

Eine Einführung in R: Programmstrukturen

Bernd Klaus, Verena Zuber

Institut für Medizinische Informatik, Statistik und Epidemiologie (IMISE),
Universität Leipzig

<http://www.uni-leipzig.de/zuber/teaching/ws12/r-kurs/>

1. November 2012

① Umgang mit fremden Funktionen

② Erstellen eigener Funktionen

- Definition eigener Funktionen

- Die `if`-Abfrage

- Schleifen und Vektorisierung

- Rechenzeit

- Scope

I. Umgang mit fremden Funktionen

Der Aufbau von Funktionen

```
funktions.name(argumente, optionen)
```

- `argumente`: Einige Argumente sind zwingend nötig, um die Funktion zu starten
- `optionen`: Einige Argumente können optional verändert werden, sonst wird die Standardeinstellung verwendet
- `?funktions.name`: Aufrufe der Hilfe
- `funktions.name`: Aufruf der Syntax

Beispiel: der `plot` und `mean`-Befehl

```
plot(x, y, ...)
```

- `x`: Daten auf der x -Achse
- `y`: Daten auf der y -Achse
- `...`: Weitere optionale Argumente, wie Farbe, Linientyp, usw.

```
mean(x, trim = 0, na.rm = FALSE)
```

- `x`: Daten
- `trim=0`: Berechnung des getrimmten Mittel
(Mittel auf Basis der Daten ohne $x\%$ der extremsten Werte)
- `na.rm = FALSE`: Sollen fehlende Werte entfernt werden?

II. Erstellen eigener Funktionen

Erste Überlegungen

- ① Welchen Namen soll die Funktion tragen?
- ② Was soll die Funktion machen? Welcher Output wird erwartet?
- ③ Welche Argumente müssen dazu übergeben werden?
- ④ Welche Argumente sollen variabel gehalten werden?

Funktionen und Objekte können beliebig benannt werden.
Doch Vorsicht: Bestehende Funktionen können überschrieben werden!

```
funktions.name<-function(argumente, optionen) {  
  ...  
  ...  
}
```

- `funktions.name`: Der gewünschte Namen der Funktion
- `argumente`: Einige Argumente sind zwingend nötig, um die Funktion zu starten
- `optionen`: Optionale Argumente mit der Standardeinstellung
- `{ }` : Die Syntax der Funktion muss in geschweiften Klammern stehen
- R gibt als Ergebnis die letzte Anweisung aus, ansonsten sollte `return()` verwendet werden
- Es werden keine Komma gesetzt, um einzelne Argumente abzutrennen

Beispiel: Der Mittelwert

```
mittelwert<-function(data) {  
  my.sum<-sum(data, na.rm=TRUE)  
  my.length<-length(which(data!="NA"))  
  my.sum/my.length  
}
```

- `mittelwert`: Funktionsname
- `data`: Daten (auch mit fehlenden Werten)
- `my.length`, `my.sum`: Lokale Variablen (d.h. nur innerhalb der Funktion gültig)
- Output: Das letzte Argument, also der berechnete Mittelwert!

Beispiel: Währungsrechner

```
euro.to.us<-function(x) {  
  x*1.355  
  # Wechselkurs vom 22.11.2011  
}
```

- `x`: Betrag in Euro
- Output: Betrag in US-Dollar
`euro.to.us(c(1,2,3))` liefert 1.355 2.710 4.065

Kommentare (`# Kommentar`) helfen die (eigenen) Funktionen zu verstehen!

Die if-Abfrage

```
if(bedingung) { anweisung }
```

- **bedingung**: Logische Abfrage; wenn TRUE, dann
- **anweisung**: Führe diese Anweisung aus
- **{ }** : Sind nur dann notwendig, wenn die Anweisung über mehrere Zeilen geht

Für komplexere Abfragen:

```
if(bedingung) { anweisung }  
else { anweisung }
```

Beispiel: Währungsrechner II

```
euro.calc<-function(x, currency="US") {  
  if(currency=="US") x*1.355  
  if(currency=="Pfund") x*0.865  
  # Wechselkurs vom 22.11.2011  
}
```

- **x**: Betrag in Euro
- **currency=="US"**: Umwandlung in US-Dollar (Standard)
- **currency=="Pfund"**: Umwandlung in Britische Pfund
- Output: Betrag in US-Dollar oder Britische Pfund
`euro.calc(1, currency="Pfund")` liefert `0.865`

```
for(index von : bis) { anweisung [index] }
```

- **index**: Definieren eines Index
- **von**: Anfangswert
- **bis**: Endwert
- **anweisung [index]**: Anweisung in Abhängigkeit vom Index

```
while(bedingung) { anweisung }
```

- **bedingung**: Solange diese Bedingung erfüllt ist
- **anweisung**: Erfülle diese Anweisung

Beispiel: for-Schleife

```
m.mean<-function(X) {  
  n<-ncol(X)  
  res<-rep("NA",n)  
  for(i in 1:n){  
    res[i]<-mean(X[,i], na.rm=TRUE)  
  }  
  return(res)  
}
```

Dies ist äquivalent zu:

```
apply(X, MARGIN=2, FUN=mean, na.rm=TRUE)
```

Ermittlung der Rechenzeit

```
system.time(expr)
```

- `expr`: R-Befehl, dessen Rechenzeit ausgewertet werden soll

Beispiel: `m.mean` gegen `apply`

```
try<-matrix(1:40000, nrow=4)  
system.time(m.mean(try))
```

```
user system elapsed  
0.408 0.000 0.407
```

```
system.time(apply(try, MARGIN=2, FUN=mean,  
na.rm=TRUE))
```

```
user system elapsed  
0.324 0.000 0.324
```

Warum ist `apply` besser als die `for`-Schleife?

Stichwort: Vektorisierung

In R wird bei jedem Schleifendurchlauf jede Zeile neu interpretiert. Dies kann dazu führen, daß Schleifen im Vergleich zu vektorartigen Operationen (hier wird nur einmal interpretiert) wesentlich länger laufen. Deswegen gilt:

Wenn möglich vektorwertige (-> `apply`-Funktionen)
Alternativen verwenden.

Scope [Gültigkeitsbereich] von Variablen bei Funktionen

Es können drei Arten von Variablen in einer Funktion auftauchen:

- **Formale Parameter:**
Werden beim Aufruf der Funktion angegeben
- **Lokale Variablen:**
Werden beim Abarbeiten einer Funktion erzeugt
- **Freie Variablen:**
Alle anderen

Frage: Wo sucht R nach freien Variablen?

Antwort: In der Umgebung der Variable

```
z <- 3
f <- function(x) {
  y <- 2*x
  print(x)
  print(y)
  print(z)
}
```

Ausgabe bei Aufruf der Funktion:

f(1)
3

f(60)
3

- **x**: Formaler Parameter
- **y**: Lokale Variable
- **z**: Freie Variable, die in diesem Bsp. von R außerhalb der Funktion gesucht wird

```
z <- 3
f <- function(x) {
  y <- 2*x
  print(x)
  z <- 5
  print(z)
}
```

Ausgabe bei Aufruf der Funktion:

f(1)
5

f(60)
5

- z ist keine freie Variable mehr, da sie nun innerhalb der Funktion definiert ist (lokale Variable) und die freie Variable z außerhalb der Funktion verdeckt
- Zugriff auf verdeckte Variablen per \llcorner - Befehl