R-Kurs: Tag 4

Lukas Mödl, Matthias Becher, Erin Sprünken

Institut für Biometrie und Klinische Epidemiologie

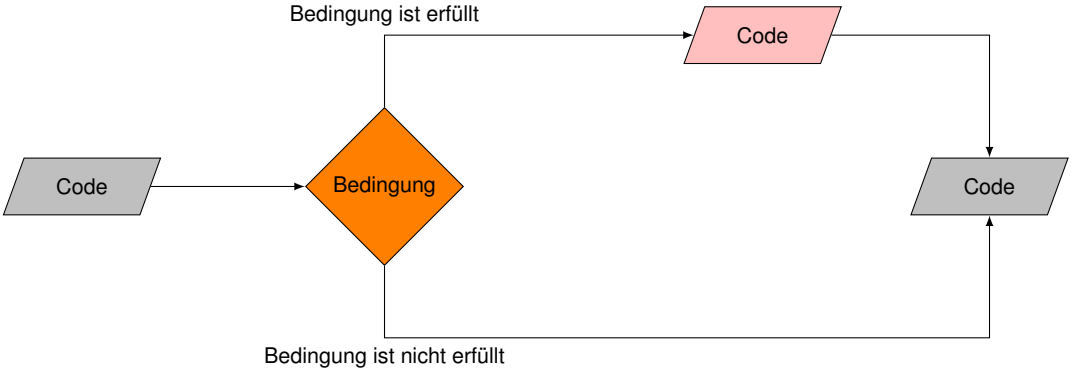
Charité - Universitätsmedizin Berlin, Berlin

erin-dirk.spruenken@charite.de

March 6, 2022

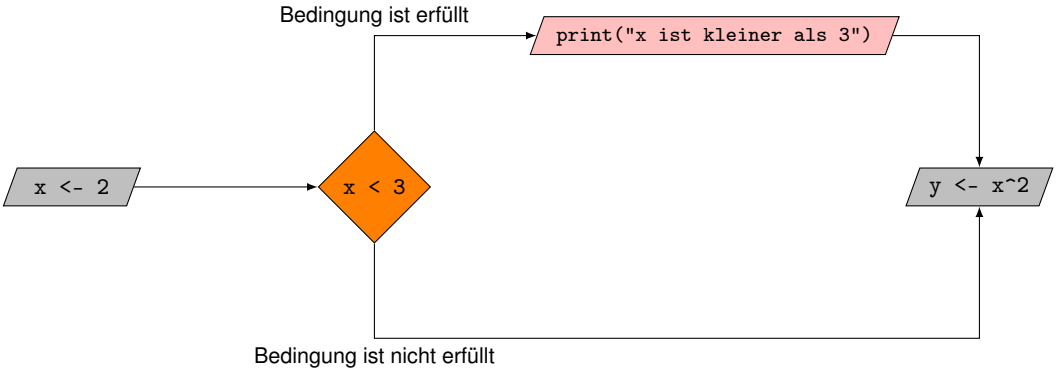
Konzeption if-Anweisung

Möchte man einen Codeblock abhängig von einem bestimmten Wert ausführen oder nicht, muss eine if-Anweisung verwendet werden. if-Anweisungen definieren einen Codeblock, der nur dann ausgeführt wird, wenn die übergebene Bedingung wahr ist.



Beispiel if-Anweisung

Um die Konzeption der if-Anweisung zu verstehen, soll in diesem Beispiel einer Variablen `x` der Wert 3 zugewiesen werden. Abhängig vom Wert von `x` soll dann eine Ausgabe ausgeführt werden oder nicht.



