Funktionen in R programmieren

Mark Heckmann

14. Juli 2013

Zusammenfassung

Eine kurze Einführung zur Programmierung von Funktionen in R.

Inhaltsverzeichnis

1.1 Funktionen definieren

In den vorherigen Kapiteln ist bereits deutlich geworden, dass in R neben Datenstrukturen (Vektoren, Dataframes, Listen etc.) vor allem Funktionen eine zentrale Rolle spielen. In R ist es relativ einfach, eigene Funktionen zu programmieren. Die meisten der Funktionen, die in Paketen verfügbar sind, sind

ausschließlich in R geschrieben. Einige nutzen ggf. noch C oder Fortran Code, wenn es nötig ist, sehr schnelle Funktionen zu schreiben. Für nahezu alle unserer Zwecke ist es jedoch völlig ausreichend R zu beherrschen und R Funktionen zu schreiben.

Da R ein Open Source Programm ist, kann der Code jeder Funktion betrachtet werden. Dies ist in R in den meisten Fällen sehr einfach. Hierzu wird einfach der Funktionsname in die Konsole eingegeben und die Eingabetaste gedrückt. Betrachten wir zunächst den Code der Funktion `fisherz` aus dem Paket `psych`, die mit den eingegebenen Daten eine Fisher Z-Transformation durchführt.[1]

```
> library(psych)
> fisherz

function (rho)
{
    0.5 * log((1 + rho)/(1 - rho))
}
<environment: namespace:psych>
```

Der Code ist sehr kurz. Es ist nur die Code-Zeile in der Mitte, die die notwendigen Berechnungen vornimmt. Umrahmt wird die Berechnung von dem notwendigen Code, der R anzeigt, dass es sich hier um eine Funktionsdefinition handelt. In der letzten Zeile steht die Ausgabe `<environment: namespace:psych>`. Die Bedeutung dieser Angaben ist für uns nicht weiter relevant. Sie zeigt uns jedoch, dass die Funktion aus dem Paket `psych` stammt.

1.1.1 Die Funktion `function`

Schauen wir uns den Code näher an. Der Code beginnt mit dem Wort `function`. Hierbei handelt es sich um eine R Funktion, die dazu benutzt wird, *neue Funktionen* zu erzeugen (siehe `help("function")`). Die Funktion `function` erlaubt in der runden Klammer die Angabe von Argumenten. Dieses Mal sind diese jedoch nicht vorher festgelegt, sondern können von uns definiert werden. In den darauffolgenden geschweiften Klammern steht der innerhalb der Funktion auszuführenden R Code. Die allgemeine Form einer Funktionsdefinition ist somit folgende.

function (*Optionale Definition der Argumente*) {
  R-Befehle
}

[1] Die ist nicht zu verwechseln mit der z-Transformation (kleines z), die eine Variable standardisiert. Die Stichprobenverteilungsfunktion von Korrelationen ist besonders für Korrelationen nahe -1 und 1 sehr schief. Fishers Z-Transformation nähert die Verteilung wieder einer Normalverteilung an. Sie ist definiert als: $f(r) = 0{,}5 \ln\left(\frac{1+r}{1-r}\right)$, wobei r die Produkt-Moment-Korrelation der Merkmale in einer Stichprobe ist.

Bisher haben die Funktionen die wir kennengelernt haben stets einen Vektor, einen Dataframe oder eine andere Datenstruktur zurückgegeben (`mean(1:10)` gibt z. B. einen Vektor zurück). Die Funktion `function` hingegen erzeugt auf Basis der Angaben die wir machen eine neue Funktion und gibt diese zurück. Um die zurückgegebene Funktion zu speichern, verfahren wir genauso wie um z. B. einen Vektor zu speichern, d. h. indem wir sie einem Objekt zuweisen. Erzeugen wir nun die einfachste mögliche Funktion, die weder Argumente hat noch etwas berechnet und weisen sie dem Objekt `funk` zu.

```
> funk <- function(){}
```

Die Funktion `funk` kann nun wie jede andere Funktion aufgerufen werden, d.h. indem der Funktionsname gefolgt von den Argumenten innerhalb der runden Klammer angegeben werden. Da wir in diesem Fall keine Argumente definiert haben, müssen keine angegeben werden.

```
> funk()

NULL
```

Die Funktion gibt ein leeres Objekt (`NULL`) zurück. Dies liegt daran, dass sie keine Berechnungen vornimmt und wir auch nicht definiert haben, was die Funktion zurückgeben soll. Da in R ausnahmslos jede Funktion ein Objekt zurückgibt, gibt unsere Funktion nun ein leeres Objekt zurück. Offenkundig erfüllt die Funktion keinen Zweck und ist somit nutzlos. Sie stellt jedoch das Grundgerüst dar, mit dem wir nun weiterarbeiten wollen.

