LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Programmieren mit statistischer Software

Eva Endres, M.Sc.

Institut für Statistik
Ludwig-Maximilians-Universität München

Vektorwertiges Programmieren

Programmieren ohne explizite Schleife I

- Vektorisierung

```
> set.seed(1234)
> x <- rnorm(5)
> y <- runif(5)
```

- Vektorisierte Berechnung (implizite Schleife)

```
> x + y
[1] -0.5134745  0.8224041  1.3671748 -1.4222642  0.7214405
```

- statt expliziter Schleife

```
> z <- numeric(length(x))
> for ( i in seq(along.with = x) ) {
+   z[i] <- x[i] + y[i]
+ }
> z
[1] -0.5134745  0.8224041  1.3671748 -1.4222642  0.7214405
```

- Vektorisierte Berechnungen sind
  - kompakter
  - besser lesbar

Programmieren ohne explizite Schleife II

- meist schneller (bei größen Eingaben)

```
> set.seed(1234)
> x <- rnorm(10^6)
> y <- runif(10^6)
```

- Implicit loop: interne Schleife, in Maschinencode programmiert (daher schneller)

```
> system.time(x + y)
```

```
   user  system elapsed
      0       0       0
```

- Explicit loop: explizite Schleife, in R programmiert

```
> z <- numeric(length(x))
> system.time(for ( i in seq(along.with = x) ) z[i] <- x[i] + y[i])
```

```
   user  system elapsed
   0.13    0.02    0.14
```

$\rightarrow$ wannimmer möglich, sollte in R vektorisiert programmiert werden

Programmieren ohne explizite Schleife III

- Vorsicht: in R werden „Recycling rules“ verwendet!
  Wenn mit zwei Vektoren unterschiedlicher Länge gearbeitet wird, und die Länge des einen Vektors ein Vielfaches des anderen ist, dann wird der kürzere Vektor wiederholt.

```
> c(1, 2, 3) + 1:6
[1] 2 4 6 5 7 9
```

- Nur wenn die Länge des einen Vektors kein Vielfaches des anderen ist, wird eine Warnmeldung ausgegeben.

```
> c(1, 2, 3) + 1:5
> # Warning message:
> # In c(1, 2, 3) + 1:5 :
> #   longer object length is not a multiple of shorter object length
```

Implizite Schleifen I

- Operatoren

```
> ?Arithmetic
> ?Comparison
```

```
> set.seed(1234)
> x <- sample(100, 5)
> ((x %% 2) == 0)
[1]  TRUE  TRUE  TRUE FALSE FALSE
```

- Statistische Maßzahlen

```
> mean(x)     # Summe als implizite Schleife, geteilt durch n
[1] 55.6
> sum(x) / length(x)
[1] 55.6
```

Implizite Schleifen II

- dasselbe mit expliziter Schleife

```
> z <- 0
> for ( i in seq(along.with = x) ) {
+   z <- z + x[i]
+ }
> z <- z / length(x)
> z
```

```
[1] 55.6
```

- auch möglich für Datensätze - Zeilen- und Spalten-Summen (und Mittelwerte)

```
> data("cars", package = "datasets")
> # ?cars
> str(cars)
```

```
'data.frame': 50 obs. of  2 variables:
 $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

```
> colMeans(cars)  # implizite Schleife
```

```
speed  dist
15.40 42.98
```

Implizite Schleifen III

```
> ?rowMeans
> ?rowSums
> ?colSums
```

- Subsets - Extraktion bzw. Ersetzungen von Teilmengen

```
> set.seed(1234)
> x <- letters[1:10]
> ind <- sample(100, 10)
```

- Teilmenge der Buchstaben, für die ind gerade ist

```
> x[ind %% 2 == 0]
[1] "a" "b" "c" "h"
> # bzw. mit which()
> x[which(ind %% 2 == 0)]
[1] "a" "b" "c" "h"
```

Implizite Schleifen IV

- als explizite Schleife

