Programmierung und Deskriptive Statistik

BSc Psychologie WiSe 2025/26

Belinda Fleischmann und Dirk Ostwald

	Gruppe 1/2	Gruppe 3	Format	Thema
1	Mi, 15.10.	Do, 16.10.	Seminar	(1) Grundbegriffe der Informatik
2	Mi, 22.10.	Do, 23.10.	Seminar	(2) Arithmetik und Variablen
3	Mi, 29.10.	Do, 30.10.	Übung	(2) Arithmetik und Variablen
4	Mi, 05.11.	Do, 06.11.	Seminar	(3) Vektoren und Matrizen
5	Mi, 12.11.	Do, 13.11.	Übung	(3) Vektoren und Matrizen
6	Mi, 19.11.	Do, 20.11.	Seminar	(4) Listen und Dataframes
7	Mi, 26.11.	Do, 27.11.	Übung	(4) Listen und Dataframes
8	Mi, 03.12.	Do, 04.12.	Seminar	(5) Datenmanagement
9	Mi, 10.12.	Do, 11.12.	Übung	(5) Datenmanagement
	Mo, 15.12		Abgabe	*Individualleistung (1/2)*
10	**Mi, 17.12.**	**Do, 18.12.**	**Seminar**	**(6) Strukturiertes Progr.: Kontrollfluss, Debugging**
11	Mi, 07.01.	Do, 08.01.	Seminar	(7) Häufigkeitsverteilungen
12	Mi, 14.01.	Do, 15.01.	Übung	(7) Häufigkeitsverteilungen
13	Mi, 21.01.	Do, 22.01.	Seminar	(8) Maße der zentralen Tendenz und Datenvariabilität
14	Mi, 28.01.	Do, 29.01.	Übung	(8) Maße der zentralen Tendenz und Datenvariabilität
	Mo, 02.02		Abgabe	*Individualleistung (2/2)*

(6) Strukturiertes Programmieren:

Kontrollfluss und Debugging

Kontrollfluss

Debugging

Programmierübungen

Kontrollfluss

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**:

- **Konditionale Kontrollstrukturen** legen fest, dass Befehle nur ausgeführt werden, wenn bestimmte Bedingungen erfüllt sind (`if`, `switch`).
- **Iterative Kontrollstrukturen** wiederholen Codeabschnitte in Schleifen (`for`, `while`, `repeat`).

Kontrollstrukturen: `if`-statements

if-statements

Ermöglicht die Ausführung eines spezifischen Codeblocks, wenn eine Bedingung `TRUE` ist.

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

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

if-else-statements

Bietet alternative Ausführungsmöglichkeiten, je nachdem, ob eine Bedingung `TRUE` oder `FALSE` ist.

```
# 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` ausgeführt.
- Wenn `Bedingung` `FALSE` ist, wird `FalseAktion` ausgeführt.

Kontrollstrukturen: `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: Logische 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.

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."
```

Häufige Fehler bei Bedingungen

Bei if-statements muss die Bedingung einem einzigen logischen Wert entsprechen.

```
# Bedingung ist kein logischer Wert
if ("x") { print(1) }
# Error: argument is not interpretable as logical

# Bedingung ist NA
if (NA) { print(1) }
# Error: missing value where TRUE/FALSE needed

# Bedingung hat Länge > 1
if (c(TRUE, FALSE)) { print(1) }
# Error: the condition has length > 1
```

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"
```

Kontrollstrukturen: `switch`-statements

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

Wenn der Input numerisch gegeben ist, 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"
```

Wenn der Input als character gegeben ist, 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"
```

Anmerkung:

- Jedes `if`-statement lässt sich äquivalent als `switch`-statement formulieren und umgekehrt. `if`-statements sind in der Praxis häufiger. Bei vielen Fallunterscheidungen können `switch`-statements die Lesbarkeit verbessern.

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
```

Hinweise zur Verwendung

Die Aktion innerhalb der Schleife muss nicht auf das `item` Bezug nehmen:

```
for (i in 1:3) {
  print("a")
}
```

```
[1] "a"
[1] "a"
[1] "a"
```

Bei iterativem Speichern sollte die Datenstruktur vorab in ihrer endgültigen Größe erstellt werden, um den Speicherplatz zu reservieren.

```
# Iterative Auffüllen eines Vektors mit Mittelwerten
ns     <- 3                       # Anzahl an Simulationen
x_bar <- rep(NaN, ns)             # Speicherstruktur

for (i in 1:ns) {                 # Iterationsindizes sind typischerweise i,j,k
  x        <- rnorm(12)           # Realisierung von 12 Z-Variablen
  x_bar[i] <- mean(x)             # Mittelwert der i-ten Realisierung
  print(x_bar)                    # Iterative Anzeige zur Demonstration
}
```

```
[1] 0.177691      NaN      NaN
[1]  0.17769103 -0.08225113         NaN
[1]  0.17769103 -0.08225113 -0.21261356
```

Kontrollstrukturen: `for`-Schleifen

Bei Vektoren mit variablen Einträgen ist die Funktion `seq_along()` hilfreich, die Indizes von 1 bis zur Länge eines Vektors erzeugt.

```
mu    <- c(0,5,50)                  # Drei Erwartungswertparameter
X_bar <- rep(NaN, ns)               # Speicherstruktur

# Simulationsiterationen
for(i in seq_along(mu)){            # Iterationsindizes sind typischerweise i,j,k
  X         <- rnorm(12, mu[i], 1)  # Realisierung von 12 N(\mu,1) Variablen
  X_bar[i]  <- mean(X)              # Mittelwert der i-ten Realisierung
  print(X_bar)                      # Anzeige zur Demonstration
}
```

```
[1] -0.04830394         NaN         NaN
[1] -0.04830394  4.54174681         NaN
[1] -0.04830394  4.54174681 49.79257455
```

Dies ist robuster als `1:length(mu)`, da `seq_along()` bei leeren Vektoren korrekt einen leeren Vektor zurückgibt.

