LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Programmieren mit statistischer Software

Eva Endres, M.Sc.

Institut für Statistik
Ludwig-Maximilians-Universität München

Kontrollstrukturen

Kontroll-Fluss I

- Übersicht mit `?Control`
- Grundlegende Kontroll-Fluss-Konstrukte um die Reihenfolge festzulegen, in welcher die einzelnen Befehle ausgeführt werden
  1. Choices (Bedingungen) - `if-else`
  2. Loops (Schleifen) - `for` bzw. `while`
  3. Functions (Funktionen)
  4. Ending (Beenden eines Programmes)
  5. Exceptions (Ausnahmen)

Bedingungen (choices) I

- `if-else` Anweisung

```
> # if ( condition ) {
> #   statement1
> # } else {
> #   statement2
> # }
```

- Beispiel: `sign` (Vorzeichen)-Funktion

```
> # gibt fuer x<0 den Wert -1 zurueck,
> # gibt fuer x>0 den Wert +1 zurueck
> # sollte fuer x=0 den Wert 0 zurueckgeben
> sign <- function(x) {
+   if ( x > 0 ) {  # condition1
+     y <- +1       # statement1
+   } else {        # !condition1
+     y <- -1       # statement2
+   }
+   y
+ }
```

Bedingungen (choices) II

- Funktionsaufrufe

```
> c(sign(-10), sign(pi), sign(10))
[1] -1  1  1
> sign(0)    # die Funktion funktioniert fuer x=0 noch nicht korrekt
[1] -1
```

Bedingungen (choices) III

- Verbesserte Funktion - verschachtelte `if-else`-Statements

```
> sign <- function(x) {
+   if ( x > 0 ) {          # condition1
+     y <- +1               # statement1
+   } else if ( x < 0 ) {   # condition2
+     y <- -1               # statement2
+   } else {                # !(condition1 | condition2)
+     y <- 0                # statement3
+   }
+   y
+ }
>
> # Funktionsaufruf
> c(sign(-10), sign(0), sign(10))
[1] -1  0  1
```

Bedingungen (choices) IV

- Falls die Bedingung in einer `if-else` Anweisung ein Vektor ist, wird nur das erste Element verwendet

```
> sign(c(-1.2, exp(1), 13.873))
> # [1] -1
> # Warning messages:
> # 1: In if (x > 0) { :
> #   the condition has length > 1 and only the first element will be used
> # 2: In if (x < 0) { :
> #   the condition has length > 1 and only the first element will be used
```

- Funktion `ifelse ()` - funktioniert auch für Vektoren

```
> # Alternative zu: if (condition) {statement1} else {statement2}
> # ?ifelse
> # ifelse(test, yes, no)
> # Arguments
> # test   an object which can be coerced to logical mode.
> # yes    return values for true elements of test.
> # no    return values for false elements of test.
```

Bedingungen (choices) V

```
> sign <- function(x) {
+   ifelse(test=x>0,
+          yes=+1,
+          no=ifelse(test=x<0,
+                    yes=-1,
+                    no=0)
+   )
+ }

> # Funktionsaufrufe
> c(sign(-10), sign(0), sign(10))
[1] -1  0  1
> sign(c(-10, 0, 10))
[1] -1  0  1
> sign(c(-1.2, exp(1), 13.873))
[1] -1  1  1
```

Schleifen (loops) I

- `for`-Schleife
  wenn die Anzahl an Iterationen im Vorhinein bekannt ist

```
> # for ( var in seq ) {
> #   statement1
> # }
```

  - Beispiel: Funktion, die die Elemente e eines Vektors x aufsummiert

```
> adder <- function(x) {
+   y <- 0
+   for ( e in x ) {  # var in seq
+     #   print(y)   # zur Veranschaulichung eingefuegt
+     #   print(e)
+     y <- y + e      # statement1
+   }
+   y
+ }
```

Schleifen (loops) II

```
> adder(x=1:10)
[1] 55
> adder(x=c(1,2,4,8,16))
[1] 31
> adder(x=NA)
[1] NA
> adder(x=NULL)
[1] 0
> adder(x=numeric(length=0))
[1] 0
```

Schleifen (loops) III

- Alternative Programmierung - Schleife mit Zählvariable `i`

```
> # mit seq_along(x)
> adder2 <- function(x) {
+   y <- 0
+   for ( i in seq_along(x) ) {  # var in seq
+     #      print(y)            # zur Veranschaulichung eingefuegt
+     #      print(x[i])
+     y <- y + x[i]       # statement1
+   }
+   y
+ }
```

Schleifen (loops) IV

```
> adder2(x=1:10)
[1] 55
> adder2(x=c(1,2,4,8,16))
[1] 31
> adder2(x=NA)
[1] NA
> adder2(x=NULL)
[1] 0
> adder2(x=numeric(length=0))
[1] 0
```

Schleifen (loops) V

```
> # mit 1:length(x)
> adder3 <- function(x) {
+   y <- 0
+   for ( i in 1:length(x) ) {  # var in seq
+     #      print(y)           # zur Veranschaulichung eingefuegt
+     #      print(x[i])
+     y <- y + x[i]       # statement1
+   }
+   y
+ }
```