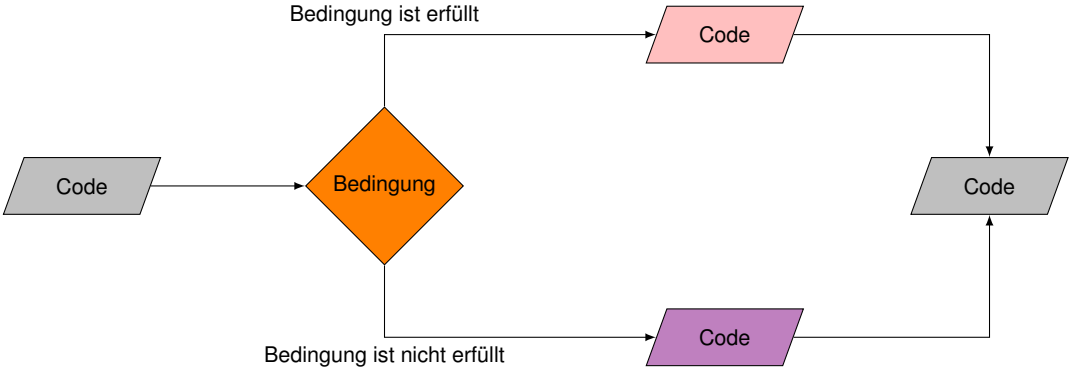
Beispiel if-Anweisung

Um die Konzeption der if-Anweisung zu verstehen, soll in diesem Beispiel einer Variablen `x` der Wert 3 zugewiesen werden. Abhängig vom Wert von `x` soll dann eine Ausgabe ausgeführt werden oder nicht.

```
> x <- 2
> if(x < 3)
+ {
+     print("x ist kleiner als 3")
+ }
[1] "x ist kleiner als 3"
> y <- x^2
```

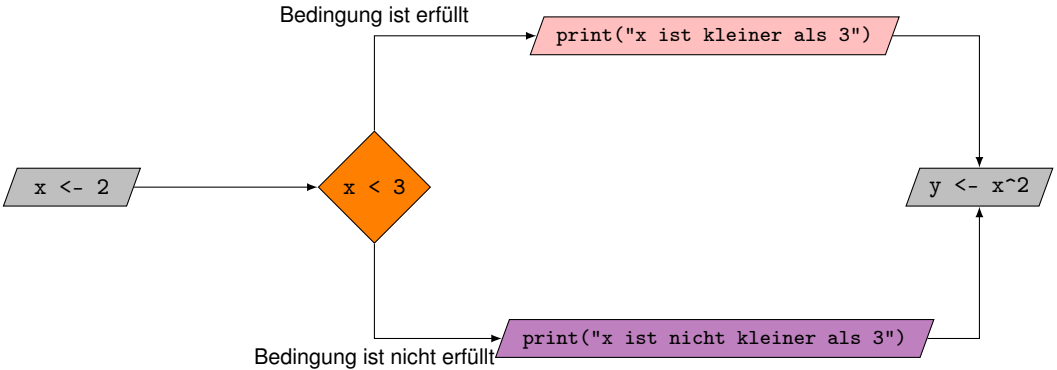
Konzeption if-else-Anweisung

Bei der if-Anweisung wird der Code, der nach der if-Anweisung steht immer ausgeführt. Das ist in einer Entweder-Oder-Situation natürlich nicht wünschenswert. Hier schafft die if-else-Anweisung Abhilfe.



Beispiel if-else-Anweisung

Um die Konzeption der if-else-Anweisung zu verstehen, wird die bedingte Ausgabe des vorherigen Beispiels leicht modifiziert.



Beispiel if-else-Anweisung

Um die Konzeption der if-else-Anweisung zu verstehen, wird die bedingte Ausgabe des vorherigen Beispiels leicht modifiziert.