Geschachtelte Schleifen

Bei geschachtelten `for`-Schleifen werden für jede Iteration der äußeren Schleife alle Iterationen der inneren Schleife durchlaufen.

```
n_i <- 2                          # Anzahl der Iterationen äußere Schleife
n_j <- 3                          # Anzahl der Iterationen innere Schleife
for(i in 1:n_i){                  # Äußere Schleife
  for (j in 1:n_j){               # Innere Schleife
    print(c(i,j))                 # Anzeigen der Iterationsindizes [i,j]
    }
}
```

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

Beispiel: Matrix befüllen

Geschachtelte Schleifen eignen sich zum Befüllen von Matrizen.

```
# Matrix zeilenweise mit Spaltenindizes befüllen
n <- 4                                  # Zeilenanzahl
m <- 5                                  # Spaltenanzahl
A <- matrix(rep(NaN, n*m), nrow = n)    # Matrixvorallokation

for (i in 1:n) {                        # Äußere Schleife: iterieren der Zeilenindizes
  for (j in 1:m) {                      # Innere Schleife: iterieren der Spaltenindizes
    A[i, j] <- j                        # Element = Spaltenindex
  }
}
print(A)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    1    2    3    4    5
[3,]    1    2    3    4    5
[4,]    1    2    3    4    5
```

Beispiel: Matrix befüllen (fortgeführt)

```
# Matrix zeilenweise mit Zeilenindizes befüllen

n <- 4                                  # Zeilenanzahl
m <- 5                                  # Spaltenanzahl
B <- matrix(rep(NaN, n*m), nrow = n)    # Matrixvorallokation

for (i in 1:n) {                        # Äußere Schleife: iterieren der Zeilenindizes
  for (j in 1:m) {                      # Innere Schleife: iterieren der Spaltenindizes
    B[i, j] <- i                        # Element = Zeilenindex
  }
}
print(B)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    2    2    2    2    2
[3,]    3    3    3    3    3
[4,]    4    4    4    4    4
```

Beispiel: Matrix befüllen (fortgeführt)

Kombination von geschachtelten Schleifen mit `if`-statement:

```
# Eine Matrix mit Spaltenindizes befüllen, wenn Spaltenindex >= Zeilenindex

n <- 4                                  # Zeilenanzahl
m <- 5                                  # Spaltenanzahl
C <- matrix(rep(NaN, n*m), nrow = n)    # Matrixvorallokation

for (i in 1:n) {                        # Äußere Schleife: iterieren der Zeilenindizes
  for (j in 1:m) {                      # Innere Schleife: iterieren der Spaltenindizes
    if (j >= i) {                       # Bedingung: Spaltenindex >= Zeilenindex
      C[i, j] <- j                      # Element = Spaltenindex für j >= i
    } else {
      C[i, j] <- 0                      # Element = 0 für j < i
    }
  }
}
print(C)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    0    2    3    4    5
[3,]    0    0    3    4    5
[4,]    0    0    0    4    5
```

Kontrollstrukturen: `while` -Schleifen

while-Schleifen

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

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

Kontrollfluss

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.

Beispiel (fortgeführt)

Problemanalyse: Im Folgenden wird die Variable `listen_element` über den Printbefehl `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
ergebnis <- 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.

Kontrollfluss

Debugging

Programmierübungen

Programmierübungen

1. Erstellen Sie R-Code, der überprüft, ob eine Zahl positiv, negativ oder null ist, und geben Sie eine entsprechende Nachricht aus.
2. Erstellen Sie R-Code, der eine numerische Eingabe für einen Wochentag akzeptiert und den vollständigen Namen des Wochentages ausgibt (z.B. 1 für Montag, 2 für Dienstag, usw.).
3. Schreiben Sie ein R-Skript, das die Zahlen von 1 bis 10 ausgibt. Jede Zahl soll in einer neuen Zeile stehen.
4. Schreiben Sie ein R-Skript mit einer `while`-Schleife, das beginnend bei 1, so lange aufeinanderfolgende Zahlen ausgibt, bis die Summe aller ausgegebenen Zahlen größer als 20 ist.
5. Erstellen Sie mithilfe geschachtelter `for`-Schleifen eine 3x3 Matrix, deren Elemente die Summe aus Zeilen- und Spaltenindex enthalten.
6. 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. Ihre Aufgabe ist es, den Fehler zu finden. Verwenden Sie dazu `print()` und `browser()` als Debugging-Methoden, um den Fehler zu identifizieren.

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