Unser nächstes Ziel ist es, dass die Funktion den Vektor `1:10` zurückgibt, wenn sie aufgerufen wird. Um zu definieren, welches Objekt die Funktion zurückgeben soll, wird die Funktion `return` genutzt. Sobald `return` aufgerufen wird, wird die definierte Funktion beendet und das Objekt, dass `return` als Argument übergeben wurde, wird von der Funktion zurückgegen. Üblicherweise steht `return` am Ende einer Funktion, da Code der nach dem Aufruf von `return` folgt, nicht mehr ausgeführt wird. Folgende Funktion gibt den Vektor `1:10` zurück.

```
> funk <- function(){
   return(1:10)
 }
> funk()

 [1]  1  2  3  4  5  6  7  8  9 10
```

Bei der Funktion `fisherz` aus dem `psych` Paket (s. o.) fällt auf, dass in dem Code kein `return` Ausdruck enthalten ist. In diesem Fall nimmt R an, dass das Objekt, das zurückgegeben werden soll, das Ergebnis der letzen Zeile des Codes ist. Es ist somit gleichgültig, ob um die `1:10` ein `return` geschrieben wird oder nicht. In R ist es sehr üblich, die `return` Angabe in der letzen Zeile wegzulassen. Nachfolgende Funktion ist somit ist identisch mit dem vorherigen.

```
> funk <- function(){
   1:10
 }
> funk()
 [1]  1  2  3  4  5  6  7  8  9 10
```

1.1.2 Argumente hinzufügen

Was Funktionsargumente sind wurde in Kapitel **XXX** besprochen. Dort wurde auch gezeigt, dass es Argumente gibt, die stets angegeben werden müssen, damit eine Funktion ausgeführt werden kann. Dieser Argumenttyp wird im Folgenden *nicht-optional* genannt. Für andere Argumente existieren Voreinstellungen und es ist *optional*, diesen Argumenten beim Funktionsaufruf einen Wert zu übergeben. Auch ohne einen Wert für diese Argumente wird beim Funktionsaufruf kein Fehler verursacht. [2]

Nicht-optionale Argumente

Fügen wir unserer Funktion nun ein erstes *nicht-optionales* Argument hinzu. Unser Ziel ist, dass nicht `1:10` ausgegeben wird, sondern ein Objekt, das wir der Funktion `funk` beim Aufruf übergeben. Hierzu definieren wir innerhalb der runden Klammer die gewünschten Argumente der Funktion; in diesem Fall das Argument `x`. Im Code in der Funktion kann `x` nun wie jedes andere Objekt für Berechnungen etc. genutzt werden. Wir wollen es zunächst jedoch lediglich wieder ausgeben. Die Funktion tut somit nichts anderes, als das Objekt `x`, das ihr übergeben wird, wieder auszugeben.

```
> funk <- function(x){
   return(x)
 }
> funk(x=1:10)
 [1]  1  2  3  4  5  6  7  8  9 10
> funk(c("a", "b", "c"))
[1] "a" "b" "c"
```

Bei der Funktiondefinition haben wir festgelegt, dass die Funktion das Argumnet `x` erwartet. Wenn der User dieses nicht angibt, wird ein Fehler erzeugt.

```
> funk()
Error in funk() : argument "x" is missing, with no default
```

[2] So kann z.B. `rnorm()` nicht ohne Angabe eine Wertes für die Anzahl der zu erzeugenden zufälligen Werte ausgeführt werden und verursacht eine Fehler; `rnorm(10)` erzeugt 10 Werte mit den Standardeinstellungen `mean=0` und `sd=1`, wenn die keine Werte für die Argumente `mean` und `sd` übergeben werden.

Optionale Argumente

Um ein Argument optional zu machen, ist es notwendig, dass das Argument in der Funktionsdefinition einen *Standardwert* (engl. *default value*) zugewiesen bekommt.[3] Sollte der Nutzer für das entsprechende Argument beim Aufruf der Funktion keinen Wert übergeben, so wird der Standardwert verwendet. Ein Standardwert wird zugewiesen, indem hinter dem Argument ein Gleicheitszeichen gefolgt von den Standardwert eingesetzt wird. Im Folgenden erhält `x` die Zeichenkette `"mein standardwert"` als voreingestellten Wert. Nun kann die Funktion auch ohne `x` Argument aufgerufen werden, ohne einen Fehler zu generieren.

```
> funk <- function(x="mein standardwert"){
   return(x)
 }
> funk(1:10)                # Aufruf mit übergebenem Argument
 [1]  1  2  3  4  5  6  7  8  9 10
> funk()                    # Aufruf ohne Argument
[1] "mein standardwert"
```

Mehrere Argumente

Eine Funktion kann mit beliebig vielen nicht-optionalen und optionalen Argumenten ausgestattet werden. Die Argumente werden hierzu durch ein Komma getrennt nacheinander aufgelistet. Die Namen der Argument müssen hierbei eindeutig sein. Nachfolgende Funktionen enthält ein nicht-optionales und zwei optionale Argumente mit Voreinstellung.

```
> add <- function(eins, zwei=1, drei=2){
   eins + zwei + drei
 }
> add(10)
[1] 13
> add(10, 20)
[1] 32
> add(10, drei=30)
[1] 41
```

Argument name matching

[3] Diese Aussage ist im Grund nicht richtig, aber vorerst zum Verständnis hilfreich. Wann dies nicht zutrifft sehen wir im Abschnitt 1.3.2.

