KAPITEL 5

Variablen

Sobald Sie eine komplexere Operation ausführen wollen, benötigen Sie die Möglichkeit, Werte zwischenzuspeichern, um sie an anderer Stelle wieder einsetzen zu können. Dazu verwenden Sie Variablen.

Eine Variable ist im Prinzip nichts weiter als ein Name (*Symbol*), dem Sie einen Wert zuweisen. Dadurch bleibt der Wert erhalten und kann später unter dem Namen abgerufen werden. Der zugewiesene Wert kann ein beliebiges Objekt der Sprache **R** sein, also nicht nur eine Zahl oder eine Zeichenkette, sondern auch eine Datenstruktur, eine Funktion oder eine Umgebung.

Bei der Benennung von Variablen müssen die Namensregeln für Symbole eingehalten werden (siehe Kapitel 2).

In der **R**-Dokumentation und in manchen Lehrbüchern werden Variablen auch als »benannte Objekte« (*Named Objects*) bezeichnet. Das ist nicht ganz zutreffend, da der Variablenname keine Eigenschaft des Objekts ist. Außerdem ist die Bezeichnung »Variable« sicher leichter verständlich, wenn man sich schon mit anderen Programmiersprachen beschäftigt hat.

Umgebungen

Alle Variablen sind in sogenannten Umgebungen (*Environments*) gespeichert. Die Umgebungen bilden eine baumartige Struktur, jede Umgebung hat einen Namen und umfasst ihre Variablen in Form eines Verzeichnisses benannter Objekte sowie eine Referenz auf eine

übergeordnete Umgebung. Die Wurzel dieser Baumstruktur ist eine leere Umgebung.

Verwechseln Sie die Umgebungen in **R** nicht mit der Systemumgebung Ihres Rechners. Auf diese können Sie beispielsweise mit der Funktion `Sys.getenv()` zugreifen.

Im Programm gibt es immer eine *aktuelle* oder *aktive Umgebung*, in der sich alle neu angelegten Variablen befinden. Außerhalb einer Funktion ist dies die globale Umgebung (`GlobalEnv`). Bei jedem Aufruf einer Funktion wird eine neue Umgebung angelegt. Die aktuelle Umgebung erhalten Sie durch die Funktion `environment()`.

```
environment()
## <environment: R_GlobalEnv>
```

Jede Umgebung enthält neben den in ihr enthaltenen Variablen (zu denen auch Funktionen gehören) einen Verweis auf eine übergeordnete Umgebung (*Parent Environment*). Daraus ergibt sich eine Kette von Umgebungen, an deren Wurzel sich die leere Umgebung (`EmptyEnv`) befindet. Diese enthält, wie der Name schon sagt, weder Variablen noch einen Verweis auf eine übergeordnete Umgebung.

Die Kette der Umgebungen bildet einen Suchpfad, die der Interpreter auf der Suche nach einer Variablen anhand ihres Namens durchläuft und der aus den folgenden Etappen besteht.

- Innerhalb einer Funktion die Umgebung der Funktion, danach gegebenenfalls weitere Umgebungen, in deren Kontext die Funktionen jeweils definiert worden sind.
- Danach die globale Umgebung `R_GlobalEnv`, in der der interaktive Modus bzw. ein Programm außerhalb einer Funktion läuft.
- Dann die Umgebungen aller geladenen Erweiterungspakete in der umgekehrten Reihenfolge, in der sie geladen worden sind.
- Anschließend die *Autoload*-Umgebung, in der man Umgebungen aufführen kann, die erst bei Bedarf geladen werden (vgl. die Funktion `autoload()`).

- Dann die Basisumgebung (base) mit allen zum Basispaket gehörenden Objekten.
- Schließlich die leere Umgebung, in der allerdings definitionsgemäß nichts mehr zu finden ist und die das Ende der Kette signalisiert.

Alle Umgebungen, die der globalen Umgebung übergeordnet sind, sind normalerweise gesperrt und können vom Programm nicht verändert werden. Es ist aber möglich, eine Variable, die sich in einer solchen unveränderlichen Umgebung befindet, zu verdecken und ihr auf diese Weise vorübergehend einen anderen Wert zu verleihen.

