name(arg_list)`.

Da Operatoren wie `+ / %% %*% & | < : <-` oder Konstrukte wie `if`, `while`, `for`, `[[` etwas tun, sind sie nach obigem Zitat auch Funktionen. Ihr Aufruf hat jedoch eine besondere Form.

Funktionsaufrufe können verschiedene Formen haben:

- **Präfixform**: Funktionsname vor den Argumenten, `f(x,y,z)`
- **Infixform**: Funktionsname zwischen den Argumenten `x+y`
- **Ersetzungsfunktion** (*replacement function*): `names(z) <- c("a", "b", "c")`
- **Spezialformen** ohne konsistente Struktur, zB: `[[`, `if`, `for`

Um auf das Funktionsobjekt einer Funktion zuzugreifen, die nicht in Präfixform aufgerufen wird, verwenden wir Backticks `` ` ``. Also etwa `` `+` `` oder `` `[` ``. Mit `?`+`` gelangen wir etwa zur Hilfe für arithmetische Operatoren.

```
sapply(c(T,F,T), `!`)
## [1] FALSE  TRUE FALSE
```

Alle Funktionen können in der Präfixform aufgerufen werden.

```
x + y
`+`(x, y)

names(z) <- c("x", "y", "z")
`names<-`(z, c("x", "y", "z"))

for(i in 1:10) print(i)
`for`(i, 1:10, print(i))
```

```r
`+`(1, 2) # 1 + 2
## [1] 3
`<-`(x, 11:15) # x <- 11:15
`[`(x, 2:4) # x[2:4]
## [1] 12 13 14
```

## 2.1 Präfixform

Die Zuordnung von übergebenen Werten zu Argumenten einer Funktion kann durch Position oder Argumentname bestimmt werden. Es ist möglich nur Teile des Namens anzugeben und R findet automatisch den vollständigen Namen (**partial matching**).

```r
f <- function(abc, bcd1, bcd2) {
  list(a = abc, b1 = bcd1, b2 = bcd2)
}
str(f(1, 2, 3))
## List of 3
##  $ a : num 1
##  $ b1: num 2
##  $ b2: num 3
str(f(2, 3, abc = 1))
## List of 3
##  $ a : num 1
##  $ b1: num 2
##  $ b2: num 3
str(f(2, 3, a = 1))
## List of 3
##  $ a : num 1
##  $ b1: num 2
##  $ b2: num 3
str(f(1, 3, b = 1))
## Error in f(1, 3, b = 1): argument 3 matches multiple formal arguments

seq(0, 1, len=4) # vollständiger Name "length.out"
## [1] 0.0000000 0.3333333 0.6666667 1.0000000
```

Guter Stil: Nutze nur für die ersten ein bis zwei Argumente die Position, sonst die Namen. Verzichte auf *partial matching*.

## 2.2 Infix Funktionen

Wird eine Funktion mit dem Namen `%op_name%` definiert, kann statt `` `%op_name%`(x,y) `` auch `x %op_name% y` geschrieben werden.

```r
`%asdf%` <- function(x, y) 100*x+y
`%asdf%`(5,7)
## [1] 507
5 %asdf% 7
## [1] 507
`%"%` <- function(x, y) paste(x,y)
`%"%`(3, "Äpfel")
## [1] "3 Äpfel"
3 %"% "Äpfel" %"% T
## [1] "3 Äpfel TRUE"
```

Operatoren anderer Form können nicht definiert werden. Jedoch kann man vorhandene Operatoren mit neuen Werten belegen. Dies sollte jedoch vermieden werden, um keine allzu große Verwirrung zu stiften.

```
`+` <- function(x,y) x-y # evil
3+4
## [1] -1
rm(`+`) # undo
3+4
## [1] 7
```

Eine Liste von vordefinierten Infix-Funktionen (Operatoren) kann mit `?Syntax` abgerufen werden.

Bemerkung: Es gibt den Operator `:=`, der jedoch in Base-R nicht genutzt wird. Der Operator `**` ist synonym zu `^`.

