WERKZEUGE DER MUSTERERKENNUNG UND DES MASCHINELLEN LERNENS

Vorlesung im Sommersemester 2017

Prof. E.G. Schukat-Talamazzini

Stand: 19. April 2017

Teil II

Rechnen in 

Syntax und Semantik

Übersetzte *vs.* interpretierte Programmiersprachen

Syntax

Ist der Quellcode ein zulässiger Programmtext oder nicht?

Semantik

Was wird berechnet?
Was wird bewirkt?

Syntaxfehler

{ keine Übersetzung! / kein Start! }

Laufzeitfehler

Wert oder Zustand undefiniert!

Übersetzung

Compiler transformiert Quellcode in Zielcode.

Vor- & Nachteile

⊕ Effizienz
⊖ Modifikation

Interpretation

Interpreter führt Quellcode schritthaltend aus.

Vor- & Nachteile

⊕ Interaktion
⊖ Effizienz

ist interpretativ und funktional

IM PRINZIP EINFACH ...

Funktionale Programme

bestehen aus einer Folge von **Ausdrücken**.

Ausdrücke

sind geschachtelte **Funktionaufrufe**.

Auswertung

eines Ausdrucks immer von innen nach außen.

Anweisungen

gibt es nicht.

IM DETAIL KOMPLEXER ...

- Zeilenorientierung
- Variable & Namensräume
- Explizite Typanpassung
- Operatorschreibweise
  Arithmetik & Logik
  Indexnotation (Reihung)
  Selektion (Verbund)
  Zuweisung
  Datenmodelle
- teilweise Klammerzwang
- explizite Auswerteregelung
- Verzögerung & Versprechen
- *„Kontrollstrukturen“*

Die Syntax von

Aus großer Flughöhe bei hoher Geschwindigkeit beobachtet

Programm = Folge von Ausdrücken

program	::=	*expr**
expr	::=	*objID* | *literal* | *funcall* | *fobj* | `(`*expr*`)` | *block* | *control*
block	::=	`{`[*expr exsep*]* *expr*`}`
control	::=	*if1Ctl* | *if2Ctl* | *forCtl* | *repeatCtl* | *whileCtl*
exsep	::=	`;` | `\n` | *exsep*$^+$

Funktionsaufruf (Standard- oder Operatorform)

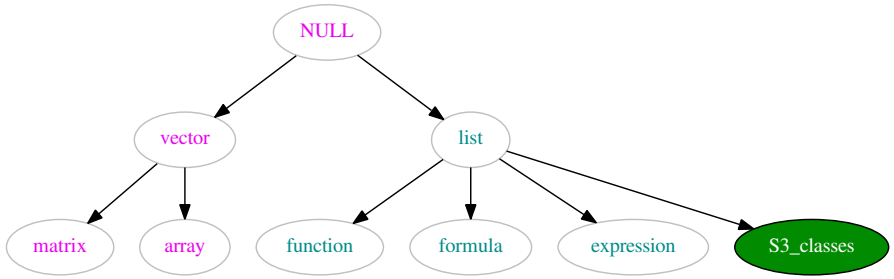
funcall	::=	*fname*`(`*arglist*`)` | *opcall*
arglist	::=	{[*arg* `,`]* *arg*}
arg	::=	*expr* | *formpar*`=`*expr* | `...`
opcall	::=	*unop expr* | *expr binop expr* | *expr*[*ilist*] | *exp*[[*idx*]]