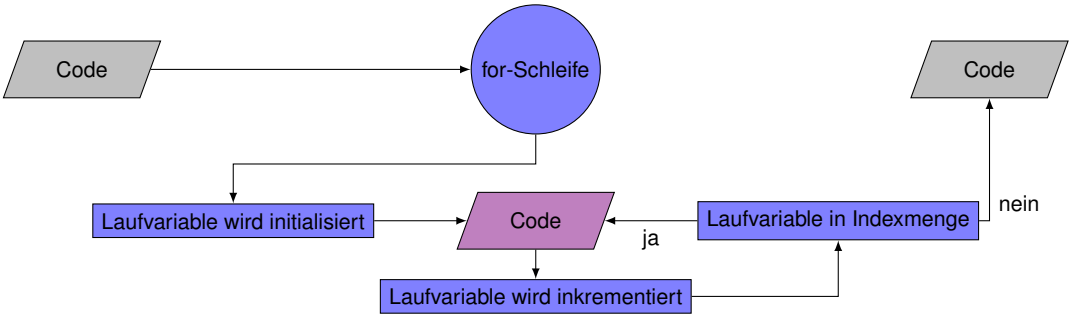
```
> x <- 3
> if(x < 3)
+ {
+     print("x ist kleiner als 3")
+ } else
+ {
+     print("x ist nicht kleiner als 3")
+ }
[1] "x ist nicht kleiner als 3"
> y <- x^2
```

Anmerkungen zur if-else-Anweisung

- Das Setzen der geschweiften Klammern { und } ist essenziell!
- Als Bedingungen sind nur logische Ausdrücke zugelassen
- Manchmal sind verschachtelte if-else-Anweisungen nötig
- Für bessere Lesbarkeit des Codes empfiehlt es sich, Codeblöcke mit Tab einzurücken

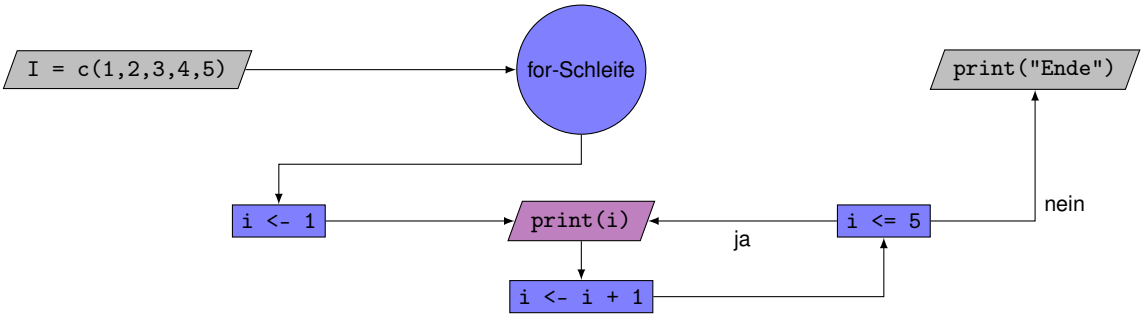
Konzeption for-Schleife

Eine beiden wesentlichen Schleifen in R ist die for-Schleife. for-Schleifen führen einen Codeblock so oft aus, wie eine Laufvariable Werte in einer Indexmenge (einem Vektor oder einer Liste) annehmen kann. Die for-Schleife wird mit dem Schlüsselwort `for()` gefolgt von einem Codeblock in geschweiften Klammern eingeleitet. In den runden Klammern wird die Laufvariable über die Indexmenge initialisiert.



Beispiel for-Schleife

Die Funktionsweise der for-Schleife soll an diesem Beispiel verdeutlicht werden. Hier entspricht `I` der Indexmenge und `i` Bezeichnet die Laufvariable.