Wenn die Namen der Argument beim Aufruf der Funktion nicht ausgeschrieben werden, versucht R die Argumente automatisch zuzuordnen (vgl. Abschnitt XXX). Dies gelingt, solange die Argumentbezeichnungen beim Aufruf eindeutig sind. Dieser Vorgang nennt sich *argument name matching*. Hierzu ist es nicht nötig bei der Programmierung spezielle Schirtte zu uneternehmen, R macht dies automatisch. Folgende Aufrufe sind somit identisch.

```
> add(eins=10, zwei=5, drei=20)
> add(e=10, z=5, d=20)
> add(z=5, e=10, 20)
```

Überprüfung der Argumente

Wir sind nun soweit, dass wir z. B. eine eigene Funktion zur Berechnung des Mittelwerts schreiben können. Hierzu definieren wir ein Argument `x`, das einen numerischen Vektor mit den Werten enthält, aus denen der Mittelwert berechnet wird.

```
> my.mean <- function(x) {
    mw <- sum(x) / length(x)
    return(mw)
  }
```

Diese Funktion funktioniert einwandfrei solange ein *numerischer* Vektor übergeben wird. Es treten jedoch Problem auf, wenn dies nicht der Fall ist. Zum einem dann, wenn der Vektor `NA`s enthält sowie wenn er nicht numerisch ist und beispielsweise Zeichen enthält.

```
> my.mean(1:5)                # funktioniert
[1] 3
> my.mean(c(1:5, NA))         # NA wird zurückgegeben
[1] NA


> my.mean(c(1:5, "test"))     # Zeichenkette verursacht Fehler


Error in sum(x) : invalid 'type' (character) of argument
```

Das erste Problem können wir eliminieren, indem wir den Vektor mit Hilfe der Funktion `na.omit` um die `NA`s bereinigen. Beim zweiten Fehler gibt es nichts zu bereinigen, da der Mittwelwert einer Zeichenketten keinen Sinn ergibt. Es ist hierbei jedoch üblich, am Anfang einer Funktion zu überprüfen, ob die Argumente den nötigen Anforderungen entsprechen. Ist dies nicht der Fall, sollte ein Fehler ausgegeben, um den Nutzer zu informieren und die Funktion abgebrochen werden. Hierzu dient die Funktion `stop`.

```
> my.mean <- function(x){
   if (! is.numeric(x))              # enthält x Zahlen?
     stop("'x' must be numeric")     # wenn nicht Fehlermeldung ausgeben
   x <- na.omit(x)                   # NAs entfernen
   mw <- sum(x) / length(x)
   return(mw)
 }

> my.mean(c(1:5, NA))

[1] 3

> my.mean(c(1:5, "test"))
Error in my.mean(c(1:5, "test")) : 'x' must be numeric
```

Im Code wird über die Funktion `is.numeric` zunächst geprüft, ob `x` numerisch ist. Wenn dies der Fall ist gibt die Funktion den Wert `TRUE` zurück. Uns interessiert jedoch der Fall, wenn `x` *nicht* numerisch ist. In der `if`-Bedingung wird deshalb die Negation geprüft (`! is.numeric(x)`). Wenn `x` *nicht* numerisch ist wird somit die Funktion `stop` aufgerufen. Diese sollte eine kurze Info enthalten, die dem Nutzer erklärt, warum ein fehler aufgetreten ist.

1.1.3 Objekte zurückgeben

Der Zweck der meisten Funktionen ist, dass sie etwas zurückgeben. Der Objekttyp, den eine Funktion zurückgibt, ist beliebig. Es kann ein Vektor, eine Matrix, eine Liste aber auch ein beliebiges anderes Objekt sein. Eine Funktion kann jedoch stets nur *ein* Objekt zur Zeit zurückgeben. Sollen mehrere Informationen zurückgegeben werden, so müssen diese in *einem* R-Objekt zusammengefasst werden.

Nehmen wir an, wir möchten eine Funktion schreiben, die den Mittelwert als auch die Standardabweichung berechnet und diese zurückgibt. Hierzu können beide Werte berechnet und z.B. in Form eines (benannten) Vektors zurückgegeben werden.

```
> mean_and_sd <- function(x) {
   m <- mean(x)                      # Mittelwert berechnen
   s <- sd(x)                        # Standardabweichung berechnen
   res <- c(mean=m, sd=s)            # m und s in Vektor zusammenfassen
   return(res)                       # Rückgabe des Vektors
 }
> mean_and_sd(1:4)

    mean       sd
2.500000 1.290994
```

Die Funktion gibt nur *ein* Objekt zurück, dass jedoch beide berechneten Werte enthält. Nach demselben Prinzip können auch noch mehr Informationen zurückgeben werden. Nehmen wir an, wir möchten, dass die Funktion zusätzlich

die Werte selber zurückgibt. In diesem Fall ist ein Vektor nicht mehr geeignet. Eine Option bietet der Datentyp Liste.

```
> mean_and_sd <- function(x) {
    m <- mean(x)                    # Mittelwert berechnen
    s <- sd(x)                      # Standardabweichung berechnen
    res <- list(x=x, mean=m, sd=s)  # alles in Liste zusammenfassen
    return(res)                     # Liste zurückgeben
  }
> mean_and_sd(1:4)

$x
[1] 1 2 3 4

$mean
[1] 2.5

$sd
[1] 1.290994
```