In `?Syntax` wird auch die Operator-Priorität (Ausführungsreihenfolge) angezeigt.

Bei gleicher Priorität wird von links nach rechts ausgeführt mit wenigen Ausnahmen wie etwa `^`.

```
`%-%` <- function(a, b) paste0("(", a, " %-% ", b, ")")
"a" %-% "b" %-% "c"
## [1] "((a %-% b) %-% c)"
2^2^3 == 2^(2^3)
## [1] TRUE
```

Achtung: Die Operatorpriorität des Sequenz-Operators `:` ist niedriger als `^` und Vorzeichen (unärem `+`, `-`) aber höher als `*`, `/` und (binären) `+`, `-`.

```
x <- 1:6
x[2:length(x)-1] # möglicherweise nicht das, was man will
## [1] 1 2 3 4 5
x[2:(length(x)-1)]
## [1] 2 3 4 5
```

Guter Stil: Leerzeichen um alle Operatoren mit strikt niedrigere Priorität als `:`, also etwa `x[1:length(x) - 1]`.

## 2.3 Ersetzungsfunktionen

Ersetzungsfunktionen sind eine der wenigen Funktionen, die ihre Argumente verändern.

Sie haben einen Namen der Form `function_name<-` und müssen Argumente mit den Namen `x` und `value` haben.

Der Aufruf einer solchen Funktion folgt dem Muster `function_name(x) <- value`. Dabei wird nicht nur die Funktion aufgerufen, sondern deren Resultat auch automatisch `x` zugewiesen.

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:5
second(x) <- 10L
x
## [1]  1 10  3  4  5
`second<-`(x, 100L)
## [1]   1 100   3   4   5
x # call by value
## [1]  1 10  3  4  5
```

```r
x <- `second<-`(x, 100L) # dies entrpricht second(x) <- 100L
x
## [1]   1 100   3   4   5
```

Würde `function_name(x) <- value` "normal" ausgeführt werden, ergäbe dieser Befehl keinen Sinn, da `function_name(x)` kein gültiger Variablenname ist.

```r
f <- function(x) x+5
f(x) <- 2 # ERROR
## Error in f(x) <- 2: could not find function "f<-"
```

Zusätzliche Argumente werden zwischen `x` und `value` gelistet.

```r
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- -10
x
## [1] -10 100   3   4   5
x <- `modify<-`(x, 2, -10)
x
## [1] -10 -10   3   4   5
```

Ersetzungsfunktionen können geschachtelt werden. Dabei spielen sowohl `` `function_name<-`() `` als auch `function_name()` eine Rolle. Die Übersetzung in Präfix-Notation wird ggf komplexer.

```r
x <- 1:3
names(x) <- LETTERS[1:3]
x
## A B C
## 1 2 3
x <- `names<-`(x, letters[1:3]) # in Präfixnotation:
x
## a b c
## 1 2 3

x[2] <- 12
x
##  a  b  c
##  1 12  3
x <- `[<-`(x, 2, 22) # in Präfixnotation:
x
##  a  b  c
##  1 22  3

names(x)[2] <- "two"
names(x)
## [1] "a"   "two" "c"
# in Präfixnotation:
`*tmp*` <- x # erstelle Variable *tmp*
x <- `names<-`(`*tmp*`, `[<-`(names(`*tmp*`), 2, "two"))
rm(`*tmp*`) # entferne Variable *tmp*
```

```r
# allgemein
outside(inside(x, args_in), args_out) <- value
```

```r
# in Präfixnotation:
`*tmp*` <- x
x <- `inside<-`(
    `*tmp*`,
    args_in,
    `outside<-`(
        inside(`*tmp*`, args_in),
        args_out,
        value
    )
)
rm(`*tmp*`)
```

## 2.4 Spezialformen

Alle Spezialformen haben auch eine Präfixform.