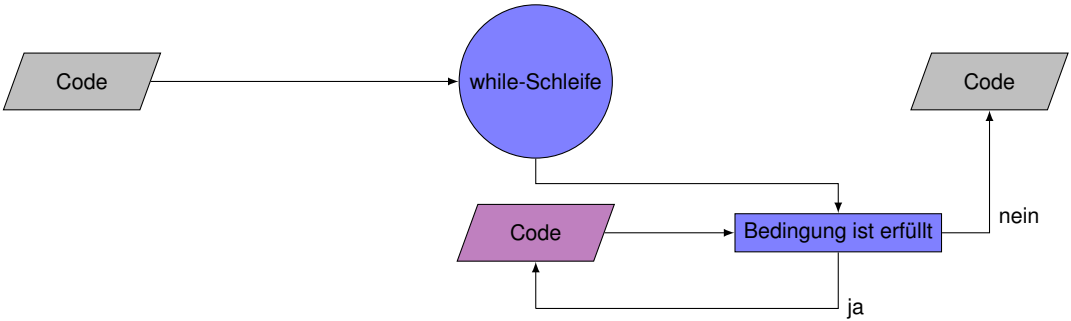
Beispiel for-Schleife

Die Funktionsweise der for-Schleife soll an diesem Beispiel verdeutlicht werden. Hier entspricht `I` der Indexmenge und `i` Bezeichnet die Laufvariable.

```
> I <- c(1, 2, 3, 4, 5)
> for(i in I)
+ {
+     print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
> print("Ende")
[1] "Ende"
```

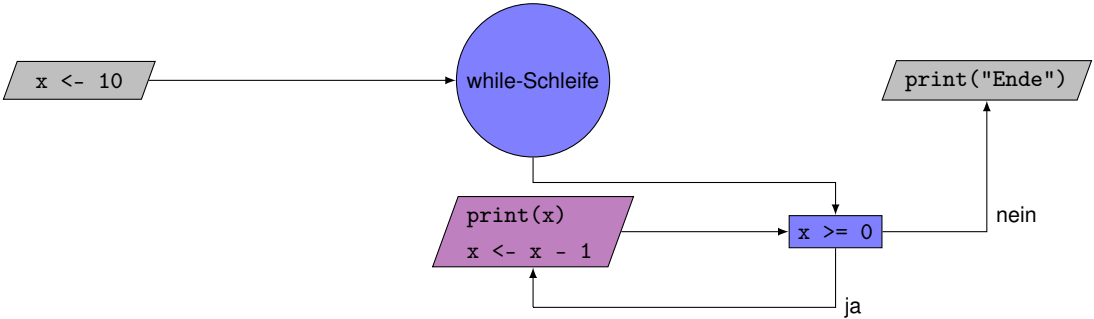
Konzeption while-Schleife

Die andere wesentliche Schleife ist die while-Schleife. Im Gegensatz zur for-Schleife wird hier ein Codeblock so lange ausgeführt, wie eine gegebene Bedingung erfüllt ist. Die while-Schleife wird mit dem Schlüsselwort `while()` gefolgt von einem Codeblock in geschweiften Klammern eingeleitet. Innerhalb der runden Klammern wird die Bedingung übergeben.



Beispiel while-Schleife

Das folgende Beispiel erklärt die Funktionsweise der while-Schleife. Die Bedingung hier ist `x >= 0`.



Beispiel for-Schleife

Das folgende Beispiel erklärt die Funktionsweise der while-Schleife. Die Bedingung hier ist `x >= 0`.

```
> x <- 10
> while(x >= 0)
+ {
+     print(x)
+     x <- x - 1
+ }
[1] 10
[1] 9
[1] 8
[1] 7
[1] 6
[1] 5
[1] 4
[1] 3
```

Anmerkungen zu for- und while-Schleifen

- Das Setzen das geschweiften Klammern `{` und `}` ist essenziell!
- Vorsicht vor unendlichen Schleifen!
- Jede for-Schleife kann auch als while-Schleife programmiert werde und umgekehrt
- Mit `break` kann eine Schleife beendet werden
- Mit `next` kann eine Iteration übersprungen werden, ohne dass die Schleife beendet wird
- Die `apply`-Familie bietet oft eine praktische Alternative zu Schleifen

