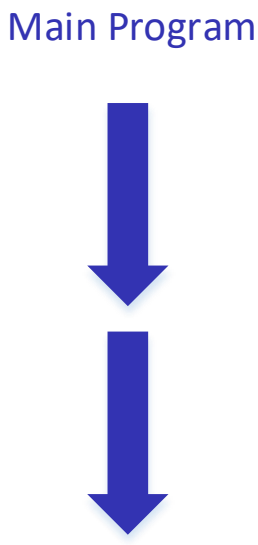
Management of Information Systems

Session 2: Kontrollstrukturen in R

Gliederung

Kontrollstrukturen

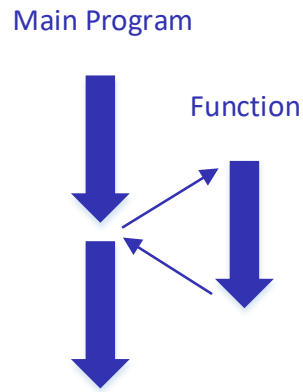
- Code hat eine feste Reihenfolge, in der dieser ausführt wird
- Standardmäßig linear "von oben nach unten"
- Kontrollstrukturen können eine Ausführung festlegen
  1. Funktionen
  2. Bedingte Anweisungen
  3. Schleifen



Gliederung

Funktionen

- Groupieren eine Folge von Anweisungen in einer “Aufgabe”
- Verlassen die aktuelle Auswertung, um eine vordefinierte Reihe an Anweisungen auszuführen
- Verhindern Codewiederholungen
- Code wird dadurch verständlicher und leichter zu warten
- Ermöglichen eine Ausweitung über die vordefinierten Funktionen hinweg



Funktionen

- Syntax von Funktionen

```
function_name <- function(argument1, argument2, ...) {
  function_body
  return(value)
}
```

- Aufruf mit `function_name(argument1, argument2, ...)`
- Rückgabewert wird mit `return(value)` gesetzt (wenn nicht vorhanden, dann der zuletzt ausgewertete Ausdruck)
- Reihenfolge der Argumente relevant
- Klammern können bei nur einer Anweisung weggelassen werden
- Werte in der Funktionen werden nicht automatisch am Bilschirm ausgegeben
  - Ausgabe manuell via `print(...)`

Funktionen

Beispiele

```
my_first_function <- function() {
  print("hello world")
}
my_first_function()
```

```
## [1] "hello world"
```

```
square <- function(x) {
  return(x * x)
}
square(10)
```

```
## [1] 100
```

Übung: Funktionen

Schreibe eine Funktion `even(n)`, die alle geraden Zahlen zwischen 1 und n ausgibt und dabei n als Parameter hat.

Übung: Funktionen

Schreibe eine Funktion `sum_to_n(n)`, die die Summe der Zahlen 1 bis n berechnet und n als Parameter hat, indem

1. eine Vektorfunktion benutzt wird
2. die Gaußsche Summenformel benutzt wird

Hinweis: Gaußsche Summenformel

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Funktionen

- Variablen werden innerhalb einer Funktion nur lokal definiert
- Auf diese besteht nach Verlassen der Funktion kein Zugriff mehr
- Beispiel

```
x <- "A"
g <- function(x) {
  x <- "B"
  return(x)
}
x <- "C"
```

- Was sind die Werte von `x` und `g(x)`?

```
g(x)
```

```
## [1] "B"
```

```
x
```

```
## [1] "C"
```

Übung: Funktionen

Gegeben sei folgende Funktion:

```
func <- function(x) {
  y <- x * 2
  x <- "I am a function."
  return(x)
}
```

