LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Statistische Software (R-Vertiefung)

Paul Fink, M.Sc.

Institut für Statistik
Ludwig-Maximilians-Universität München

Kontrollstrukturen und Funktionen

Kontrollstrukturen: Bedingte Anweisungen

- Die Bedingung kann ein komplexerer logischer Ausdruck sein, der entsprechend auch "und" und "oder" Verknüpfungen enthält

Beispiel

```
> x <- 5
> if ( x==3 ) { x <- x-1 } else { x <- 2*x }
> x
[1] 10
```


Kontrollstrukturen: Bedingte Anweisungen

Syntax

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

- Ausdruck 1 wir ausgewertet, wenn die Bedingung den Wert `TRUE` liefert
- Ist die Bedingung `FALSE`, so wird Ausdruck 2 ausgewertet
- Die Bedingung in dieser Form darf nicht vektorwertig sein bzw. wird dann nur das erste Element des Vektors ausgewertet

Logische Operatoren & Verknüpfungen

Die folgende Tabelle zeigt die Operationen und Funktionen für logische Vergleiche und Verknüpfungen.

`==, !=`	gleich, ungleich
`>, >=`	größer, größer gleich
`<, <=`	kleiner, kleiner gleich
`!`	Negation (nicht)
`&, &&`	und
`|, ||`	oder
`xor()`	entweder oder (ausschließend)
`TRUE, FALSE`	wahr, falsch

Die Operatoren `&` und `|` arbeiten vektorwertig.

Logische Operatoren & Verknüpfungen

Beispiele

```
> 8 > 7
[1] TRUE
> 7 < 5
[1] FALSE
> x <- 5
> (x < 10) && (x > 2)
[1] TRUE
> (x < 10) || (x > 5)
[1] TRUE
> (x %% 2 == 1 ) && (x > 5)
[1] FALSE
```


Logische Operatoren & Verknüpfungen

```
> logisch1 <- (x > 2)
> logisch2 <- (x < 10)
> logisch3 <- (2*x > 11)
> logisch4 <- (3*x < 20 )
> is.logical(logisch1)
[1] TRUE
> logisch1
[1] TRUE
> logisch2
[1] TRUE
> logisch3
[1] FALSE
> logisch4
[1] TRUE
```

Jetzt noch ein Beispiel für die vektorwertige Auswertung.

```
> c(logisch1, logisch2) & c(logisch3, logisch4)
[1] FALSE  TRUE
> c(logisch1, logisch2) | c(logisch3, logisch4)
[1] TRUE TRUE
```


Logische Operatoren & Verknüpfungen

Damit kann die Bedingung einer bedingten Anweisung ein "komplexer" logischer Ausdruck sein.

```
> x <- seq(from = -5, to = 5, by = 1 )
> x
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
> if ( (max(x) < 10) && (min(x) > -10) ) { x <- x+1 } else { x <- x-1 }
> x
 [1] -4 -3 -2 -1  0  1  2  3  4  5  6
```


Bedingte Anweisungen II

Die bedingte Anweisung

```
ifelse( test, yes, no)
```

(siehe auch `?'ifelse'`) dient der vektorwertigen Auswertung von Bedingungen. Dabei werden für die Komponenten, die `TRUE` liefern, die entsprechenden Komponenten von "yes" zurückgegeben. Für die Komponenten, die `FALSE` liefern, wird die entsprechende Komponente in "no" zurückgegeben.

Beispiel

```
> x <- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> ifelse( x < 6, x^2, x+1 )
 [1]  1  4  9 16 25  7  8  9 10 11
```

Schleifen

In R stehen die üblichen Kontrollbefehle für Schleifen zur Verfügung.

- Die `for` Schleife

  Syntax

  ```
  for ( i in Menge ){ Ausdruck }
  ```

  Beispiel

```
> for( i in 1:5 ) { print( i^2 ) }
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```


Schleifen

```
> for( i in c(2, 4, 6, 7) ) { print( i^2 ) }
[1] 4
[1] 16
[1] 36
[1] 49
```

Häufig wird folgendes Konstrukt benutzt.

```
> x <- 1:4
> for( i in 1:length(x) ) { print( x[i]^3 ) }
[1] 1
[1] 8
[1] 27
[1] 64
```

Daran ist zunächst nichts auszusetzen. Was passiert allerdings, wenn die Länge des Objekts 0 ist?

```
> x <- NULL
> length(x)
[1] 0
> for ( i in 1:length(x) ) { print( x[i]^3 ) }
numeric(0)
numeric(0)
```


Schleifen

D.h. die Schleife wird zweimal durchlaufen, da der Ausdruck `1:length(x)` den Vektor $(1, 0)$ zurückliefert:

```
> 1:length(x)
[1] 1 0
```

Deshalb ist es besser, den bereits eingeführten Befehl `seq(along=x)` zu verwenden, der einen Indexvektor erzeugt, der gleiche Länge wie `x` hat.