Schleifen (loops) VI

```
> adder3(x=1:10)
[1] 55
> adder3(x=c(1,2,4,8,16))
[1] 31
> adder3(x=NA)
[1] NA
> adder3(x=NULL)
numeric(0)
> adder3(x=numeric(length=0))   # kein sinnvolles Ergebnis
numeric(0)
```

Schleifen (loops) VII

```
> x <- c(1,2,4,8,16)
> x <- numeric(length=0)
>
> 1:length(x)
[1] 1 0
> seq_along(x)
integer(0)
> # ist dasselbe, solange x mindestens ein gueltiges Element hat
> # wenn x leer sein kann, ist seq_along(x) die sicherere Variante
```

Schleifen (loops) VIII

- `while`-Schleife
  wenn die benötigte Anzahl an Iterationen von einer Berechnung innerhalb der Schleife abhängt

```
> # while ( condition ) {
> #   statement1
> # }
```

  - Beispiel: Funktion, die Zufallszahlen zwischen 1 und 100 aufsummiert, bis die Summe größer ist als ein vorgegebener Wert x

```
> radder <- function(x) {
+   y <- 0    # Initialisierung von y
+   i <- 0    # Initialisierung der Zaehlvariable i
+   while ( y <= x ) {          # condition
+     y <- y + sample(100, 1)   # statement1
+     i <- i + 1    # Erhoehung der Zaehlvariable i um 1

+   }
+   c(x = x, i = i, y = y)
+ }
```

Schleifen (loops) IX

```
> set.seed(1234)
> radder(20)
```

```
 x  i  y
20  2 75
```

```
> radder(831)
```

```
  x   i   y
831  16 837
```

- Vorsicht: Es sollten keine unendlichen Schleifen erzeugt werden! Abbruch mit ESC möglich (oder STOP-Button)

```
> # x <- 1
> # while ( x > 0 ) {
> #   x <- x + 1
> # }
```

Beenden einer Funktionsausführung I

- Die `return`-Anweisung beendet die aktuelle Funktion.

```
> # return(value)
```

- Beispiel für unendliche Schleife

```
> # radder(Inf)
```

- Erweiterung der Funktion `radder`

Beenden einer Funktionsausführung II

```
> # Abfangen von ungewuenschten Eingaben mit if-return
> radder <- function(x) {
+   if ( x == Inf ) {
+     return(c(x = x, i = Inf, y = Inf))
+   }

+   y <- 0
+   i <- 0
+   while ( y <= x ) {
+     y <- y + sample(100, 1)
+     i <- i + 1
+   }
+   c(x = x, i = i, y = y)
+ }
>
> radder(Inf)
  x   i   y
Inf Inf Inf
```

Beenden einer Funktionsausführung III

```
> # oder
> radder <- function(x) {
+   if ( x == Inf ) {
+     return("Fehler: x=Inf stellt keine sinnvolle Eingabe dar")
+   }

+   y <- 0
+   i <- 0
+   while ( y <= x ) {
+     y <- y + sample(100, 1)
+     i <- i + 1
+   }
+   c(x = x, i = i, y = y)
+ }
>
> radder(Inf)
[1] "Fehler: x=Inf stellt keine sinnvolle Eingabe dar"
```

Beenden einer Funktionsausführung IV

- `if-return` ermöglicht detaillierte Fehlerbeschreibungen
- Alternative zu `stopifnot()` bzw. `stop()`
- Fehlermeldung dabei vom Programmierer nicht beeinflussbar

Ausnahmen (exceptions) I

- Exception handling

```
> # tryCatch(expr, ...)
```

- Beispiel: Funktion, die die Summe von n Zufallszahlen zwischen -0.5 und 1 berechnet mit Fehlermeldung, falls die Summe kleiner als 0 ist

```
> f <- function(n) {
+   x <- runif(n, min = -0.5, max = 1)
+   if ( sum(x) < 0 ) {
+     stop("Sorry, sum(x) < 0")
+   }
+   sum(x)
+ }
>
> set.seed(1234)
> f(10)
[1] 2.338396
> f(10)
[1] 1.819506
```

Ausnahmen (exceptions) II

- Verwendung dieser Funktion innerhalb einer anderen Funktion: Simulation mit m Wiederholungen Die Funktion soll nicht stoppen, wenn ein Fehler auftritt. Stattdessen soll der Fehler ignoriert werden und die restlichen Iterationen ausgeführt werden.

```
> g <- function(n, m) {
+   y <- numeric(length = m)

+   for ( i in seq(length = m) ) {
+     y[i] <- tryCatch(f(n),
+                      error = function(e) {
+                        warning("Error in f(); using NA instead.")
+                        NA_real_
+                      })
+   }
+   y
+ }
```

Ausnahmen (exceptions) III

```
> set.seed(1222)
> g(10, 10)
 [1] 1.344216 1.628098 3.296667 3.213688 1.587960 2.835474 3.933308
 [8] 4.026333 2.210362 3.472786

> g(10, 10)
Warning in value[[3L]](cond): Error in f(); using NA instead.
Warning in value[[3L]](cond): Error in f(); using NA instead.
 [1] 2.1004055 3.0961110 2.4744317 3.3155101 1.3407386 4.8220214        NA
 [8] 2.5773327 0.9176143        NA
```

- Besonders nützlich bei langen Simulationen.
- Ebenfalls in manchen Situationen lohnenswert ist `try()`