Die `return` Funktion

In Abschnitt XXX haben wir gesehen, dass der Funktion `return` jenes Objekt übergeben wird, das zurückgegeben werden soll. Im letzten Beispiel war dies das Objekt `res`. Weiterhin wurde darauf hingewiesen, dass es auch möglich ist, das `return` wegzulassen. In diesem Fall wird der letzte Ausdruck innerhalb des Funktionscodes zurückgegeben. Der letzte Ausdruck wird also so behandelt, als ob dieser innerhalb der `return`-Funktion steht. Nachfolgende Funktion ist somit identisch zum vorherigen Beispiel.

```
> mean_and_sd <- function(x) {
    m <- mean(x)                    # Mittelwert berechnen
    s <- sd(x)                      # Standardabweichung berechnen
    list(x=x, mean=m, sd=s)         # Liste erzeugen und zurückgeben
  }
```

Sobald die Funktion `return` aufgerufen wird die Ausführung des Funktionscodes beendet und das Objekt, dass `return` übergeben wurde, wird zurückgegeben. Somit ist es egal ob `return` am Ende des Funktioncodes noch einmal erscheint, denn es wird in beiden Fällen der letzte Ausruck zurückgegeben, mit oder ohne `return`. Anders verhält es sich jedoch, wenn `return` mitten in dem Funktionscode auftaucht. Denn aller Code der noch folgen sollte wird nicht mehr ausgeführt, wenn `return` *aufgerufen* wird.

```
> funk <- function() {
    a <- 10
    return(a)               # Funktion wird hier beendet
    a <- 20
  }
> funk()
```

```
[1] 10
```

Obwohl der Wert von `a` am Ende des Codes `20` steht spielt dies keine Rolle mehr, da die Ausführung des Codes bereits bei `return` endet.

Mehfaches `return` vermeiden

Da die Funktion `return` die Code Ausführung abbricht und ein Objekt zurückgibt, könnte man die Funktion mehrfach in den Code schreiben.

```
> auswahl <- function(x) {
   if (x > 0)
     return(1)
   if (x < 0)
     return(-1)
   if (x == 0)
     return(0)
 }
> auswahl(5)

[1] 1

> auswahl(-5)

[1] -1
```

`return` taucht hier an verschiedenen Stellen auf. Bis auf wenige Fälle ist dieser Ansatz jedoch nicht zu empfehlen, da er eine Funktion schwer lesbar macht. Wenn der Funktionsocde sehr lang ist, ist es i.d.R. schwer nachvollziehbar, an welcher Stelle ein Objekt zurückgegeben wird. Es empfiehlt sich eher der Ansatz, das zurückzugebende Objekt jeweils zu zwischenzuspeichern und dann am Ende der Funktion zurückzugeben. Die vorherige Funktion könnten ebenso folgendermaßen aussehen.

```
> auswahl <- function(x) {
   if (x > 0)
     res <- 1
   if (x < 0)
     res <- -1
   if (x == 0)
     res <- 0
   return(res)
 }
```

Unsichtbare Rückgabe von Werten

Beim Aufrufen einiger Funktionen fällt auf, dass in der Konsole gar nichts passiert. Z.B. gibt die Funktion `hist` zwar ein Histogramm aus, es wird jedoch kein Objekt in der Konsole ausgegeben.

```
> x <- rnorm(100)
> hist(x)
```

Dieses Verhalten ist bei vielen Grafikfunktionen anzutreffen. Die Funktion `hist` wird schließlich nur aufgerufen, um eine Grafik zu erzeugen, nicht um etwas in der Konsole anzuzeigen. Es ist jedoch ein Irrtum anzunehmen, dass die Funktion `hist` nichts zurückgibt (denn jede Funktion tut dies). Die Rückgabe ist in diesem Fall nur unsichtbar. Eine unsichtbare Rückgabe kann erreicht werden, indem anstelle von `return` die Funktion `invisible` genutzt wird. Die Funktion ist identisch mit `return` mit dem einzigen Unterschied, dass das zurückgegeben Objekt *nicht* in der Konsole ausgegeben wird. Folgende Funktion gibt die Werte 1 bis 10 unsichtbar zurück.

```
> funk <- function() {
    invisible(1:10)
  }
> funk()
```

Dass die Funktion `funk` trotzdem ein Objekt zurückgibt wird erst deutlich, wenn wir es speichern und danach in der Konsole ausgeben.

```
> a <- funk()
> a
 [1]  1  2  3  4  5  6  7  8  9 10
```

1.1.4 Funktionen benennen

Sprechende Namen vergeben

Es ist hilfreich, Funktionen *sprechende Namen* zu geben. Dies sind Namen, die aus sich heraus beschreiben, was eine Funktion macht. Eine Funktion mit dem Namen `werteBerechnen` ist z.B. nichtssagend. Es bleibt unklar, was genau sie berechnet. Bei der Funktion `mean` hingegen ist es eindeutig. Man wird schnell merken, dass der eigene Code furchtbar unleserlich wird, wenn die Funktionen keine sprechenden Namen tragen. Ziel ist es schließlich, den Programmcode *lesen* zu können, ohne erst an jeder Stelle nachschlagen zu müssen, was eine Funktion genau macht.