Apply-Familie

Die Apply-Familie ist eine Reihe von Funktion in R, die es uns erlaubt eine Funktion auf mehrere verschiedenen Inputs nacheinander anzuwenden. Zum Beispiel auf alle Zeilen oder Spalten einer Matrix oder alle Elemente einer Liste. Die verschiedenen Apply-Funktionen sind:

- `apply()`
- `lapply()`
- `sapply()`
- `tapply()`

apply()

Mit apply() können wir Funktionen auf alle Zeilen oder Spalten eines Data Frames oder einer Matrix anwenden um zum Beispiel alle Spaltensummen zu berechnen. Die Grundform der Funktion ist:

- `apply(data, margin, function)`

Zum Beispiel:

```
> apply(data, 1, sum)
[1] 10 26 42 58
> apply(data, 2, sum)
[1] 28 32 36 40
```

1	2	3	4	10
5	6	7	8	26
9	10	11	12	42
13	14	15	16	58
28	32	36	40	

lapply()

lapply() führt eine Funktion auf jedes Element eines Data Frames, einer Matrix, eines Vektors oder einer Liste aus. Das "l" in lapply() steht dabei für "list" und bezieht sich darauf, dass lapply() immer eine Liste zurück gibt.

- `lapply(object, function)`

Zum Beispiel:

```
> lapply(c("A","B","C"), tolower)
[[1]]
[1] "a"

[[2]]
[1] "b"

[[3]]
[1] "c"
```

sapply()

sapply() macht im Grunde das gleiche wie lapply(). Der Unterschied ist, dass sapply() einen Vektor oder eine Matrix zurück gibt, anstatt eine Liste:

- `sapply(object, function)`

Zum Beispiel:

```
> sapply(c(-1, 2, -3), abs)
[1] 1 2 3
> l <- list(a = 1:10, b = 1:20)
> sapply(l, mean)
   a    b
 5.5 10.5
```

tapply()

tapply() erlaubt es uns auf Grundlage von factor levels Gruppenzusammenfassungen zu erstellen:

- `tapply(object, index, function)`

Zum Beispiel:

```
> data <- data.frame(Sex = as.factor(c(rep("M",100), rep("W",100))))
> data$Height <- c(rnorm(100,180,5),rnorm(100,166,4))
> tapply(data$Height, data$Sex, mean)
       M        W
180.3354 165.3481
```

Konzeption eigener Funktionen

Bisher haben wir nur vordefinierte Funktionen wie beispielsweise `t.test` verwendet. R erlaubt es aber auch, eigene Funktionen zu schreiben.

Funktionen sind ein eigener Datentyp, der mit dem Schlüsselwort `function()` intialisiert werden muss, gefolgt von einem Codeblock in geschweiften Klammern. Innerhalb der runden Klammern können freie Variablen definiert werden, die von der zu definierenden Funktion als Argumente interpretiert werden.

Beispiel für eine Funktion mit einem Argument

Bei der Funktion $f(x) = x^2$ entspricht x einer freien Variablen. Die Funktion wird wie folgt in R implementiert:

```
> f <- function(x)
+ {
+     return(x^2)
+ }
>
```

Anmerkungen zur Definition eigener Funktionen

- Das Setzen das geschweiften Klammern `{` und `}` ist essenziell!
- Werden Argumente spezifiziert, müssen die bei jedem Aufruf der Funktion übergeben werden Eine Ausnahme bilden freie Variablen, die innerhalb der runden Klammern des Schlüsselworts `function()` mittels = initialisiert wurden
- Sollen mehrere freie Variablen übergeben werden, müssen diese mit Komma getrennt werden
- Sofern möglich R vektorisiert Funktionen automatisch, d.h. wird ein Vektor oder eine Matrix als Argument übergeben, wird die Funktion auf jedes Element einzeln angewendet
- Soll eine Funktion keinen Wert zurückgeben, kann auf `return` verzichtet werden