Sie können sich eine kleine iterative (d.h. sich selbst aufrufende) Funktion bauen, die Ihnen die aktuelle und alle übergeordneten Umgebungen anzeigt:

```
parlist <- function(env=environment()) {
  cat(environmentName(env),"\n")
  if (!identical(env,emptyenv())) {
    parlist(parent.env(env))
  }
}
```

Die Funktion gibt mit cat() die als Parameter erhaltene Umgebung aus. Wenn dies nicht die leere Umgebung ist, ermittelt sie die übergeordnete Umgebung und ruft sich mit dieser selbst auf. Wenn Sie die Funktion ohne Argument aufrufen, startet sie mit der aktuellen Umgebung.

```
parlist()
## R_GlobalEnv
## package:stats
## package:graphics
## package:grDevices
## package:utils
## package:datasets
## package:methods
## Autoloads
## base
## R_EmptyEnv
```

Die Ausgabe beginnt mit der globalen Umgebung, fährt mit den Umgebungen der geladenen Pakete fort und endet schließlich bei der leeren Umgebung. (In Ihrer Umgebung kann die Liste anders aussehen.)

Sie brauchen eine solche Funktion nicht wirklich selbst zu schreiben. Mit der Funktion `search()` erhalten Sie ein ähnliches Ergebnis.

Mit Umgebungen arbeiten

Es gibt eine Reihe von Funktionen, mit denen Sie Umgebungen manipulieren können. Hier sind einige von ihnen. In der Regel haben sie einen Parameter, mit dem man die Umgebung benennen kann, auf die sie sich jeweils auswirken sollen. Lässt man ihn beim Aufruf weg, beziehen sie sich auf die aktive Umgebung. (Weitere Parameter der aufgeführten Funktionen können mittels der Hilfefunktion nachgeschlagen werden.)

```
# Die globale Umgebung
globalenv()
## <environment: R_GlobalEnv>
# Die Basisumgebung
baseenv()
## <environment: base>
# Die leere Umgebung
emptyenv()
## <environment: R_EmptyEnv>
# Eine neue Umgebung erzeugen
env <- new.env()
# Variablen zuweisen
assign("x",100,envir=env)
assign("y","Text",envir=env)
# Variablen auslesen
get("x",envir=env)
## [1] 100
# Vorhandensein einer Variablen prüfen
exists("x",envir=env)
## [1] TRUE
# Variable entfernen
rm("x",envir=env)
exists("x",envir=env)
## [1] FALSE
# Alle Variablen speichern
save(list=ls(env),file="meineUmgebung.tmp",envir=env)
```

```
# Gespeicherte Variablen in neue Umgebung einlesen
env2 <- new.env()
load(file="meineUmgebung.tmp",envir=env2)
ls(env)==ls(env2)
## [1] TRUE
```

Vordefinierte Variablen

Es gibt eine lange Reihe von Variablen, die in der Basisumgebung definiert sind und daher in jedem **R**-Programm zur Verfügung stehen. Dazu gehören – neben den zahlreichen Standardfunktionen – auch einige vorab definierte Vektoren.

```
letters   # ASCII-Kleinbuchstaben
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
## [15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
LETTERS   # ASCII-Großbuchstaben
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
## [15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
month.name  # englische Monatsnamen
## [1] "January"   "February"  "March"     "April"
## [5] "May"       "June"      "July"      "August"
## [9] "September" "October"   "November"  "December"
month.abb   # englische Monatsnamen, abgekürzt
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
## [10] "Oct" "Nov" "Dec"
pi          # Kreiszahl
## [1] 3.141593
T           # Abkürzung für TRUE
## [1] TRUE
F           # Abkürzung für FALSE
## [1] FALSE
```

Bei der Verwendung von vordefinierten Variablen ist eine gewisse Vorsicht angesagt, denn sie können zwar nicht verändert, aber mit lokalen Variablen verdeckt werden.

`NULL`, `TRUE`, `FALSE`, `NA` (einschließlich der Varianten `NA_integer_` usw.) sind dagegen keine Konstanten, sondern Schlüsselwörter und können daher nicht verändert oder verdeckt werden.

Variablen zuweisen

Eine Variable entsteht dadurch, dass ihr ein Wert zugewiesen wird.

```
a <- 1
```

Hier wird einer Variablen mit dem Namen »a« die Zahl 1 zugewiesen. Wenn die Variable nicht vorhanden ist, wird sie dadurch angelegt. Variablen werden nicht deklariert, sie können immer wieder andere Werte annehmen und haben keinen festgelegten Typ, d. h., Sie können ihr beispielsweise zuerst eine Zahl, dann eine Zeichenkette und schließlich auch noch eine Funktion zuweisen.

```
u <- 100
u
# [1] 100
u <- "Text"
u
# [1] "Text"
u <- function(r) pi * r * 2
u
function(r) pi * r * 2
```

Zusätzlich gibt es eine Funktion `assign()`, der Sie den Variablennamen als Text, den zu setzenden Wert und gegebenenfalls die betreffende Umgebung mitgeben. Diese Funktion ermöglicht es auch, Variablen programmgesteuert zu definieren

```
assign("u",100,envir=environment())
```

Beim Zuweisen einer Variablen mit den einfachen Pfeiloperatoren (`<-`, `->`) oder mit `=` greift der Interpreter immer auf die aktuelle Umgebung zu. Wenn sich eine Variable mit dem gleichen Namen in einer übergeordneten Umgebung befindet, wird sie gegebenenfalls verdeckt.

