

Fortgeschrittene Programmierung in R

Schleifen

for-Schleife

for-Schleifen führen den Schleifenkörper (*body*) eine vorgegebene Anzahl von Malen aus, gesteuert durch einen Zähler oder Index, der in jedem Durchlauf erhöht wird:

```
for (Variable in Sequenz) {  
  Anweisung  
}
```

Beispiel

Zähler (Index)

```
for (i in c(2, 4, 7)) {  
  print(i)  
}
```

Zählerwerte

Schleifenkörper: wiederholte Ausführung des Anweisungsblocks, jedes Mal mit einem anderen *i* Wert.

Nach der Schleife sollte der letzte Wert von *i* 7 sein.

Nützliche Funktion zum Erzeugen einer Sequenz:

`seq_along(x)` → ist *x* NULL, wird `integer(0)` zurückgegeben und es erfolgt kein Durchlauf (anders als bei `1:length(x)`, das über 1 und 0 iteriert).

while-Schleife

while-Schleifen basieren auf einer logischen Bedingung, die zu Beginn jedes Durchlaufs geprüft wird. (**repeat**-Schleifen haben keine Bedingung und laufen, bis `break` aufgerufen wird.)

```
while (Bedingung) {  
  Anweisung  
}
```

Beispiel

```
i <- 3  
while (i < 10) {  
  print(i)  
  i <- i + 1  
}
```

Nach der Schleife sollte der letzte Wert von *i* 10 sein.

Bedingungen

if-Anweisung

```
if (Bedingung) {  
  Anweisung  
}
```

Beispiel

```
x <- sample(1:10,1)  
if (x < 6) {  
  print("true")  
}
```

Ist *x* größer oder gleich 6, passiert bei dieser Anweisung nichts. Die Bedingung muss die Länge 1 haben (andernfalls Fehler seit R 4.2.0); für Vektoren `ifelse()` verwenden.

if..else-Anweisung

```
if (Bedingung) {  
  Anweisung 1  
} else {  
  Anweisung 2  
}
```

Beispiel

```
x <- sample(1:10,1)  
if (x < 6) {  
  print("true")  
} else {  
  x^2  
}
```

ifelse-Anweisung

```
ifelse(Bedingung,  
       Wert 1, Wert 2)
```

Beispiel

```
x <- sample(1:10, 5)  
ifelse(x < 6,  
       "low", "high")
```

Vektorisierte Variante: gibt einen Vektor mit der Länge der Bedingung zurück.

Funktionen

Bestandteile einer Funktion

Selbst definierte Funktionen (sog. *Closures*) bestehen aus 3 Komponenten:

- den **formals** = der Argumentliste
- dem **body** = dem Code innerhalb der Funktion
- dem **environment** = der „Karte“ mit den Speicherorten der Variablen der Funktion

Der Funktionsname, mit der die Anweisung im Funktionskörper aufgerufen wird.

Die Argumente: bestimmen, wie die Anweisungen ausgeführt werden.

Standardwerte der Argumente, falls der user keine anderen Werte (Parameter) angibt.

```
roll_dice <- function(roll = 2) {  
  x <- sample(1:6, size = roll, replace = TRUE)  
  sum(x)  
}
```

Die letzte Codezeile: Wenn nicht mit `return()` explizit angegeben wird, was ausgegeben werden soll, wird die letzte Zeile ausgegeben.

Der Funktionskörper: R führt diesen Codeblock aus, wann immer die Funktion aufgerufen wird.

Beispiel für eine Funktion

Beispiel

```
std_error <- function(x) {
  val <- sd(x) / sqrt(length(x))
  return(val)
}
std_error(1:10)
[1] 0.9574271
```

Überprüfen

```
formals(std_error)
$x
body(std_error)
{
  val <- sd(x)/sqrt(length(x))
  return(val)
}
environment(std_error)
<environment: R_GlobalEnv>
```

Funktionale Programmierung

Nützliche Base-Funktionen

Funktion	Beschreibung	Beispiel
<code>do.call(what, args)</code>	Führt einen Funktionsaufruf aus, ausgehend von einem Namen oder einer Funktion und einer Liste der zu übergebenden Argumente.	Data Frames einer Liste zeilenweise verbinden: <code>xlist <- split(iris, iris\$Species)</code> <code>do.call(rbind, xlist)</code>
<code>with(data, expr)</code>	Wertet einen R-Ausdruck in einer aus den Daten aufgebauten Umgebung aus.	<code>with(iris, list(summary(aov(Sepal.Length~Species)), summary(aov(Sepal.Width~Species))))</code>

Die *apply*-Familie

Funktion auf die Dimensionen (margins) zwei- oder dreidimensionaler Objekte anwenden