```
> x <- 1:4
> for ( i in seq(along=x) ){ print( x[i]^3 ) }
[1] 1
[1] 8
[1] 27
[1] 64
> x <- NULL
> for ( i in seq(along=x) ){ print( x[i]^3 ) }
```

D.h. im zweiten Fall wird erwartungsgemäß nichts zurückgeliefert.

Schleifen

- Die `while` Schleife

  Syntax

  ```
  while( Bedingung ){ Ausdruck }
  ```

  Beispiel

```
> i <- 1
> while (i < 5) {
+   print(i^2)
+   i <- i+2
+ }
[1] 1
[1] 9
```

- Die `repeat` Schleife

  Syntax

  ```
  repeat { Ausdruck }
  ```

Schleifen

Die `repeat` Schleife testet also keine Bedingung, die die Fortsetzung oder den Abbruch der Schleife steuert. Dafür ist der Progarmmierer zuständig, indem er an geeigneter Stelle einen `break` Befehl einbaut.

Beispiel

```
> i <- 1
> repeat{
+   print( i^2 )
+   i <- i+2
+   if ( i > 10 ) break
+ }
[1] 1
[1] 9
[1] 25
[1] 49
[1] 81
```

Zusätzlich steht noch der Befehl `next` zur Verfügung, um zum Beginn der Schleife zurückzuspringen.

Schleifen

Beispiel

```
> i <- 1
> repeat{
+   i <- i+1
+   if (i < 10) next
+   print(i^2)
+   if (i >= 13) break
+ }
[1] 100
[1] 121
[1] 144
[1] 169
```


Vermeiden von Schleifen, Vektorisierung

- Bei `R` handelt es sich um eine sog. Interpreter-Sprache, d.h. jeder Befehl wird erst zur Laufzeit interpretiert und ausgeführt (im Gegensatz zu kompilierten Sprachen)
- Insbsondere Schleifen mit vielen Durchläufen, insbesondere auch ineinander geschachtelte Schleifen können sehr langsam sein, z.B. für den sequentiellen Zugriff auf alle Elemente `X[i,j]` einer Matrix
- Verwende, wenn möglich, alternativ spezielle Befehle, die eine Operation auf alle Elemente eines Objekts (Liste, Vektor, Matrix) anwenden
- Es stehen die Befehle `apply()`, `lapply()`, `sapply()`, `mapply()` und `tapply()` zur Verfügung

Vermeiden von Schleifen, Vektorisierung

- Für die schnelle Berechnung von Zeilen– und Spaltensummen, sowie von Zeilen– und Spaltenmittelwerten, stehen die Funktionen `rowSums()`,`colSums()`, `rowMeans()` und `colMeans()` zur Verfügung

lapply und sapply

Diese Funktionen sind für Listen (`lapply()`) und Vektoren gedacht. Dabei liefert `lapply` als Ergebnis eine Liste zurück, auch wenn der Eingabe–Vektor als Elemente nur Skalare enthält. `sapply()` dagegen versucht, die Ergebnisse in diesem Fall als Array, Vektor, etc. auszugeben.

Syntax

```
lapply(X, FUN, ...)
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

Beispiele

lapply und sapply

```
> x <- 1:4
> lapply(X = x, FUN = function(x) x^2 )
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 9

[[4]]
[1] 16
> sapply(X = x, FUN = function(x) x^2 )
[1]  1  4  9 16

> X <- matrix(nrow = 2, ncol = 2, data = 7:10 )
> sapply(X = X, FUN = function(x){ x/2 } )
[1] 3.5 4.0 4.5 5.0
```


lapply und sapply

```
> X2 <- matrix(nrow=2, ncol=2, data=1:4)
> l1 <- list(X, X2)
> lapply(l1, FUN = function(x){ x%*%x } )
[[1]]
     [,1] [,2]
[1,]  121  153
[2,]  136  172

[[2]]
     [,1] [,2]
[1,]    7   15
[2,]   10   22
```


lapply und sapply

```
> X2 <- matrix(nrow = 2, ncol = 2, data = 1:4)
> l1 <- list(X, X2)
> lapply(X = l1, FUN = function(x){ x%*%x } )
[[1]]
     [,1] [,2]
[1,]  121  153
[2,]  136  172

[[2]]
     [,1] [,2]
[1,]    7   15
[2,]   10   22
> sapply(X = l1, FUN = function(x){ x%*%x } )
     [,1] [,2]
[1,]  121    7
[2,]  136   10
[3,]  153   15
[4,]  172   22
```