Zwar sind sprechende Namen sehr hilfreich, sie können jedoch auch sehr schnell sehr lang werden. Hier muss man einen Mittelweg zwischen Verständlichkeit und Länge des Namens finden. Die Funktion `replace_all_values_in_string` ist verständlich aber zugleich sehr lang. Viele Programmierer nutzen deshalb Akronyme. Die Funktion könnte z.B. stattdessen `str_replace_all` heißen.[4] Auch die Funktion `rnorm`, die uns schon häufig begegnet ist, ist ein Akronym:

[4] Dieser Name ist nicht zufällig gewählt sondern ist eien Funktion aus dem `stringr` package, die genau diese Aufgabe hat.

`r` steht für *random*, `norm` für *normal distribution*.

Lesbarkeit der Namen erhöhen

Um die Lesbarkeit eines längeren Funktionsnamen zu erhöhen, werden i.d.R. zwei verschiedene Schreibvarianten genutzt. Die erste Variante ist die gebräuchlichere und nennt sich *CamelCase* (deu. *Binnenmajuskel*). Der Name spielt auf die Höcker eines Kamels an, wobei sich auf dem Rücken des Tieres hohe und tiefe Stellen abwechseln. Bei CamelCase wechseln sich im Wort analog Groß- und Kleinbuchstaben ab. Jeder neuer Bestandteil eines Funktionsnamens bekommt einen Großbuchstaben. Der erste Buchstabe ist üblicherweise ein Kleinbuchstabe. Die CamelCase Variante der Funktion `str_replace_all` wäre somit `strReplaceAll`.

Die zweite weniger übliche Variante ist die Nutzung von Unterstrichen um den Funktionsnamen leserlich zu gestalten und wird *snake_case* genannt. An der Stelle vor der beim CamelCase ein Großbuchstabe steht, steht stattdessen ein Unterstrich. Großbuchstaben sind dann im Namen nicht enthalten (z.B. `str_replace_all`).

Ich selber wurde mit der CamelCase Variante sozialisiert, bevorzuge inzwischen jedoch Variante mit den Unterstrichen, da ich sie für besser lesbar halte. Hier scheiden sich jedoch die Geister.

1.2 Philosphie des modulares Programmierens

Als Anfänger der (R)-Programmierung neigt man dazu, sehr langen Funktionscode zu schreiben und teilweise viele Einzelschritte in *eine* große Funktion zusammenzufügen. Die Folge davon ist, dass der Code schnell unübersichtlich und somit schwer lesbar und schwer zu warten ist. Um diesem Nachteil entgegenzuwirken wird in vielen Programmiersprachen versucht *modular* zu programmieren. Dies bedeutet, dass logisch zusammengehörige Teile eines Ccodes in einzelne Module zusammengefasst werden. In R kann ein Modul z.B. eine Funktion sein, die eine eigene klar definierte Aufgabe übernimmt.

Unser Ziel sei es, eine Funktion schreiben, die die Varianz für einen numerischen Vektor berechnet.[5] Nehmen wir weiter an, dass uns die Funktion `mean` hierzu nicht zur Verfügung steht. Eine Lösung wäre folgende.

```
> varianz <- function(x) {
   mw <- sum(x) / length(x)
   sum((x - mw)^2) / (length(n) - 1)
 }
```

[5] Die Varianzformel lautet $s^2 = \frac{\sum(x-\bar{x})^2}{n-1}$.

Zwar ist der Code hinreichend einfach, so dass es nicht nötig wäre, diesen weiter aufzusplitten. Er kann uns jedoch dienen, um das Grundprinzip zu veranschaulichen.

Zusammengehörige Programmteile zu Einheiten zusammenfassen

Einen Mittelwert zu berechnen ist eine Aufgabe, die immer wieder vorkommt. Es ist somit sinnvoll, diesen Schritt als eine eigene Einheit zu begreifen und ihn als unabhängiges Modul zu programmieren. Die Zusammenfassung in eine Einheit wird erreicht, indem wir aus der Berechnung des Mittelwerts eine eigenständige Funktion machen (was in R ja ohnehin der Fall ist).

```
> mittelwert <- function(x) {
    sum(x) / length(x)
 }


> varianz <- function(x) {
   mw <- mittelwert(x)
   sum((x - mw)^2) / (length(n) - 1)
 }
```

Die Berechnung des Mittelwerts ist nun in eine eigene Funktion ausgegliedert. Dies macht den Code von der Varianzfunktion etwas lesbarer. Zum anderen haben wir nun erreicht, dass wenn wir Änderungen in der Funktion `mittelwert` vornehmen können, ohne dies in jeder Funktion die die Funktion `mittelwert` nutzt, einzeln tun zu müssen. Dies wäre z.B. dann relevant, wenn wir eine Möglichkeit fänden, den Mittelwert schneller zu berechnen als mit obigen Code.

Die Vorteile einer modularen Programmierweise werden i.d.R. erst mit der Zeit deutlich, wenn man beginnt, längere Funktionen oder Pakete zu programmieren. Die ausgegliderten Teile haben dann anders als in unserem Beispiel mehrere Dutzend Zeilen Code, so dass der Übersichtliehkeitsgewinn beträchtlich ist. Wenn man mit dem Prgramieren beginnt ist es zunächst unklar, wann es sinnvoll ist, einen Programmteil in separate Funktionen auszugliedern. Mit der Zeit gewinnt man jedoch ein Gefühl dafür. Meine Erfahrung ist, dass man zu Beginn viel zu wenig von dem Gestaltungsprinzip des modularen Programmierens Gebrauch macht. Versuchen Sie möglichst früh diesem Prinzip zu folgen.