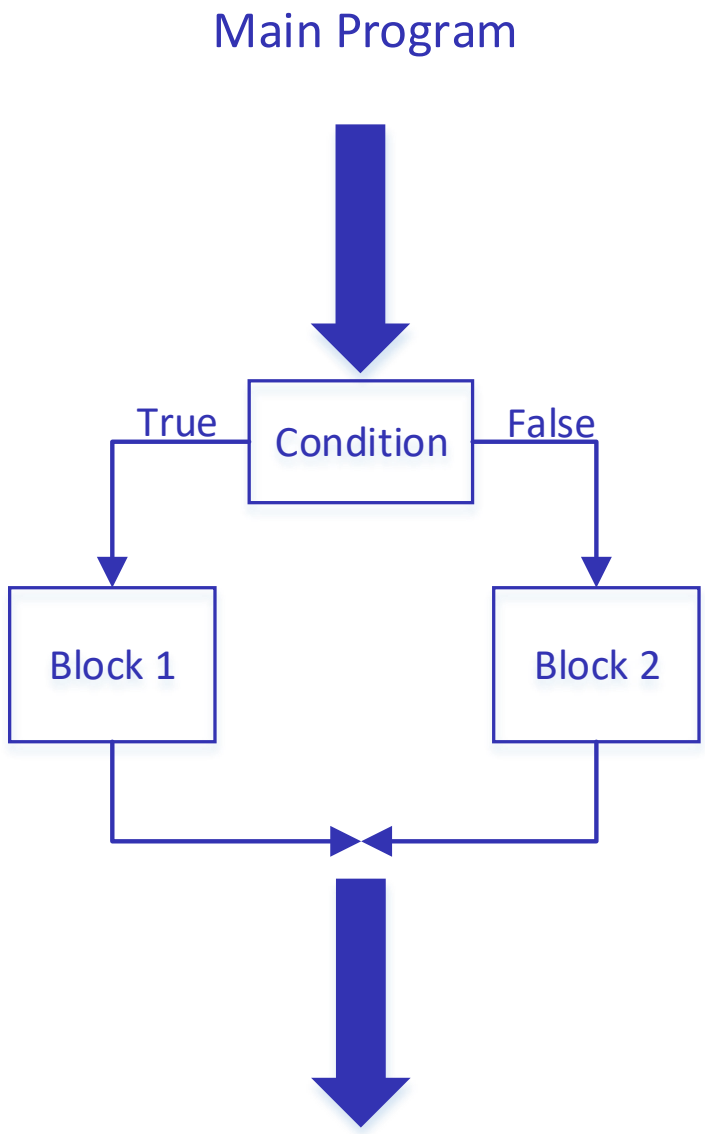
Was ist die Ausgabe der folgenden Anweisungen?

```
func(5)
y
func(func(5))
```

Gliederung

Bedingte Anweisungen

- Auswertung eines Ausdrucks in Abhängigkeit von einer Bedingung



if-else-Bedingungen

- Syntax einer `if`-Bedingung: wenn `condition` wahr, wird `statement1` ausgewertet

```
if (condition) {
  statement1
}
```

- Syntax einer `if-else`-Bedingung: wenn `condition` wahr wird `statement1` ausgewertet, sonst `statement2`

```
if (condtion) {
  statement1
} else {
  statement2
}
```

if-else-Bedingungen

- Beispiel:

```
grade <- 2
if (grade <= 4) {
  print("Passed")
} else {
  print("Failed")
}
```

```
## [1] "Passed"
```

- Bedingung muss Länge 1 haben und entweder als `TRUE` oder `FALSE` ausgewertet werden

Übung: bedingte Anweisungen

Definiere eine Variable `alter` und schreibe eine `if-else`-Bedingung, die abhängig vom Alter ausgibt, ob der Konsum von Bier erlaubt ist.

Verschachtelte `if-else`-Bedingungen

- Mehrere Bedinungen können ineinander verschachtelt werden
- Gleiche Ausgabe des obigen Beispiels lässt sich auch durch folgende Variante realisieren

```
if (grade == 1) {
  print("very good")
} else {
   if (grade == 2) {
   print("good")
   } else {
    print("not a good grade")
   }
}
```

Übung: Verschachtelte `if-else`-Bedingungen

Definiere eine Variable `alter` und nutze verschachtelte `if-else`-Bedingungen, um dir in Abhängigkeit vom Alter ausgeben zu lassen, ob kein Konsum von Alkohol, oder nur der Konsum von Bier und Wein, oder der Konsum aller alkoholhaltigen Getränke erlaubt ist.

Verschachtelte `if-else`-Bedingungen

- Einfachere Syntax ist häufig folgende
- Gleiche Ausgabe des obigen Beispiels lässt sich auch durch folgende Variante realisieren

```
if (grade == 1) {
  print("very good")
} else if (grade ==2) {
  print("good")
} else {
  print("not a good grade")
}
```

Übung: Verschachtelte `if-else`-Bedingungen

Definiere eine Variable `alter` und nutze verschachtelte `if-else`-Bedingungen, um dir in Abhängigkeit vom Alter ausgeben zu lassen, ob kein Konsum von Alkohol, oder nur der Konsum von Bier und Wein, oder der Konsum aller alkoholhaltigen Getränke erlaubt ist.