Der letzte `sapply()` Befehl zeigt ein nachvollziehbares, aber sicher nicht das gewünschte Ergebnis. Im Fall, dass die Eingabeliste komplexe Objekte (hier: Matrizen) enthält, ist also der `lapply()`

lapply und sapply

Befehl zu empfehlen. Neben `sapply()` existiert noch der verwandte Befehl `replicate()`, siehe `?replicate`.

apply für Matrizen

Syntax

```
apply(X, MARGIN, FUN, ...)
```

Die Argumente sind

- die Eingabematrix `X`
- die beizubehaltende Dimensionsnummer (1 für Zeilen, 2 für Spalten)
- die auf jede Zeile bzw. jede Spalte anzuwendende Funktion. Dabei kann es sich wiederum um bereits in R definierte

apply für Matrizen

Funktionen oder eine selbst definierte Funktion handeln. Definiert man die Funktion erst beim Aufruf von `apply()` als `FUN` Argument, so spricht man von einer *anonymen* Funktion, da diese nach Ausführen des Befehls nicht mehr bekannt ist

Beispiele

apply für Matrizen

```
> X <- matrix( nrow = 3, ncol = 2, byrow = TRUE, data = 4:9 )
> X
     [,1] [,2]
[1,]    4    5
[2,]    6    7
[3,]    8    9
> apply(X = X, MARGIN = 1, FUN = max)
[1] 5 7 9
> apply(X = X, MARGIN =  2, FUN = min)
[1] 4 5
> # liefert den Index des jeweils maximalen Elements einer Zeile zurueck
> apply(X = X, MARGIN = 1, FUN = which.max)
[1] 2 2 2
> # eigene Funktion
> apply(X = X, MARGIN = 2, FUN = function(x){ max(x)-min(x) } )
[1] 4 4
```

tapply für Datensätze

Die Idee von `tapply()` ist die Anwendung auf Datensätze zur Berechnung zusammenfassender Statistiken einer Variable bezüglich der Gruppierung nach einer anderen Variable. Im folgenden Beispiel ist `x` ein nmerischer Vektor und `fac` ein Faktor. Es soll die Summe in `x`, gruppiert nach `fac` berechnet werden.

Beispiel

```
> x <- 1:10
> fac <- factor(x = c(1, 1, 1, 1, 1, 2, 2, 2, 2, 2) )
> tapply(X = x, INDEX = fac, FUN = sum )
 1  2
15 40
```

`tapply()` liefert 2 Werte zurück, und zwar 15 für den Faktorwert 1 und 40 für den Faktorwert 2. D.h. es wurde die Summe der Zahlen von 1 bis 5 berechnet (15, Faktorstufe 1) und die Summe der Zahlen von 6 bis 10 (40, Faktorstufe 2).

mapply

Der Befehl `mapply()` ist eine Art multivariate Version von `sapply()`. Folgendes Beispiel ist der Hilfeseite `?mapply` entnommen.

Beispiel

```
> mapply(rep, times=1:4, x=4:1)
[[1]]
[1] 4

[[2]]
[1] 3 3

[[3]]
[1] 2 2 2

[[4]]
[1] 1 1 1 1
```


Funktionen

- Obwohl S-Plus und R bereits in der Standardinstallation extrem mächtige Werkzeuge zur Datenanalyse darstellen, wollen die meisten Benutzer nach einiger Zeit dieses Werkzeug ihren eigenen Bedürfnissen anpassen.
- Einer der größten Vorteile von S ist, daß jeder Benutzer das System leicht um eigene Funktionen erweitern oder bestehende Funktionen modifizieren kann.
- Der größte Teil von R selber ist in der Sprache S geschrieben.
- S ist eine vollwertige Programmiersprache in deren Design die Notwendigkeiten der statistischen Datenanalyse mit einbezogen wurden, es ist *nicht nur* eine Sprache zur Datenanalyse.

Formale Definition

```
function ( ARGLIST ) BODY
```

- Das Schlüsselwort `function` markiert den Beginn einer Funktionsdefinition.
- Die Argumente der Funktion werden durch eine mit Kommas getrennte Liste von Ausdrücken der Form `SYMBOL = AUSDRUCK` oder durch das spezielle formale Argument `...` angegeben.
- Jeder gültige R Ausdruck kann als `BODY` der Funktion verwendet werden, zumeist ist es ein durch geschwungene Klammern gruppierter Block von Anweisungen.
- In den meisten Fällen werden Funktionen Symbolen zugewiesen („bekommen einen Namen“), aber auch *anonyme* Funktionen sind an gewissen Stellen sehr praktisch (`apply()`, ...).