Mithilfe des Zuweisungsoperators mit doppelter Spitze (`<<-` oder `->>`) können Sie auch Variablen in übergeordneten Umgebungen ändern. Hierbei sucht der Interpreter in den übergeordneten Umgebungen, ob er eine Variable mit dem betreffenden Namen findet, und setzt deren Wert. Wenn er sie nicht findet, erhalten Sie eine Fehlermeldung: »Objekt *xyz* nicht gefunden«.

Der Versuch, eine Variable zu ändern, die bereits in einer Umgebung definiert ist, die der globalen Umgebung übergeordnet ist, führt zu einer Fehlermeldung, da diese Umgebungen normalerweise gesperrt sind.

Die folgende selbst gemachte Funktion `setzeX()` setzt die Variable `x` in der übergeordneten Umgebung auf den übergebenen Wert.

```
setzeX <- function(y) { x <<- y }
```

Den Effekt können Sie sehen, wenn nach dem Aufruf der Funktion der Wert von `x` noch erhalten ist.

```
x <- 41
setzeX(42)
x
## [1] 42
```

Wenn Sie in der Funktion nur den einfachen Zuweisungsoperator (`<-`) verwenden, funktioniert dies nicht, da dann die Variable `x` nur in der lokalen Umgebung der Funktion gesetzt wird, die nach der Beendigung der Funktion nicht mehr vorhanden ist.

Variablen verwenden

Um den Inhalt einer Variablen auszulesen, genügt es, den Namen der Variablen anzugeben.

```
x
## [1] 42
```

In **R** verwenden Sie eine Variable, indem Sie diese einfach benennen, also ihren Namen schreiben. Der Interpreter durchsucht dann die ganze Kette der Umgebungen, beginnend mit der aktuellen Umgebung, bis er eine Übereinstimmung gefunden hat oder bei der leeren Umgebung am Ende angekommen ist. Wenn er dabei nicht fündig wird, meldet er den Fehler »Objekt *x* nicht gefunden«.

Wenn Sie eine Variable in einer bestimmten Umgebung auslesen möchten oder wenn Sie den Variablennamen selbst in einer Variablen haben, verwenden Sie die Funktion `get()`, der Sie den Variablennamen als `character`-Objekt und die betreffende Umgebung übergeben.

Im folgenden Beispiel wird die vordefinierte Variable `pi` mit einem anderen Wert verdeckt. Beim Aufruf von `pi` erscheint dieser Wert, da er in der Suchreihenfolge zuerst auftaucht.

```
pi <- "Anderer Wert"
pi
## [1] "Anderer Wert"
```

Wenn wir in einem `get()`-Aufruf die Umgebung angeben, in der die Variable gesucht werden soll, erscheint der Originalwert der Variablen.

```
get("pi",envir=baseenv())
## [1] 3.141593
```

Nach dem Löschen der verdeckenden Variablen aus der aktiven Umgebung erscheint wieder der ursprüngliche Wert.

```
rm(pi)
pi
## [1] 3.141593
```

Umgebungen als Objekte

Oben haben Sie bereits gesehen, dass Umgebungen normale Objekte sind, die Sie Variablen zuweisen, an Funktionen übergeben usw. können. Sie sind aber nicht von atomischen Vektoren oder Listen abgeleitet, daher gelten für sie einige besondere Regeln.

- Anders als alle anderen Objekttypen wird eine Umgebung nicht als Kopie, sondern als Referenz übergeben. Daher kann es hier vorkommen, dass eine Funktion, die eine Umgebung als Parameter übernimmt, diese modifiziert.
- Der Gleichheitsoperator (`==`) ist nicht anwendbar. Wenn Sie wissen möchten, ob zwei Variablen dieselbe Umgebung referenzieren, verwenden Sie die Funktion `identical()`.
- In mancher Hinsicht können Umgebungen wie Listen behandelt werden, obwohl sie keine sind. So ist es beispielsweise möglich, auf einzelne Variablen mit doppelten eckigen Klammern oder mit dem `$`-Operator zuzugreifen.

```
env <- environment()
env$x
## [1] 42
env[['x']]
## [1] 42
env[['x']] <- 24
x
## [1] 24
```

Da die Variable env hier eine Referenz auf die aktive Umgebung hält, ist der veränderte Wert der Variablen x hier auch zu sehen.

```
x
## [1] 24
```