1.3 Fortgeschrittene Themen

Sie haben bisher die Grundlagen kennengelernt, um eigene Funktionen in R zu schreiben. In diesem Kapitel soll es darum gehen, ein besseres Verständnis von der Funktionsweise und den Möglichkeiten bei der Funktionsprogrammierung zu gewinnen.

In dem Abschnitt das *Drei-Punkte Argument* wird eine besonderer Argumentetyp vorgestellt, der nicht nur eine sondern beliebig viele Argument-Werte-Paare aufnehmen kann.

In dem Teil *Lazy Evaluation* wird auf eine grundlegende Funktionweise eingegangen, wie R auszuwertende Ausdrücke behandelt.

1.3.1 Das Drei-Punkte-Argument (...)

Im Abschnitt wurde erklärt welche Bedeutung das Drei-Punkte-Argument (...) beim Funktionsaufruf hat.[6] Hier soll die Frage geklärt werden, wie das dots-Argument innerhalb eigener Funktionen nutzen kann.

Das Drei-Punkte-Argument unterscheidet sich im Handling ein wenig von den Verfahren für Standardargument. Beim Handling können wir zwei Fälle unterscheiden. Im ersten Fall wird das dots-Argument dazu genutzt viele Argumente-Werte-Paare an eine weitere Funktion weiterzugeben. Im zweiten Fall wird gezielt auf einzelne Argumente-Werte-Paare zugegriffen.

Viele Argumente-Werte-Paare an eine Funktion weitergeben

Nehmen wir an, wir wollen eine Funktion schreiben, die eine Berechnung anstellt und einen Wert zurückliefert. Wir wollen in der Funktion angeben können, auf wieviele Stellen gerundet werden soll. Hierzu fügen wir ein Argument `digits` ein, dass die Voreinstellung `1` hat.

```
> addition_1 <- function(x, y, digits=1) {
   s <- x + y
   round(s, digits=digits)
 }
> addition_1(2.123, 1.123, digits=2)

[1] 3.25
```

Wir hätten dieses Verhalten alternativ mittels des dots-Arguments erreichen können.

```
> addition_2 <- function(x, y, ...) {
   s <- x + y
   round(s, ...)
 }
> addition_2(2.123, 1.123, digits=2)

[1] 3.25
```

[6] Die korrekte Bezeichnung ist hier eigentlich *Ellipse* (eng. *ellipsis*). Viele Leute schreiben jedoch *dot-dot-dot* oder *dots* argument, so dass Sie unter *ellipsis* nicht immer fündig werden.

An der Stelle wo vorher das Argument `digits` stand stehen nun die drei Punkte. Dies hat zur Folge, dass alle Argument-Werte-Paare, die hinter das Argument `y` geschrieben werden, in den drei Punkten gespeichert werden. Die drei Punkte können innerhalb einer Funktion an andere Funktionen weiteregegeben werden. In unserem Fall geben wir die Punkte an die Funktion `round` weiter. Die Funktion `round` durchsucht nun alle Argument-Wert-Paare in den drei Punkten und nutzt automatisch all jene, die sie kennt. In diesem Fall ist es nur das Argument `digits`, das `round` kennt. Würden in den Punkten weitere Argumentenamen stehen, über die `round` ebenfalls verfügt würden diese auch von `round` genutzt werden. Auch hier greift im übrigen das *argument name matching*. Selbst wenn `digits` nicht ganz ausgeschrieben wird, erkennt `round`, dass der Anfang `dig` zu dem Argument `digits` gehört.

```
> addition_2(2.123, 1.123, dig=3)
[1] 3.246
```

In welchen Situation ist es hilfreich das dots-Argument zu nutzen? Generell dann, wenn es sich um sehr viele Funktionsargument handelt, die wir weitergeben wollen. Dies ist z.B. häufig bei der `plot` Funktion der Fall. Plot verfügt über die Argumente `x`, `y` und `....`

```
> args(plot)
function (x, y, ...)
NULL
```

Im Hilfetext von `plot` steht, dass die Argumente zu weiteren Funktionen weitergegeben werden. Eine dieser Funktionen ist `par`, über die viele Grafikeinstellungen festgelegt werden können (siehe `?par`). Angenommen wir wollen eine Funktion schreiben, die sich genauso wie `plot` verhält, als Voreinstellung jedoch die Farbe blau verwendet.

```
> blue_points <- function(x, y, col="blue") {
   plot(x, y, col=col)
 }
> blue_points(1:10, 1:10)
```

Das Problem ist, dass es in der Funktion `blue_points` nun nicht mehr möglich ist, weitere Argumente zu übergeben. Wollten wir z.B. den plot character verändern (Argument `pch`), so müssen wir dafür sorgen, dass auch dieser an `plot` weitergegeben wird. Wir könnten hierzu `blue_points` ein weiteres Argument `pch` hinzufügen. Es wäre jedoch mühselig, dies für jedes einzelne Argument zu tun. Einfacher ist es, das dots-Argument zu nutzen.

```
> blue_points <- function(x, y, col="blue", ...) {
   plot(x, y, col=col, ...)
 }
> blue_points(1:10, 1:10, pch=16)
```

An die Funktion `blue_points` können nun beliebig viele Argumente übergeben werden, die alle an `plot` weitergegeben werden.Die Funktion ist identisch mit `plot`, mit dem einzigen Unterschied, dass sie nun `col="blue"` als Voreinstallung besitzt.