```
> z <- character()
> for ( i in seq(along.with = x) ) {
+   if ( ind[i] %% 2 == 0 ) {
+     z <- c(z, x[i])
+   }
+ }
> z
[1] "a" "b" "c" "h"
```

- oder mit `ifelse` (aber längerer Vektor mit zusätzlichen NA-Einträgen)

```
> ifelse(test = ind %% 2 == 0, yes = x, no = NA)
 [1] "a" "b" "c" NA  NA  NA  NA  "h" NA  NA
```

- Matrizenrechnung
  - für Matrizen (zweidimensional) - `matrix`
  - und Arrays (drei- und mehrdimensional) - `array`

Implizite Schleifen V

z.B. Matrixmultiplikation, Transponierung, Bilden von Teilmengen

- Transponieren mit impliziter Schleife

```
> set.seed(1234)
> A <- matrix(sample(10), ncol = 2)
> A
     [,1] [,2]
[1,]    2    4
[2,]    6    1
[3,]    5    7
[4,]    8   10
[5,]    9    3
> t(A)
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    6    5    8    9
[2,]    4    1    7   10    3
```

Implizite Schleifen VI

- Transponieren mit expliziter Schleife

```
> n <- nrow(A)
> m <- ncol(A)
> At <- matrix(NA, nrow = m, ncol = n)
> for ( i in seq(length.out = n) ) {
+   for ( j in seq(length.out = m) ) {
+     At[j, i] <- A[i, j]
+   }
+ }
> At
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    6    5    8    9
[2,]    4    1    7   10    3
```

Die `apply`-Familie I

- hiermit kann man für alle weiteren Funktionen implizite Schleifen programmieren
- wiederholte Anwendung einer Funktion für
- alle Elemente eines Vektors
- alle Zeilen / Spalten einer Matrix
- alle Dimensionen eines Arrays
- Vorteile von apply gegenüber expliziter Schleife
  - kompaktere Berechnung
  - Parallelisierbar (mit Paket `plyr`)
  - oft schneller

Die apply-Familie II

Beispiele:

- Funktion angewendet auf eine Liste bzw. einen Vektor

```
> ab <- function(x) {
+   if ( x < 10 ) {
+     "a"
+   } else {
+     "b"
+   }
+ }
```

- anwendbar für Skalare

```
> ab(4)
[1] "a"
> ab(20)
[1] "b"
```

Die apply-Familie III

- nicht korrekt anwendbar für Vektoren

```
> ab(5:15)
> # Warning message:
> # In if (x < 10) { :
> #   the condition has length > 1 and only the first element will be used
```

- anwendbar für Vektoren mit

```
> # ?lapply
> # gibt immer eine Liste zurueck
> lapply(X=5:15, FUN=ab)

> # ?sapply
> # versucht, das Ergebnis zu vereinfachen
> # (zu einem Vektor, einer Matrix, einem Array)
> sapply(X=5:15, FUN=ab)
```

Die apply-Familie IV

- Liste mit Ergebnissen von verschiedenen Durchgängen eines Experiments

```
> set.seed(1234)
> l <- list(exp1 = rnorm(100),
+           exp2 = rnorm(100),
+           exp3 = rnorm(100))
```

- Mittelwerte (von Hand berechnet)

```
> c(mean(l$exp1),
+   mean(l$exp2),
+   mean(l$exp3))
```

```
[1] -0.15676174  0.04124318  0.15460367
```

- mit expliziter Schleife

```
> ms <- numeric(length = length(l))
> for ( i in seq(along.with = l) ) {
+   ms[i] <- mean(l[[i]])
+ }
> ms
```

```
[1] -0.15676174  0.04124318  0.15460367
```

Die apply-Familie V

- einfacher mit

```
> sapply(l, mean)
       exp1        exp2        exp3
-0.15676174  0.04124318  0.15460367
```

- Unterschied zwischen lapply und sapply