Funktionen aufrufen

```
moment <- function(x, n=2) {
  sum(x^n)/length(x)
}
```

- Funktionen retournieren nur eine einziges Objekt: den Wert des letzten Ausdruckes oder das Argument von `return()`. In der Praxis ist dies nicht allzu restriktiv, da das zurückgelieferte Objekt eine beliebig komplexe Liste sein kann.
- Benannte Argumente: Wenn eine Funktion aufgerufen wird, müssen die Argumente nicht in derselben Reihenfolge wie in der Funktionsdefinition angegeben werden, sondern können auch direkt mit Namen angesprochen werden, z.B. `moment(n=3, x=myx)`.
- Argumente mit Defaultwerten müssen beim Aufruf nicht spezifiziert werden, z.B. `moment(myx)`.
- Defaultwerte können Funktionen anderer Argumente sein.

Das ... Argument

- Funktionen können beliebig viele *unspezifizierte* Argumente besitzen, falls die Argumentliste das spezielle Argument ... enthält, z.B. `max(..., na.rm = FALSE)`.
- Eine zweite Verwendungsmöglichkeit ist das Weiterleiten von Argumenten an eine andere Funktion:

```
myplotfun <- function(x, y, myarg, ...)
{
  ## optionale Berechnungen mit x, y und myarg

  ## Aufruf Standard Plot Funktion
  plot(x, y, ...)
}
```

  Damit „versteht“ `myplot()` zusätzlich zu seinen eigenen Argumenten implizit auch alle Argumente von `plot()` ohne weiteren Programmieraufwand.
- `x <- list(...)` im Körper einer Funktion konvertiert alle unspezifizierten Argumente in eine benannte Liste.

Das ... Argument

Das ... Argument kann zwar an beliebig viele Funktionen weitergeben werden, jedoch bekommen diese dann alle dieselbe Argumentliste. Um mehreren Funktionen verschiedene Argumentlisten zu übergeben, empfiehlt es sich, diese als benannte Listen zu spezifizieren und die Funktionen mittels `do.call()` aufzurufen:

```
myfun <- function(x, fun2.args=NULL, fun3.args=NULL, ...)
{
  ## Berechnungen (eventuell mit fun2.args etc.)

  fun1(x, ...)
  do.call(fun2, fun2.args)       # erstes Arg entweder Funktion
  do.call("fun3", fun3.args)     # oder Zeichenkette

  ## weitere Berechnungen
}

R> myfun(x, fun2.args=list(arg1=wert))
```


Argumente zusammenfügen

Wenn eine Funktion aufgerufen wird, werden die *tatsächlichen Argumente* des Funktionsaufrufen mit den *formalen Argumenten* der Funktionsdefinition in der folgenden Reihenfolge zusammengefügt:

1. alle benannten Argumente deren Namen dem eines formalen Argumentes *exakt* entspricht
2. wenn kein ... Argument definiert wurde genügt eine eindeutige Übereinstimmung am Beginn des Namens
3. *Position* in der Argumentliste
4. *Verbleibende* tatsächliche Argumente werden Teil von ... (falls definiert).

Anmerkung: Nicht alle formalen Argumente müssen auch durch tatsächliche Argumente abgedeckt werden.

Zuweisungsoperatoren: <- und =

Außerhalb von Funktionsaufrufen haben die beiden dieselbe Bedeutung:

```
> x <- 42; x
[1] 42
> x = 42; x
[1] 42
```

Innerhalb eines Funtionsaufrufes verhält sich <- wie immer (Zuweisung von Wert an Symbol), während = den Wert einem benannten Argument zuweist:

```
> sin(x = 3)
[1] 0.14112
> x
[1] 42
> sin(x <- 3)
[1] 0.14112
> x
[1] 3
```


Lazy Evaluation

Beim Aufruf einer Funktion werden die Argumente nur geparsed, aber nicht evaluiert. Die Auswertung des Argumentes erfolgt erst, wenn das Argument zum ersten Mal *verwendet* wird:

```
> myfun <- function(x, y){
+     if(x < 0)
+         return(NaN)
+     else
+         return( y * log(x))
+ }
> myfun(-1)
[1] NaN
> myfun(2,3)
[1] 2.079442

> myfun(2)
Error in myfun(2) : Argument "y" is missing, with no default
```


Funktionen sind Objekte

Funktionen sind in S sogenannte „Objekte erster Klasse“ und können wie Daten behandelt werden, insbesonders können Funktionen

- direkt als Argumente anderer Funktionen verwendet werden,
- Funktionen als Rückgabewert liefern,
- in R selber in ihre Bestandteile zerlegt werden.

Funktionen sind Objekte

```
> apply(trees, 2, FUN = median)
 Girth Height Volume
  12.9   76.0   24.2

> f <- approxfun(x = 1:10, y = sin(1:10))
> f(3)
[1] 0.14112

> g <- function(x) x+3
> g(5)
[1] 8
> body(g)
x + 3
> body(g)[[3]]
[1] 3
> body(g)[[3]] <- 10
> g(5)
[1] 15
```