Programmierung und Deskriptive Statistik

BSc Psychologie WiSe 2024/25

Belinda Fleischmann

Datum	Einheit	Thema	Form
15.10.24	R Grundlagen	(1) Einführung	Seminar
22.10.24	R Grundlagen	(2) R und Visual Studio Code	Seminar
29.10.24	R Grundlagen	(2) R und Visual Studio Code	Übung
05.11.24	R Grundlagen	(3) Vektoren, (4) Matrizen	Seminar
12.11.24	R Grundlagen	(5) Listen und Dataframes	Seminar
	Leistungsnachweis 1		
19.11.24	R Grundlagen	(6) Datenmanagement	Seminar
26.11.24	R Grundlagen	(2)-(6) R Grundlagen	Übung
03.12.24	Deskriptive Statistik	(7) Häufigkeitsverteilungen	Seminar
10.12.24	Deskriptive Statistik	(8) Verteilungsfunktionen und Quantile	Seminar
	Leistungsnachweis 2		
17.12.24	Deskriptive Statistik	(9) Maße der zentralen Tendenz und Datenvariabilität	Seminar
	Weihnachtspause		
07.01.25	**R Grundlagen**	**(10) Strukturiertes Programmieren: Kontrollfluss, Debugging**	**Seminar**
14.01.25	Deskriptive Statistik	(11) Anwendungsbeispiel	Übung
	Leistungsnachweis 3		
21.01.25	Deskriptive Statistik	(11) Anwendungsbeispiel	Seminar
28.01.25	Deskriptive Statistik	(11) Anwendungsbeispiel, Q&A	Seminar

(10) Strukturiertes Programmieren:

Kontrollfluss und Debugging

Kontrollstrukturen

Debugging

Programmierübungen

Kontrollstrukturen

Debugging

Programmierübungen

Motivation

Programmiercode wird streng sequentiell Befehl für Befehl ausgeführt.

Manchmal möchten wir von dieser rein sequentiellen Befehlsreihenfolge 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

Ermöglicht die Ausführung eines spezifischen Codeblocks, wenn eine bestimmte Bedingung den Wert `TRUE` hat.

```
# Pseudocode
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

Bietet alternative Ausführungsmöglichkeiten, je nachdem, ob eine Bedingung den Wert `TRUE` oder `FALSE` hat.

```
# Pseudocode
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."
```

Motivation

Wenn wir mehrere ähnliche Bedingungen überprüfen möchten, von denen immer nur eine zutrifft, können kombinierte `if-else`-Statements schnell unübersichtlich und schwer lesbar werden. In solchen Fällen bietet sich das `switch`-Statement als strukturierte und kompakte Alternative an.

```
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

Vereinfacht die Auswahl zwischen mehreren Alternativen, ohne eine lange Verkettung von `if-else`-Statements zu verwenden.

```
# Pseudocode
switch(
  x,                              # Variable, die geprüft wird
  Aktion 1,                       # Ausführung, wenn x == 1
  Aktion 2,                       # Ausführung, wenn x == 2
  Aktion 3,                       # Ausführung, wenn x == 3
  Aktion 4                        # Ausführung, wenn x == 4
)
```

Kontrollstrukturen: `switch`-statements

switch-statement mit Integer

Wenn der Input numerisch gegeben ist (z. B. `x <- 2`), wird die Position des entsprechenden Falls ausgewählt.

```
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"
```

switch-statement mit Character

Wenn der Input als character gegeben ist (z. B. `x <- "a"`), wird der Fall anhand des Namens ausgewählt.

```
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

Wiederholt einen Codeblock für jedes Element in einer Sequenz oder Sammlung.

```
# Pseudocode
for (item in sequenz) {
  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

Führt einen Codeblock wiederholt aus, solange eine Bedingung den Wert `TRUE` hat.

```
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

Wiederholt einen Codeblock unbegrenzt oft, bis eine `break`-Anweisung die Schleife beendet.