```
> lapply(l, quantile)  # gibt Liste zurueck
> sapply(l, quantile)  # gibt Matrix zurueck
```

- multivariate Version von sapply

```
> # ?mapply
> mapply(rep, 1:4, times=4:1)
> mapply(FUN=quantile, l, probs=list(c(0,0.5,0.75,1),
+          c(0.2,0.4), c(0.1,0.3,0.6)))
```

- wiederholte Auswertung von Ausdrücken mit

```
> # ?replicate
> replicate(n=5, expr=rnorm(10,0,1))
```

Die apply-Familie VI

- Funktion angewendet auf eine Matrix, ein Array oder einen Datensatz

```
> data("cars", package = "datasets")
>
> # ?apply
> # Maximum pro Zeile bzw. Spalte
> apply(X=cars, MARGIN=1, FUN=which.max)
> apply(X=cars, MARGIN=2, FUN=which.max)
> # MARGIN=1 fuer Zeilen
> # MARGIN=2 fuer Spalten
> # MARGIN=c(1,2) fuer Zellen
```

Anonyme Funktionen I

- Funktion ohne Name

```
> function(x) {
+   x + 1
+ }
```

```
function(x) {
  x + 1
}
```

- Anwendung innerhalb von `apply`-Befehlen

```
> sapply(5:15,
+        function(x) {
+          if ( x < 10 ) {
+            "a"
+          } else {
+            "b"
+          }
+        })
```

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

Apply-family gems I

- Vereinfachung des Ergebnisses bzw. Zugriff auf Teile des Ergebnisses

```
> ds <- lapply(1:10, function(i) data.frame(
+   a = runif(4), b = gl(2, 2), c=runif(4)))
> # ds <- replicate(10, data.frame(
> # a = runif(4), b = gl(2, 2), c=runif(4)),
> # simplify=FALSE) # alternativ
> # str(ds, 1)
```

- Vereinfachung des Ergebnisses - in ein großes Objekt

```
> # do.call(, lapply())
```

- Beispiel: Liste von Datensätzen in einen großen Datensatz umwandeln

Apply-family gems II

```
> # ?do.call
> d <- do.call(what=rbind, args=ds)
> str(d)
'data.frame': 40 obs. of  3 variables:
 $ a: num  0.281 0.174 0.17 0.561 0.509 ...
 $ b: Factor w/ 2 levels "1","2": 1 1 2 2 1 1 2 2 1 1 ...
 $ c: num  0.429 0.389 0.844 0.209 0.307 ...
```

- Zugriff auf Teile einer Liste

```
> # lapply(, "[[", )
```

- Beispiel: Spalte b für jeden Listeneintrag auswählen

```
> bs  <- lapply(ds, "[[", "b")
> # lapply(ds, "[[", c("b","c")) # funktioniert nicht
```

Apply-family gems III

- daher mit anonymen Funktionen

```
> bs2 <- lapply(1:10, function(j) ds[[j]][,c("b","c")])
> bs3 <- lapply(1:10, function(j) ds[[j]][c("b","c")])
>
> # Liste aus Data-frames
> bs4 <- lapply(1:10, function(j) ds[[j]]["b"])
> # Liste aus Vektoren
> bs5 <- lapply(1:10, function(j) ds[[j]][,"b"])
> # Liste aus Data-frames
> bs6 <- lapply(1:10, function(j) ds[[j]][, "b", drop=FALSE])
```

- auch kompliziertere Operationen/Selektionen möglich

```
> bs7 <- lapply(1:10, function(j) ds[[j]][2,"b"])
```

Funktionale Programmierung I

- Berechnungen als Auswertung von Funktionen
- higher-order functions - tun mindestens eines der beiden folgenden:
  - nehmen eine oder mehrere Funktionen als Input
  - geben eine Funktion zurück

  → `apply`-Funktionen sind higher-order functions

  → die meisten anderen (wie `mean()`) sind first order functions
  - weitere higher-order functions in R

    ```
    > ?Map
    ```