Auf Argumente-Werte-Paare direkt zugreifen

Bisher wurde der häufigste Nutzungsfall des dots-Arguments behandelt, bei dem der Inhalt der dots komplett an eine andere Funktion weitergeben wird. Eine weitere Möglichkeit besteht darin, auf den Inhalt der dots direkt zurückzugreifen. Hierzu müssen die dots zunächst in eine Liste umgewandelt werden. Dies geschieht durch den Befehl `list(...)`. Es wird eine Liste erzeugt, die alle Argument-Werte-Paare enthält, die in den dots enthalten waren.

```
> show_dots <- function(...) {
   dots <- list(...)
   dots
 }
> show_dots(a="eins", b=1:5)

$a
[1] "eins"

$b
[1] 1 2 3 4 5
```

Die Liste mit dem Inhalt der dots wird i.d.R. `dots` genannt. IM Folgenen wird auf das Argument `y` zugegriffen.

```
> f <- function(...) {
   dots <- list(...)
   dots$y
 }
> f(y=1)

[1] 1
```

In diesem Zusammenhang ist weiterhin die Funktion `hasArg` besonders nützlich. `hasArg` überprüft, ob innerhalb des Funktionsaufrufs ein bestimmtes Argument zu finden ist. Im Folgenden wird nach dem Argument `y` geschaut. Wenn dieses vorhanden ist wird.

```
> get_y <- function(...) {
   if (hasArg(y)) {
     dots <-list(...)
     dots$y
   } else {
     "no argument y"
   }
 }
> get_y(x=1)
> get_y(y=1)
```

1.3.2 Lazy Evaluation

Schreiben wir nun eine weitere Funktion, die wir in den nachfolgenden Schritten erweitern und verbessern werden. Als D-Mark Nostalgiker wünschen wir uns eine Funktion, die Euro in D-Mark umrechnet. Der Wechselkurs beträgt $1\,Euro = 1.95583\,DM$. Dies könnte wie folgt aussehen.

```
> euro_zu_dm <- function(x){
   x * 1.95583
 }
> euro_zu_dm(1)

[1] 1.95583
```

Wir wollen die Funktion noch ein wenig verbessern, indem wir sowohl die DM als auch die Euroangabe auf zwei Nachkommastellen gerundeter zugleich ausgeben.

```
> euro_zu_dm <- function(x){
   dm <- 1.95583 * x
   r <- c(Euro=x, DM=dm)
   round(r, 2)
 }
> euro_zu_dm(1)

Euro   DM
1.00 1.96
```

Wir sind noch nicht ganz zufrieden, denn wir würden die Umrechnung auch gerne in die andere Richtung vornehmen können. Hierzu ist es hilfreich, wenn wir zunächst betrachten, wie R die übergebenen Argumente auswertet. Machen wir einen kurzen Exkurs, bevor wir die Funktion verbessern. Wir haben vorhin gesagt, dass R einen Fehler ausgibt, wenn Argumente, die keine Standardeinstellung haben, im Funktionsaufruf nicht angegeben werden. Nachfolgende Funktion `foo` hat zwei Argumente, `x` und `y`. Beide verfügen über keine voreingestellten Werte. Dementsprechend würden wir davon ausgehen, dass nachfolgender Funktionsaufruf einen Fehler erzuegt.

```
> foo <- function(x, y){
   return(x)
 }
> foo(x=100)

[1] 100
```

Warum ist dies nicht der Fall? Der Grund hierfür liegt darin, dass R beim Durchlaufen der Funktion kein Mal auf das Objekt `y` trifft. Denn `y` wird nirgendwo im Code benutzt. R versucht jedoch erst dann den Wert des Objekts `y` zu erhalten, sobald es dieses, z.B. für eine Berechnung, benötigt. Vorher fällt es R nicht auf, dass `y` nicht vorhanden ist. R ist suzusagen *faul* und wird erst aktiv, wenn es muss. Es schaut nicht gleich eifrig beim Aufruf einer Funktion,

ob alle Argumente da sind, sonden erst später. Dieser Mechanismus wird als *lazy evaluation* bezeichnet.

Verdeutlichen wir dies an einem zweiten Beispiel. Nachfolgende Funktion addiert die Werte der Argumente von `x` und `y` wenn `x` kleiner als Null ist. Ansonsten gibt es einfach den Wert von `x` wieder aus.

```
> f <- function(x, y){
   if (x < 0) {
     r <- x + y
   } else {
     r <- x
   }
   r
 }
```

Für die Auswertung der Argument innerhalb der Funktion bedeutet dies, dass R den Wert `y` nur dann auswerten will, wenn `x` kleiner als Null ist. Nur in diesem Fall wird ein Fehler generiert.

```
> f(x=1)

[1] 1

> f(x=-1)
Error in x + y : 'y' is missing
```

Der Mechanismus der `lazy evaluation` hat interessante Konsequenzen, die wir uns zunutze machen können. Nachfolgende Funktion gibt die Argumnet `x` und `y` aus. Das Argument `x` muss hierbei angegeben werden. Für den Fall jedoch, dass das der Wert für `y` fehlt, wird dieser aus dem Wert des Arguments `x` berechnet.

```
> f <- function(x, y=x/2){
   c(x=x, y=y)
 }
> f(1)

  x   y
1.0 0.5
```

Warum funktioniert dies? R will den Vektor, der zurückgegen werden soll erzeugen. In diesem ist das Objekt `y` enthalten. R sucht nun nach dem Objekt und findet es bei den vordefinierten Argumenten. Die Voreinstellung ist, dass `y` die Hälfte von `x` ist. `x` ist ebenfalls bekannt und somit kann R nun auch den Wert für `y` berechnen.

