Programmierung und Deskriptive Statistik

BSc Psychologie WiSe 2024/25

Belinda Fleischmann

(10) Kontrollstrukturen, Debugging und Datensimulationen in R

Kontrollstrukturen

Debugging

Datensimulationen

Programmierübungen und Selbstkontrollfragen

Kontrollstrukturen

Debugging

Datensimulationen

Programmierübungen und Selbstkontrollfragen

Motivation

Programmiercode wird streng sequentiell Befehl für Befehl ausgeführt.

Manchmal möchten wir von dieser rein sequentiellen Befehlsreiehenfolge abweichen.

Die prinzipiellen Werkzeuge dafür sind **Kontrollstrukturen**. Dazu gehören `if`-statements, `switch`-statements und Schleifen mit `for`, `while` oder `repeat`.

Kontrollstrukturen: `if`-statements

if-statements

```
if (Bedingung) {
  TrueAktion        # Befehl, der ausgeführt wird, falls Bedingung TRUE ist
}
```

- Wenn `Bedingung` `TRUE` ist, wird `TrueAktion` ausgführt.
- Wenn `Bedingung` `FALSE` ist, wird `TrueAktion` nicht ausgeführt.

if-else-statements

```
if (Bedingung) {
  TrueAktion        # Befehl, der ausgeführt wird, falls Bedingung TRUE ist
} else {
  FalseAktion       # Befehl, der ausgeführt wird, falls Bedingung FALSE ist
}
```

- Wenn `Bedingung` `TRUE` ist, wird `TrueAktion` ausgführt.
- Wenn `Bedingung` `FALSE` ist, wird `FalseAktion` ausgeführt.

Kostrollstrukturen: `if`-statements

Beispiele

```
x <- 3
if (x > 0) {
  print("x ist größer als 0")
}
```

```
[1] "x ist größer als 0"
```

```
y <- -3
if (y > 0){
  print("y ist größer als 0")
} else{
  print("y ist nicht größer als 0")
}
```

```
[1] "y ist nicht größer als 0"
```

Wiederholung: Logischer Operatoren

- Die Boolesche Algebra und R kennen zwei *logische Werte*: TRUE und FALSE
- Bei Auswertung von Relationsoperatoren ergeben sich logische Werte

Relationsoperator	Bedeutung
$==$	Gleich
$!=$	Ungleich
$<$, $>$	Kleiner, Größer
$<=$, $>=$	Kleiner gleich, Größer gleich
$\vert$	ODER
$\&$	UND

- $<, <=, >, >=$ werden zumeist auf numerische Werte angewendet.
- $==, !=$ werden zumeist auf beliebige Datenstrukturen angewendet.
- $\vert$ und $\&$ werden zumeist auf logische Werte angewendet.
- $\vert$ implementiert das inklusive *oder*. Die Funktion xor() implementiert das exklusive ODER.

Kontrollstrukturen: `if`-statements

Beispiele

```
x <- 3
y <- 2

# Logisches UND/ODER
if (x > 0 | y > 0) {
  print("Es sind beide, oder eine der beiden Variablen größer 0.")
} else{
  print("Keine der Variablen ist größer 0.")
}
```

```
[1] "Es sind beide, oder eine der beiden Variablen größer 0."
```

```
# Logisches UND
if (x > 0 & y > 0) {
  print("x und y sind beide größer 0.")
} else{
  print("Es sind nicht beide Variablen x und y größer 0, aber vielleicht eine der beiden.")
}
```

```
[1] "x und y sind beide größer 0."
```

```
# Exklusives ODER
if (xor(x > 0, y > 0)){
  print("Genau eine der 2 Variablen x und y ist größer 0, aber nicht beide.")
} else{
  print("Es sind entweder keine der Variablen x und y oder beide größer 0.")
}
```

```
[1] "Es sind entweder keine der Variablen x und y oder beide größer 0."
```

Kontrollstrukturen: `switch`-statements

Motivation

Kombinierte `if-else` -statements können leicht unübersichtlich werden.

```
x <- 2
if (x == 1){
  print("Aktion 1")
} else if(x == 2){
  print("Aktion 2")
} else if(x == 3){
  print("Aktion 3")
} else if(x == 4){
  print("Aktion 4")
}
```

```
[1] "Aktion 2"
```

switch-statement mit Integer

```
x <- 2
switch(
  x,                            # switch Variable
  print("Aktion 1"),            # 1. Aktion
  print("Aktion 2"),            # 2. Aktion
  print("Aktion 3"),            # 3. Aktion
  print("Aktion 4")             # 4. Aktion
)
```

```
[1] "Aktion 2"
```