```r
`(`(x) # (x)
`{`(x) # {x}
`[`(x, i) # x[i]
`[[`(x, i) # x[[i]]
`if`(cond, true) # if (cond) true
`if`(cond, true, false) # if (cond) true else false
`for`(var, seq, action) # for(var in seq) action
`while`(cond, action) # while(cond) action
`repeat`(expr) # repeat expr
`next`() # next
`break`() # break
`function`(alist(arg1, arg2), body, env) # function(arg1, arg2) body
```

Ist die Präfixform einer Funktion bekannt, kann damit die Dokumentation aufgerufen werden. `?(` ergibt einen Fehler. `?`(`` ruft die Dokumentation auf.

Alle Funktionen mit Spezialform sind sogenannte primitive Funktionen, dazu später mehr.

```r
`for`
## .Primitive("for")
```

## 2.5 Funktionsaufruf mit Argumentliste

Um Argumente in einer Liste an eine Funktion zu übergeben, nutze `do.call()`.

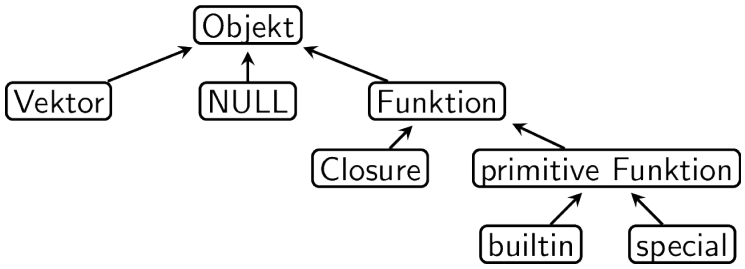
```r
x <- c(1 ,5, NA, 7)
args <- list(x, na.rm = TRUE)
do.call(mean, args)
## [1] 4.333333
# entspricht
mean(x, na.rm = TRUE)
## [1] 4.333333
```

# 3 Klassifizierung von Funktionen

Die Klasse eines Funktions-Objektes ist `function`. Der Typ ist `closure`, `builtin` oder `special`.

```r
c(class(mean), class(sum), class(`[`))
## [1] "function" "function" "function"
c(typeof(mean), typeof(sum), typeof(`[`))
## [1] "closure" "builtin" "special"
```

Funktionen vom Typ `builtin` oder `special` sind sogenannte **primitive Funktionen**. Sie existieren ausschließlich in Base-R. Es gibt nur rund 200 davon. Alle anderen Funktionen, insbesondere alle mit `function` erstellten Funktionen, sind Closures.




# 4 Komponenten einer Funktion

Closures bestehen aus drei Teilen: `formals()` (Argumente), `body()` und `environment()` (Umgebung). Der Funktionsname zählt nicht als Teil der Funktion.

```r
f02 <- function(x) {
  # A comment
  x ^ 2
}
formals(f02)
## $x
body(f02)
## {
##     x^2
## }
environment(f02)
## <environment: R_GlobalEnv>
```

Die Funktionsumgebung (`environment()`) gibt an, wo die Funktion erstellt wurde. Dies ist wichtig dafür, wie Variablennamen ausgewertet werden und wird im Kapitel über Umgebungen ausführlich besprochen.

```r
environment(mean)
## <environment: namespace:base>
```

Wie andere Objekte können auch Funktionen Attribute haben. Das Attribut `srcref` (*source reference*) gibt den Code an, mit dem die Funktion erstellt wurde. Dabei werden (im Gegensatz zu `body()`) auch Kommentare angegeben.

```r
attr(f02, "srcref")
## function(x) {
##   # A comment
##   x ^ 2
## }
```

Primitive Funktionen haben keine `formals()`, `body()` oder `environment()`, da sie direkt kompilierten C-Code aufrufen.

```r
typeof(sum) # primitive Funktion
## [1] "builtin"
formals(sum)
## NULL
body(sum)
## NULL
environment(sum)
## NULL
```

Um auf die Argument-Liste primitiver Funktionen zuzugreifen, nutze `args()`.