Vereinfache den Code der letzten Aufgabe, indem du die `else if()`-Syntax benutzt.

ifelse-Funktion

- Selbe Kontrollstuktur wie `if-else`-Bedingung mit der Funktion `ifelse(...)`
- Syntax von `ifelse`-Funktion: wenn `condition` wahr wird `statement1` ausgewertet, sonst `statement2`

```
ifelse(condition, statement1, statement2)
```

- funktioniert auch komponentenweise mit ganzen Vektoren, z.B.

```
grades <- c(1, 2, 3, 4, 5)
ifelse(grades <= 4, "Passed", "Failed")
```

```
## [1] "Passed" "Passed" "Passed" "Passed" "Failed"
```

Übung: bedingte Anweisungen

Was ist der jeweilige Ausgabewert?

```
if (c(TRUE, FALSE)) { print("something") }
```

```
grade <- 3
if (grade == 1) {
    print("very good")
} else {
    ifelse((grade != 2) & (grade != 3), "not a good grade",
        "good")
}
```

Übung: bedingte Anweisungen

Was ist die Ausgabe des folgenden Ausdrucks?

```
ifelse(0, 0, 1)
```

Wie kann man mit Hilfe der `ifelse()`-Funktion die `NaN` im folgenden Vektor durch `0` ersetzen?

```
x
```

```
## [1]   1   2 NaN   4   5 NaN
```

Tipp: die Funktion `is.nan()` gibt aus, ob ein Argument gleich `NaN` ist.

Übung: Funktionen und bedingte Anweisungen

Definiere eine Funktion `adress(name, gender)` mit den Argumenten `name` und `gender`, die eine korrekte Anrede ausgibt
Zum Beispiel:

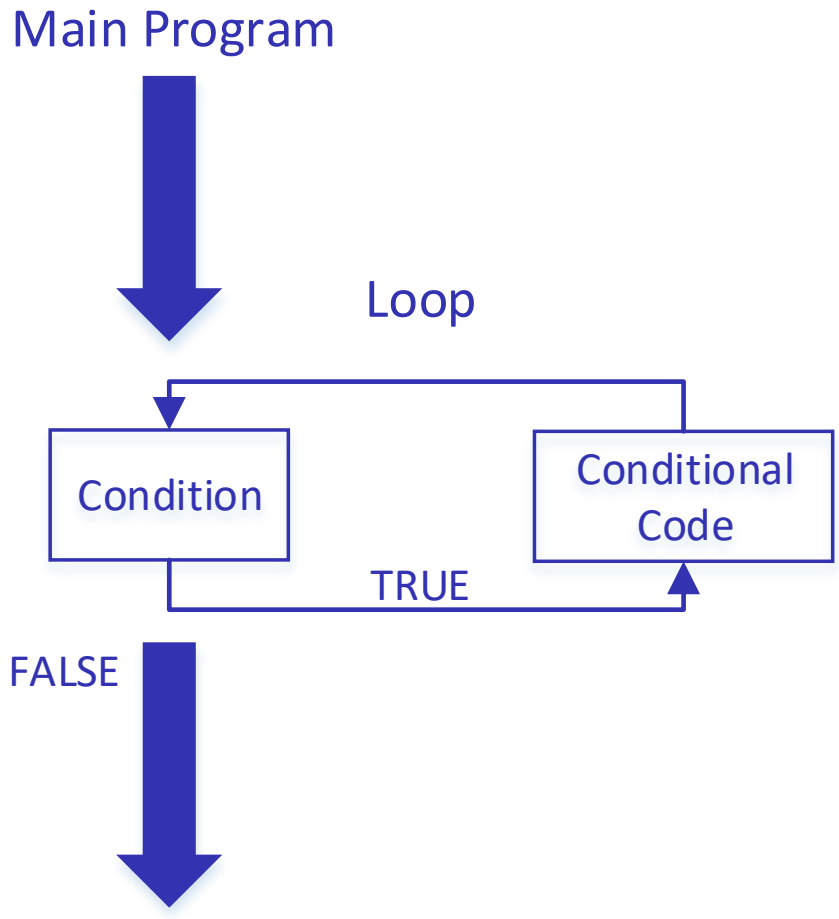
- `adress(Johanna, f) = Ms. Johanna`
- `adress(John, m) = Mr. John`

Hinweis: `paste(a, b, sep="")` fügt zwei Strings zusammen

Gliederung

Schleifen

- Folge von Anweisungen, die mehr als einmal ausgeführt werden kann



for-Schleife

- for-Schleifen werten eine Reihe von Anweisungen mit einer festen Anzahl an Wiederholungen aus
- Syntax

```
for (counter in looping_vector) {
  code to be executed for each element in the sequence
}
```

- counter nimmt in jeder Iteration der Schleife einen neuen Wert an
- Beispiel

```
for (i in 4:7) {
  print(i)
}
```

```
## [1] 4
## [1] 5
## [1] 6
## [1] 7
```

Übung: Schleifen

Schreibe eine `for`-Schleife, die die ersten 10 Quadratzahlen ausgibt.