Kontrollstrukturen: `switch`-statements

switch-statement mit Character

```
x <- "a"
switch(
  x,                              # switch Variable
  a = print("Aktion 1"),          # 1. Aktion
  b = print("Aktion 2"),          # 2. Aktion
  c = print("Aktion 3"),          # 3. Aktion
  d = print("Aktion 4")           # 4. Aktion
)
```

```
[1] "Aktion 1"
```

for-Schleifen

```
for (item in sequenz){
  zu_wiederholende_Aktion          # Aktion, die wiederholt werden soll
}
```

Beispiel

```
for (i in 1:3) {
  print(i)                         # Aktion, die wiederholt werden soll
}
```

```
[1] 1
[1] 2
[1] 3
```

while-Schleifen

while-Schleifen iterieren Codeabschnitte basierend auf einer Bedingung.

```
while (Bedingung) {
  TrueAktion            # TrueAktion wird ausgeführt, solange Condition == TRUE
}
```

Beispiel

```
i <- 5
while (i < 11) {
  print(i)
  i <- i + 1
}
```

```
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

repeat-Schleifen

repeat-loops wiederholen Codeabschnitte bis zu einem `break` Befehl

```
repeat {
  TrueAktion            # Aktion wird ausgeführt, bis ein break Befehl evaluiert wird
}
```

Beispiel

```
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i == 5) {
    break
  }
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
```

Kontrollstrukturen

Debugging

Datensimulationen

Programmierübungen und Selbstkontrollfragen

Debugging

"Als Debuggen (dt. Entwanzen) oder Fehlerbehebung bezeichnet man in der Informatik den Vorgang, in einem Computerprogramm Fehler oder unerwartetes Verhalten zu diagnostizieren und zu beheben. Die Suche von Programmfehlern (sogenannten Bugs) ist eine der wichtigsten und anspruchsvollsten Aufgaben der Softwareentwicklung und nimmt einen großen Teil der Entwicklungszeit in Anspruch."

Wikipedia

printf debugging

TODO

Debugging mit `browser()`

Die `R base` Funktion `browser()` erlaubt das Pausieren der Exekution eines Skripts und Inspektion der aktuellen *environment*.

Beispiel

```
# Beispiel 1
erste_variable <- 1
zweite_variable <- 3
ergebnis <- c()
browser()                                    # Pausiert Skript
print("Das Ergebnis ist: ", ergebnis)
ergebnis <- erste_variable + zweite_variable
browser()

# Beispiel 2
for (i in 1:5) {
  print(i + 2)
  browser()
}
```

Über das Argument `expr` kann auch eine Bedingung als boolesche Operation spezifiziert werden.

Mit Enter wird die Exekution fortgeführt.

Mit Q wird der browser beendet.

Kontrollstrukturen

Debugging

Datensimulationen

Programmierübungen und Selbstkontrollfragen

Datensimulation

```r
# Seed setzen
set.seed(5)                                          # Startwert für den Zufallsgenerator setzen

# Simulationsparameter
n      <- 50                                         # Proband:innnen pro Gruppe
mu     <- c(                                         # Erwartungswertparameter
  18, 12,                                            # Pre und Post der Gruppe Klassisch
  19, 14)                                            # Pre und Post der Gruppe Online
sigsqr <- 3                                          # Varianzparameter (gleich für alle Gruppen)

# Datensimulation
D  <- data.frame(
        "Bedingung" = c(
          rep("Klassisch", n), rep("Online", n)), # n-mal "Klassisch", n-mal "Online"
        "Pre BDI" = c(
          round(rnorm(n, mu[1], sqrt(sigsqr))),    # n Zufallswerte aus Normalveritung mit mu[1]
          round(rnorm(n, mu[3], sqrt(sigsqr)))),   # n Zufallswerte aus Normalveritung mit mu[3]
        "Post BDI" = c(
          round(rnorm(n, mu[2], sqrt(sigsqr))),    # n Zufallswerte aus Normalveritung mit mu[2]
          round(rnorm(n, mu[4], sqrt(sigsqr))))    # n Zufallswerte aus Normalveritung mit mu[4]
        )

# Datenspeicherung
fname  <- file.path(data_path, "psychotherapie_datensatz.csv")
write.csv(D, file = fname)
```

Kontrollstrukturen

Debugging

Datensimulationen

Programmierübungen und Selbstkontrollfragen

Programmierübungen und Selbskontrollfragen