```r
formals(args(sum))
## $...
##
##
## $na.rm
## [1] FALSE
```

# 5 Funktionsargumente

## 5.1 Lazy Evaluation 1

Argumente von Funktionen werden erst dann ausgewertet, wenn darauf zugegriffen wird. Dies wird als **lazy evaluation** bezeichnet.

```r
stop("This is an error!") # erzeugt ERROR
## Error in eval(expr, envir, enclos): This is an error!
f <- function(x) {
  10
}
f(stop("This is an error!")) # kein Error, da x nicht ausgewertet
## [1] 10
```

Auch die logischen Operatoren (vektorisiert: `|`, `&`. skalar: `||`, `&&`) sind Funktionen. Ausnutzen lässt sich *lazy evaluation* nur für die skalaren Operatoren.

```r
TRUE | stop("!")
## Error in eval(expr, envir, enclos): !
FALSE | stop("!")
## Error in eval(expr, envir, enclos): !
TRUE || stop("!")
## [1] TRUE
FALSE || stop("!")
## Error in eval(expr, envir, enclos): !
```

## 5.2 Default arguments

*Lazy evaluation* ermöglicht es Default-Argumente in Abhängigkeit von anderen Argumenten oder später definierten Variablen zu schreiben. Allerdings führt dies zu schwer verständlichem Code.

```r
f <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100
  c(x, y, z)
}
```

```r
f()
## [1]   1   2 110
```

Default-Argumente werden im Inneren der Funktion ausgewertet. Vom Benutzer übergebene Argumente werden in der äußeren Umgebung ausgewertet.

```r
# ls() gibt die in der aktuellen Umgebung definierten Variablennamen aus
f <- function(x = ls()) {
  a <- 1
  x
}
f()
## [1] "a" "x"
f(ls())
## [1] "f"
```

## 5.3 Missing arguments

Die Funktion `missing()` gibt an, ob ein Argument beim Funktionsaufruf übergeben wurde. Insbesondere lässt sich unterscheiden, ob ein Argument vom Nutzer kommt oder der Default-Wert ist.

```r
f <- function(x, y = 10) {
  list(misssing_x=missing(x), misssing_y=missing(y), y=y)
  # missing(z) # ERROR
}
str(f())
## List of 3
##  $ misssing_x: logi TRUE
##  $ misssing_y: logi TRUE
##  $ y         : num 10
str(f(5))
## List of 3
##  $ misssing_x: logi FALSE
##  $ misssing_y: logi TRUE
##  $ y         : num 10
str(f(y=10))
## List of 3
##  $ misssing_x: logi TRUE
##  $ misssing_y: logi FALSE
##  $ y         : num 10
str(f(5,15))
## List of 3
##  $ misssing_x: logi FALSE
##  $ misssing_y: logi FALSE
##  $ y         : num 15
```

Guter Stil: Ist ein Argument optional, sollte es einen Default-Wert haben. Dies ist leider selbst in Base-R nicht immer umgesetzt.

```r
print(sample) # zeige R-Code der Funktion sample()
## function (x, size, replace = FALSE, prob = NULL)
## {
##     if (length(x) == 1L && is.numeric(x) && is.finite(x) && x >=
##         1) {
##         if (missing(size))
```

```
##             size <- x
##         sample.int(x, size, replace, prob)
##     }
##     else {
##         if (missing(size))
##             size <- length(x)
##         x[sample.int(length(x), size, replace, prob)]
##     }
## }
## <bytecode: 0x00000000120ed138>
## <environment: namespace:base>
args(sample) # x und size haben keine Default Werte
## function (x, size, replace = FALSE, prob = NULL)
## NULL
sample(1:5) # size wegzulassen ist jedoch möglich
## [1] 5 3 4 1 2
sample <- function(x, size = NULL, replace = FALSE, prob = NULL) { # besserer Stil
  if (is.null(size)) size <- length(x)
  x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

## 5.4 dot-dot-dot

Funktionen mit dem Sonderargument `...` können beliebige zusätzliche Argumente übergeben werden.