Funktion	Beschreibung	Beispiel
<code>apply(X, MARGIN, FUN, ...)</code>	<i>MARGIN</i> ist ein Vektor von Indizes, über die die Funktion angewendet wird. Ist <i>X</i> ein Data Frame/eine Matrix: 1 = zeilenweise, 2 = spaltenweise und <code>c(1,2)</code> = beides. <code>..</code> steht für weitere Funktionsargumente. Gibt einen Vektor, ein Array oder eine Liste zurück.	<code>apply(iris[, 1:4], 2, mean, na.rm=TRUE)</code>

Funktion auf jedes Element eines Vektors oder einer Liste *X* anwenden:

Funktion	Beschreibung	Beispiel
<code>lapply(X, FUN, ...)</code> <code>sapply(X, FUN, ..., simplify)</code>	Gibt eine Liste derselben Länge wie <i>X</i> zurück. Wrapper für <code>lapply()</code> , gibt aber einen Vektor zurück (bei <code>simplify=TRUE</code>) → problematisch, wenn die Funktion eine unterschiedliche Anzahl von Werten liefert!	<code>lapply(as.list(iris), class)</code> <code>sapply(as.list(iris), class)</code>
<code>vapply(X, FUN, FUN.VALUE)</code>	Ähnlich wie <code>sapply()</code> , das Ausgabeformat muss aber (mit <code>FUN.VALUE</code>) angegeben werden.	<code>vapply(as.list(iris), class, FUN.VALUE = character(1))</code>
<code>tapply(X, INDEX, FUN, ...)</code>	Wendet die Funktion auf Gruppen von Werten in <i>X</i> an, die durch eine eindeutige Kombination einer Liste von Faktoren in <i>INDEX</i> festgelegt sind.	<code>tapply(iris\$Sepal.Length, iris\$Species, mean)</code>
<code>replicate(n, expr)</code>	Wrapper für <code>sapply()</code> zur wiederholten Auswertung eines Ausdrucks.	<code>hist(replicate(20, mean(rnorm(5))))</code>
<code>mapply(FUN, ..., MoreArgs)</code>	Multivariate Version von <code>sapply()</code> : wendet die Funktion auf die jeweils ersten Elemente aller <code>...</code> -Argumente an, dann auf die zweiten usw.	<code>mapply(rep, 1:4, 4:1)</code>

Von der Schleife zur funktionalen Programmierung

```
set.seed(1)
df <- data.frame(
  x = rnorm(20),
  y = rnorm(20),
  z = rnorm(20)
)
```

Stellen Sie sich vor, Sie wollen den Mittelwert jeder Spalte berechnen. Das könnten Sie mit einer for-Schleife tun:

```
output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}
output
[1] 0.190523876 -0.006471519 0.138796773
```

Wenn Sie diese Iteration häufiger für viele Spalten durchführen wollen, ist eine Funktion vielleicht praktischer:

```
col_mean <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
```

Stellen Sie sich nun vor, Sie wollen dies für viele andere Statistiken tun, z. B. Median, Stabw. → dann müssen Sie viel kopieren und einfügen!

```
col_median <- function(df) {... }
col_sd <- function(df) {... }
```

LÖSUNG: Verallgemeinern Sie Ihre Funktion und fügen Sie die Funktion für die Statistik als Argument 'fun' ein:

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, mean)
[1] 0.190523876 -0.006471519 0.138796773
col_summary(df, median)
[1] 0.35967550 -0.05496689 0.11438674
```

Ein Tidyverse-Werkzeugkasten: purrr

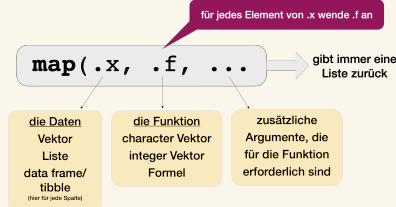


library(purrr) oder library(tidyverse)

Funktionen mit der map-Familie anwenden

Map-Funktionen arbeiten ähnlich wie lapply & Co, sind aber konsistenter und leichter zu erlernen.

Die grundlegendste Funktion



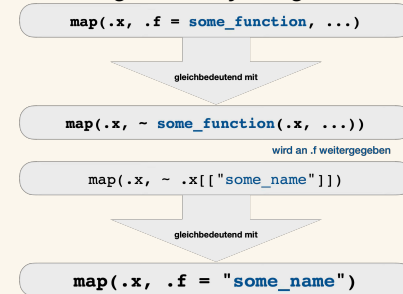
Andere Ausgabe als Liste

map_lgl() → logischer Vektor
 map_int() → Integer-Vektor
 map_dbl() → Double-Vektor
 map_chr() → Zeichenvektor
 walk() → löst Nebeneffekte aus, gibt die Eingabe unsichtbar zurück
 Data Frames: das Ergebnis von map() an list_rbind() (zeilenweise) oder list_cbind() (spaltenweise) übergeben.

Beispiele