Übung: Schleifen

Geben sei die folgende Matrix `A`. Nutze `for`-Schleifen, um jeden Eintrag der Matrix zu quadrieren.

```
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Übung: Schleifen

Definiere eine `for`-Schleife, die alle ungeraden Zahlen zwischen 1 und 10 ausgibt.
Hinweis: der Modulo-Operator %% prüft, ob eine Zahl durch 2 teilbar ist:

- `8 %% 2` gibt Rest 0 aus
- `9 %% 2` gibt Rest 1 aus

Übung: Schleifen

Erstelle einen Vektor, der die Zahlen 7 bis 50 enthält. Verdoppele dann mit einer `for`-Schleife jeden Eintrag, der durch 7 teilbar ist.

while-Schleife

- Bei `while`-Schleifen wird die Anzahl der Iterationen durch eine Bedingung bestimmt
- Bedingung wird vor jeder Iteration überprüft
- Syntax

```
while (condition) {
 code to be executed while condition is true
}
```

- Beispiel:

```
z <- 1
while (z <= 4) {
  print(z)
  z <- z + 1
}
```

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

Übung: Schleifen

Schreibe jeweils eine `while`-Schleife, die

1. die ersten 10 Quadratzahlen ausgibt,
2. alle Quadratzahlen unter 1000 ausgibt,
3. alle ungeraden Zahlen zwischen 1 und 10 ausgibt.

Übung: Funktionen und Schleifen

Die Fibonacci-Folge $(f_n)_{n\in\mathbb{N}}$ ist eine mathematische Folge, die bei $f_0 = 0$ und $f_1 = 1$ startet und dann immer die letzen beiden Folgenglieder addiert, also

$$f_n = f_{n-1} + f_{n-2} \text{ für } n \geq 2.$$

Folgende Funktion gibt die ersten `n` Elemente der Fibonacci-Folge aus.

```
fibonacci <- function(n) {
  x <- c(1, 1)
  for (k in 1:(n - 2)) {
    x[k + 2] <- x[k] + x[k + 1]
  }
  return(x)
}
fibonacci(10)
```

```
##  [1]  1  1  2  3  5  8 13 21 34 55
```

Wie kann man die Funktion mit einer `while`-Schleife formulieren?

Übung: Bedingte Anweisungen und Schleifen

- Gib mit Hilfe einer `for`-Schleife alle Primzahlen zwischen 1 und 20 aus.
- Wie sieht die `while`-Schleife aus?
- Warum ist dieser Algorithmus nicht effizient?

Hinweis: Eine Primzahl ist eine natürliche Zahl $n \geq 2$, die nur durch 1 und sich selber teilbar ist.

Übung: Funktionen, bedingte Anweisungen und Schleifen

Lege die angegebene Matrix `mat` an. Schreibe eine Fuktion `find(mat,element)` mit den Argumenten `mat` und `element`, die den Index des ersten Eintrags, der mit `element` übereinstimmt, zurück gibt. Die Matrix soll dabei zeilenweise durchgegangen werden.

$$\mathrm{mat} = \begin{bmatrix} 1 & 7 & 3 \\ 3 & 5 & 1 \\ 0 & 12 & 4 \end{bmatrix}$$

Gliederung

Zusammenfassung

- Funktionen (Variablen werden nur lokal definiert!)

```
square <- function(x) {
  res <- x*x
  return(res)
}
```

- if-Bedingungen

```
x <- 3
if (x == 1) {
  print("Sehr gut")
} else if (x < 3) {
  print("Gut")
} else {
  print("OK")
}
```

- ifelse()-Funktion

```
ifelse (x < 3, "Gut", "OK")
```

Zusammenfassung

- `for`- Schleifen

```
for (i in 1:5) {
  print(i)
}
```

- `while`- Schleifen

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