Mittels `..n` wird auf das $n$-te Zusatzargument zugegriffen.

```
f <- function(...) {
  list(first = ..1, third = ..3)
}
str(f(1, 2, 3))
## List of 2
##  $ first: num 1
##  $ third: num 3
```

Im Inneren der Funktion können die Zusatzargumente mit `...` weitergegeben werden.

```
f <- function(y, z) {
  list(y = y, z = z)
}
g <- function(x, ...) {
  f(...)
}
str(g(x = 1, y = 2, z = 3))
## List of 2
##  $ y: num 2
##  $ z: num 3
```

Mit `list(...)` werden die Zusatzargumente in einer Liste gespeichert und können dann (ggf mit Namen) aufgerufen werden.

```
f <- function(...) {
  list(...)
}
str(f(a = 1, b = 2))
## List of 2
```

```
##  $ a: num 1
##  $ b: num 2
```

Bei Funktionen der `apply()`-Familie kann wegen `...` der übergebenen Funktion weitere Parameter mitgegeben werden.

```
x <- list(c(1, 3, NA), c(4, NA, 6))
sapply(x, mean, na.rm = TRUE)
## [1] 2 5
```

# 6 Beenden einer Funktion

Funktionsaufrufe können auf zwei Arten beendet werden: durch die Rückgabe eines Wertes oder durch ein Fehlerausgabe.

Rückgabe eines Wertes kann implizit geschehen (zuletzt ausgewerteter Ausdruck) oder explizit durch `return()`.

```
f <- function(x) {
  0
  if (x < 0) -1 else 1
}
f(-5)
## [1] -1
f(5)
## [1] 1
g <- function(x) {
  return(0)
  if (x < 0) return(-1) else return(1)
}
g(1)
## [1] 0
```

Normalerweise werden zurückgegebene Werte von Funktionsaufrufen auf der obersten Ebene (in der Konsole, nicht im Inneren von anderen Funktionen) ausgegeben. Dies kann jedoch mit `invisible()` verhindert werden.

```
f <- function() 1
f()
## [1] 1
g <- function() invisible(1)
g() # keine Ausgabe
print(g()) # Ausgabe erzwingen
## [1] 1
```

Die Funktion `` `(` `` entspricht der Identität `function(x) x`. Mit ihr wird `invisible()` aufgehoben.

```
(42)
## [1] 42
g()
(g())
## [1] 1
```

Alle Funktionen - auch die Sonderformen - haben einen Rückgabewert. ZT wird dieser unsichtbar zurückgegeben.

Falls spezielle Rückgabewerte nicht sinnvoll sind, wird meist `invisible(NULL)` zurückgegeben.

```r
if (FALSE) 42
(if (FALSE) 42)
## NULL
```

Die Funktion `<-` gibt ihr rechtes Argument unsichtbar zurück. Dies ermöglicht die Verkettung von Zuweisungen.

```r
x <- 2
(x <- 2)
## [1] 2
x <- y <- z <- 3
c(x, y, z)
## [1] 3 3 3
```

Die Funktion `stop()` erzeugt eine Fehlerausgaben. Dabei wird die Funktion sofort beendet.

```r
f <- function() {
  stop("I'm an error")
  print("HALLO!")
  return(10)
}
f()
## Error in f(): I'm an error
```

Nutze `on.exit()`, um Code bei jeglichem Verlassen der Funktion (mit Rückgabewert oder Error) auszuführen.

```r
f <- function(x) {
  print("Hello")
  on.exit(print("Goodbye!"), add = TRUE)
  print("so...")
  if (x) return(10) else stop("Error")
  print("blub")
}
f(TRUE)
## [1] "Hello"
## [1] "so..."
## [1] "Goodbye!"
## [1] 10
f(FALSE)
## [1] "Hello"
## [1] "so..."
## Error in f(FALSE): Error
## [1] "Goodbye!"
```