```
# Verschiedene Angaben für .f
map(.x=1:10, .f=rnorm, n=10)
map(.x=1:10, ~rnorm(10, .x))
# Anonyme Funktion
map(1:10, .f=function(x) rnorm(10,x))
map(1:10, \(x) rnorm(10, x))
# .x hier ein Data Frame
map(iris[,1:4], mean, na.rm=TRUE)
# Zwei map-Funktionen kombinieren
map(1:10, rnorm, n=10) |>
  map_dbl(mean)
```

Möglichkeiten, f anzugeben

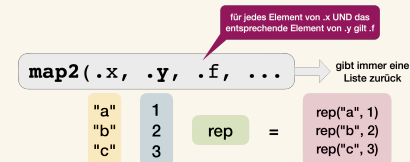


Funktion bedingt auf Elemente anwenden

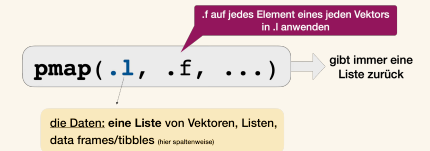
map_if(.x, .p, .f, ..., else) Nutzt eine Prädikatsfunktion .p, um zu bestimmen, welche Elemente von .x mit .f transformiert werden. Beispiel: map_if(iris, is.factor, as.character, .else=as.integer)
 map_at(.x, .at, .f, ...) Nutzt einen Vektor von Namen oder Positionen .at, um festzulegen, welche Elemente von .x mit .f transformiert werden. Beispiel: iris |> map_at(c(4, 5), as.character)
 map_depth(.x, .depth, .f, ...) Wendet .f auf eine bestimmte Tiefenebene eines verschachtelten Vektors an.

Über 2 oder mehr Argumente iterieren: map2() und pmap()

2 Argumente



>2 Argumente



- Für eine andere Ausgabe als eine Liste z. B. map2_lgl(), map2_dbl(), map2_chr(), walk2() oder pmap_lgl(), pmap_chr() usw. verwenden.
- Weitere Map-Funktionen
 - lmap(), imap()

Nützliche Funktionen für die Arbeit mit Listen

Die folgenden Beispiele beziehen sich auf diese Dummy-Liste:

```
x <- list(a=list("x", 1:5), b=NULL, c=NA, d=list(), e=list(NULL, 10:1))
```

Funktion	Beschreibung	Beispiel
Liste filtern		
keep(.x, .p, ...)	Wählt Elemente der obersten Ebene aus, die einen logischen Test bestehen.	keep(x, ~length(.x) > 1)
discard(.x, .p, ...)	Wählt Elemente der obersten Ebene aus, die einen logischen Test NICHT bestehen.	discard(x, ~length(.x) > 1)
compact(.x)	Entfernt Elemente der obersten Ebene, die NULL sind oder die Länge null haben.	compact(x)
Liste zusammenfassen		
every(.x, .p, ...)	Bestehen alle Elemente einen Test?	every(x, is.list)
some(.x, .p, ...)	Bestehen einige Elemente einen Test?	some(x, is.list)
has_element(.x, .y)	Enthält eine Liste ein bestimmtes Element?	has_element(x, list())
Liste umformen		
list_flatten(.x)	Entfernt eine Hierarchieebene aus einer Liste. Ähnlich wie unlist(), hier wird aber nur EINE Ebene entfernt. Gibt eine Liste zurück.	list_flatten(x)
list_transpose(.l)	Keht eine Liste von Listen „von innen nach außen“ um, z. B. wird ein Paar von Listen zu einer Liste von Paaren.	keep(x, ~length(.x) > 1) > list_transpose()
Mit Listen arbeiten		
set_names(x, nm)	Setzt die Namen eines Vektors oder einer Liste direkt oder mithilfe einer Funktion (importiert aus dem Paket rlang).	set_names(x, toupper)

Weitere Funktionen siehe Dokumentation: purrr.tidyverse.org.

Debugging und Fehlerbehandlung

Conditions an Nutzer:innen kommunizieren

<code>stop()</code>	Löst einen schwerwiegenden Fehler aus und bricht die gesamte Ausführung ab (verwenden, wenn die Funktion nicht fortgesetzt werden kann).
<code>warning()</code>	Zeigt mögliche Probleme an (verwenden, wenn einzelne Elemente einer vektorisierten Eingabe ungültig sind).
<code>message()</code>	Gibt informative Hinweise aus, die Nutzer:innen leicht unterdrücken können (z. B. die Schritte einer Programmausführung oder welche Werte für fehlende Argumente gewählt wurden).

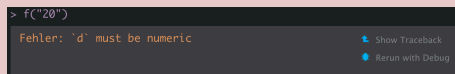
Umgang mit Fehlern, Warnungen und Hinweisen

<code>try(expr, silent)</code>	Setzt die Ausführung auch bei einem Fehler fort. Nützlich, wenn z. B. viele Modelle angepasst werden und einige nicht konvergieren. <code>expr</code> kann ein oder mehrere Funktionsaufrufe enthalten; bei mehrzeiligen Ausdrücken den Code in geschweifte Klammern setzen: <code>try({expr})</code> . Mit <code>silent=TRUE</code> wird die Meldung unterdrückt.
<code>tryCatch(expr, ...)</code>	Kann mit allen Conditions umgehen und erlaubt es, Handler-Funktionen anzugeben (= benannte Funktionen, die mit der Condition als Eingabe aufgerufen werden), die steuern, was bei einer ausgelösten Condition passiert. Beispiel: <code>tryCatch(sqrt("a"), error=function(e) print("You can't take the square root of a character."))</code>
<code>suppressMessages(expr)</code>	Unterdrückt Hinweise, die eine Funktion ausgibt. Beispiel: <code>suppressMessages(library(tidyverse))</code>
<code>suppressWarnings(expr)</code>	Unterdrückt Warnungen, die eine Funktion ausgibt.

Weitere Optionen siehe `?condition`.

Methoden zur Fehlersuche

RStudio bietet verschiedene Werkzeuge zur Fehlersuche. Standardmäßig ist die Einstellung **Error handler** aktiv, die unter **Debug > On Error** geändert werden kann. Tritt ein Fehler auf, erscheinen rechts in der Konsole diese beiden Optionen:



Fehler lokalisieren

Tritt ein Fehler auf, auf das Symbol **Show Traceback** in der Konsole klicken. Dies zeigt die Abfolge der Aufrufe (den sog. *Call Stack*), die zum Fehler geführt hat. Nützlich bei Fehlermeldungen, die sich nicht eindeutig zuordnen lassen. Ohne RStudio liefert `traceback()` dieselbe Information. Um diese Option dauerhaft für den Fehlerfall zu setzen: `options(error = traceback)`

Interaktiver Debugger

Reicht `traceback()` nicht aus, muss der Debug-Modus gestartet werden:

Browsen nach einem Fehler

Das RStudio-Werkzeug **Rerun with Debug** öffnet eine interaktive Debug-Sitzung, in der der Befehl, der den Fehler verursacht hat, erneut ausgeführt wird und an der Fehlerstelle pausiert (in der Konsole erscheint `Browse[1]>`).

Zu sehen sind:

- ein Tab im Editor mit dem entsprechenden Code
- die Objekte der aktuellen Umgebung im Environment-Fenster
- der Call Stack im Traceback-Fenster

In diesem Modus kann regulärer R-Code sowie eine Reihe spezieller Befehle ausgeführt werden:



- **n**: nächsten Schritt in der Funktion ausführen
- **s**: in die (neue) Funktion hineinspringen
- **f**: Schleife oder Funktion zu Ende ausführen
- **c**: interaktives Debugging verlassen und regulär fortfahren
- **Q**: Debugging beenden, Funktion abbrechen
- **Enter** wiederholt den vorherigen Befehl; abschalten mit `options(browserNLdisabled=TRUE)`
- **where**: gibt den Stack Trace der aktiven Aufrufe aus

Beliebigen Code browsen

`browser()`

Mit dieser Funktion lässt sich manuell in den interaktiven Debug-Modus wechseln. `browser()` an der Stelle einfügen, an der pausiert werden soll, und den Funktionsaufruf erneut ausführen (nach der Fehlerbehebung wieder entfernen!):

```
foo <- function(x) {  
  browser()  
  log(x)  
}  
foo("5")
```

RStudio-Werkzeug **Breakpoints**

Breakpoints verhalten sich ähnlich wie `browser()` und werden durch einen Klick links neben die Zeilennummer im Code gesetzt.

`recover()`

Aktiviert intern `browser()`, erlaubt es aber, die Umgebung eines beliebigen Aufrufs im Call Stack zu betreten. Üblicherweise wird es aktiviert, indem einmalig die folgende Option gesetzt wird: `options(error = recover)`. Zurück zur Standardbehandlung: `options(error = NULL)`

`debug()`

Diese Funktion fügt automatisch die Anweisung `browser()` in die ERSTE Zeile der angegebenen Funktion ein. `undebug()` entfernt sie wieder. Alternativ kann `debugonce()` verwendet werden.

Weitere Informationen in den Kapiteln **Conditions** und **Debugging** des Buchs *Advanced R* von Hadley Wickham.