Dieser Ansatz kann erweitert werden, indem das Argument `x` in der Voreinstallung durch `y` und `y` durch `x` definiert wird.

```
> f <- function(x=2*y, y=x/2){
   c(x=x, y=y)
 }
```

In diesem Fall ist es gleich, welches der Argumente vom Nutzer ausgelassen wird. Wenn es `x` ist wird `y` aus `x` berechnet, wenn es `y` ist wird `x` aus `y` berechnet. Auch beide Argumente können natürlich angegeben werden. In diesem Fall wird keines der beiden aus dem anderen berechnet.

```
> f(y=1)

x y
2 1

> f(x=2)

x y
2 1

> f(2,1)

x y
2 1
```

Kehren wir nun zu unserer Euro-Umrechnungs-Funktion zurück und machen uns den `lazy evaluation` Mechanismus zu nutze. Hierzu schreiben wir die Umwandlung der Währungen direkt in die Voreinstellung der Argumente. Nun ist es egal, ob Euro oder DM angegeben wird. Die Umwandlung funktioniert nun in beide Richtungen.

```
> euro_dm_converter <- function(euro = dm / 1.95583,
                                dm = 1.95583 * euro){
   c(Euro=euro, DM=dm)
 }
> euro_dm_converter(euro=1)

   Euro      DM
1.00000 1.95583

> euro_dm_converter(dm=1)

     Euro        DM
0.5112919 1.0000000
```

In einem letzten Schritt ersetzen wir noch den Wechselkurs durch ein Objekt, dass im Code definiert wird, so dass die Funktionsdefinition etwas übersichtlicher wird.

```
> euro_dm_converter <- function(euro = dm / w,
                                dm = w * euro){
   w <- 1.95583
   round(c(Euro=euro, DM=dm), 2)
 }
> euro_dm_converter(euro=1)
```

```
Euro   DM
1.00 1.96
```

```
> euro_dm_converter(dm=1)
```

```
Euro   DM
0.51 1.00
```

1.3.3 Eigene Operatoren definieren

Operatoren wie +, −, ∗, / etc. sind in R ebenfalls Funktionen. Durch die Schreibweise `(1 + 1)` wird dies häufig nicht deutlich. Wir können die Funktionen jedoch auch wie jede andere aufrufen (d.h. mit Hilfe der runden Klammern), was die Gleichheit beider Varianten deutlich macht. Hierzu wird der Name des Operator in Anführungsstriche gesetzt.

```
> "+"(1, 2)         # entspricht 1 + 2
```

```
[1] 3
```

```
> "*"(1, 2)         # entspricht 1 * 2
```

```
[1] 2
```

Dies betrifft z.B. auch den Zugriffsoperator `[`.

```
> "["(letters, 1:3)       # entspricht letters[1:3]
```

```
[1] "a" "b" "c"
```

Auch die Dokumentation der Funktionen kann auf diese Weise aufgerufen werden.

```
> ?"+"
```

Neben den Standardoperatoren ist es in R möglich, eigene Operatoren zu definieren. Nehmen wir an, es sei unser Ziel, eine Funktion zu schreiben, die feststellt, ob die Werte eines Vektor innerhalb eines bestimmten Wertebereichs liegen. Hierzu definieren wir die Fnktion `inside` wie folgt.

```
> inside <- function (x, limits) {
          x >= min(limits) & x <= max(limits)
 }
```

Überprüfen wir nun ob, die Werte 1 bis 5 zwischen 2 und 4 liegen.

```
> inside(1:5, c(2,4))
```

```
[1] FALSE  TRUE  TRUE  TRUE FALSE
```

Das Ziel ist nun einen Operator mit derselben Funktion zu definieren, der wie das Plus- oder Multiplikationszeichen genutzt werden kann. Hierzu wird der Name des neuen Operators in Prozentzeichen gesetzt. Diese zeigen R an, dass es sich um einen Operator handelt.

```
> "%inside%" <- inside
```

Danach kann der Operator `%inside%` in derselben Weise wie ein Plus-Zeichen benutzt werden.

```
> 1:5 %inside% c(2,4)
[1] FALSE  TRUE  TRUE  TRUE FALSE
```

Nutzen wir den neu erschaffenen Operator nun für die Auswahl von Fällen aus einem Datensatz. Wir wollen aus dem Datensatz `anscombe` alle Fälle auswählen, bei denen die `x1` Werte zwischen 7 und 10 annimmt. Klassicherweise würde man wie folgt vorgehen.

```
> subset(anscombe, x1 >= 7 & x1 <= 10)
```

Mit dem neuen `%inside%` Operator ist nun auch Folgendes möglich.

```
> subset(anscombe, x1 %inside% c(7,10))
```