```
# Pseudocode
repeat {
  Aktion                # 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

Programmierübungen

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

print debugging

Gezieltes Einfügen von `print`-Anweisungen im Code an den Stellen, an denen bestimmte Variablenwerte oder Zwischenergebnisse inspiziert werden sollen. Diese Methode hilft, die Ausführungsschritte und den aktuellen Zustand des Programms besser zu verstehen.

Beispiel

```
# Fehlerhaftes Skript
meine_Liste    <- list("1", 2, "drei", TRUE)
listen_element <- meine_Liste[2]
rechenergebnis <- exp(listen_element)
print(rechenergebnis)
```

```
Error in exp(listen_element): non-numeric argument to mathematical function
```

```
Error: object 'rechenergebnis' not found
```

Problem: Die Variable `listen_element` scheint `non-numeric` zu sein.

print debugging

Beispiel fortgeführt

Problemanalyse:Im Folgenden wird die Variable `listen_element` über `print()` bzw. `cat()` inspiziert.

```
# Fehlerhaftes Skript mit print-debugging
meine_Liste    <- list("1", 2, "drei", TRUE)
listen_element <- meine_Liste[2]
cat(                                          # print debugging
  "listen_element:", listen_element,          # Ausgabe des Werts der Variable
  "Typ:", typeof(listen_element)              # Ausgabe des Typs der Variable
)
rechenergebnis <- exp(listen_element)
print(rechenergebnis)
```

```
[1] "listen_element: 2 Typ: list"
```

```
Error in exp(listen_element): non-numeric argument to mathematical function
```

```
Error: object 'rechenergebnis' not found
```

Ergebnis: `listen_element` ist vom Typ `list`, obwohl wir den Wert eines einzelnen Elements der Liste erwarten.

Fehlerursache: Falsche Indizierung der Liste: Es wurde `[]` verwendet, statt der korrekten `[[]]`, um auf das Element der Liste zuzugreifen. `[]` gibt eine Liste zurück, während `[[]]` den tatsächlichen Wert des Elements extrahiert.

Debugging mit `browser()`

Die `R base` Funktion `browser()` pausiert die Ausführung des Programms und erlaubt das Inspizieren der aktuellen Umgebung (*Global Environment*). Variablenwerte, Funktionen und andere Elemente können interaktiv geprüft werden, um den Ursprung von Fehlern einzugrenzen.

Beispiele

```
# Beispiel 1
var_1    <- 1
var_2    <- 3
summe    <- c()
browser()                                  # Pausiert Skript
print("Das Ergebnis ist: ", ergebnis)
ergebnis <- var_1 + var_2
browser()                                  # Pausiert Skript

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

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

Programmierübungen

Programmierübungen

1. Erstelle R-Code, der überprüft, ob eine Zahl positiv, negativ oder null ist. Gib eine entsprechende Nachricht aus.
2. Erstelle R-Code, in dem eine Eingabe für die Wahl eines Wochentages akzeptiert und den vollständigen Namen des Wochentages ausgegeben wird (z.B., 1 für Montag, 2 für Dienstag, usw.).
3. Schreibe ein R-Skript, das die Zahlen von 1 bis 10 ausgibt. Jede Zahl soll in einer neuen Zeile stehen.
4. Der untenstehende Code soll in jeder Iteration der Schleife das Quadrat eines Elements aus dem Vektor `zahlen` ausgeben (d.h. in der ersten Iteration `1`, in der zweiten `4`, usw.). Stattdessen werden in jeder Iteration fünf Werte ausgegeben. Deine Aufgabe ist es, den Fehler zu finden. Verwende dazu einmal `print()` und einmal `browser()`, um den Fehler zu identifizieren.

```r
zahlen <- c(1, 2, 3, 4, 5)
for (zahl in zahlen) {
  quadrat <- zahlen^2
  print(quadrat)
}
```