Die Operatoren von

Syntaktischer Zucker & seine Bindungsfähigkeit

I	`$`			Komponentenauswahl	`list$item`
II	`[`	`[[`		Elementzugriff	`x[i]` `A[5,3]` `df$p[5]`
III	`^`			Exponentiation	`x^3`
IV	`-`			Minusvorzeichen	`-08.15` `-5^2`
V	`:`			Indexfolgen	`1:8` `5:-3` `-5:3`
VI	`%op%`			benutzerdefiniert	`x %*% y` `10%% 2`
VII	`*`	`/`		Punktrechnung	`8*4` `21/7` `883+0:5`
VIII	`+`	`-`		Strichrechnung	`17+4` `x[n-1]` `a*x+b`
IX	`<`	`<=`	`==`	Vergleichs-	`1+1 != 3`
	`>`	`>=`	`!=`	operatoren	`(x<y) == (y>x)`
X	`!`			Negation	`(!3==1) == TRUE`
XI	`&`	`&&`		Konjunktion	`p & (q | r)` `!a||b`
	`|`	`||`		Disjunktion	`is.vector(x) && plot(x)`
??	`~`			Modellformel	`z ~ (x1+x2):(z1+z2)`
XII	`<-`	`->`	`=`	Zuweisung	`13->x` `names(x)<-"Kevin"`

Die Datentypen von



Elementare und ...

```
logical()
integer()
numeric()
complex()
character()
```

komplexe Datentypen (Verbundobjekte)

Liste = Elemente + Attribute

Rudimentäre Objektorientierung:

- Hierarchisches Klassenattribut (Spezialisierungspfad)
- Polymorpher Methodenaufruf (Argument #1)

Atomare Datentypen

Es gibt **keine** skalaren Typen in 'R' — nur Vektoren der Länge 1

- **Datentyp** `NULL` *„undefiniert“*
  exkl.: `NULL`
- **Datentyp** `logical` Wahrheitswerte
  z.B.: `TRUE`, `FALSE` oder `NA`
- **Datentyp** `numeric` ganze und Gleitkommazahlen
  z.B.: `17`, `3.14`, `-1.2e6`, `pi`, `NaN`, `Inf`, `NA`
- **Datentyp** `complex` komplexe Zahlen
  z.B.: `2.13+1i`, `0-47.11i`, `2e-7i` oder s.o.
- **Datentyp** `integer` ganze Zahlen
  z.B.: `1:12`, aber nicht `17` usw.
- **Datentyp** `character` Buchstaben und Zeichenfolgen
  z.B.: `"Hello World!"`, `'a'`

Typabfrage und Typkonversion · Literale

- **Typidentifikation**
```
mode (3.141593)
[1] 'numeric'
```
- **Typverifikation**
```
is.character (47.11)
[1] 'FALSE'
is.numeric (47.11)
[1] 'TRUE'
is.complex (47.11)
[1] 'FALSE'
```
- **Typkonversion**
```
sqrt (as.complex (-1))
[1] 0+1i
sqrt (as.numeric (-1))
[1] NaN
```
- **Nullreferenz** `NULL`
```
is.null (NULL)      T
is.null (list())    F
is.null (integer(0)) F
```
- **Fehlanzeige** `NA`
```
is.na (NA)          T
is.na (883)         F
```
- **Unendlich** `Inf`
```
is.finite (pi)      T
is.infinite(pi/0)   T
is.infinite(1/0+1/0) T
is.infinite(1/0-1/0) F
```
- **Undefiniert** `NaN`
```
is.nan (0/0)        T
is.nan (1/0-1/0)    T
```


Erzeugung von Vektoren

- **Konstruktor**
```
vector (mode='numeric', length=12)
```
- **Konkatenation**
```
c (2,3,5,7,11,13,17)
c (12, c(4,5,6), 7, v4)
```
- **Arithmetische Progressionen**
```
1:8, 5:2
seq (1,17, by=2)
seq (1,17, length.out=50)
seq (along.with=c(2,3,5,7,11,13))
```
besser als `1:length(x)`
- **Wiederholung von Elementen und Folgen**
```
rep (x, times=5)
rep (x, length.out=17)
rep (x, each=5)
rep (x, times=y)
```
wie `rep(x,5)` oder `c(x,x,x,x,x)`

für `length(y) > 1`

Indizierter Zugriff auf Vektoren

- **Indexmenge** $= \{1, 2, \ldots, \ell\}$
  `length(x)`
  `length(x) <- lnew` (kürzen oder mit `NA`s auffüllen)
- **Einzelelemente**
  `x[5]`
- **Elementfolgen selektieren**
  `x[c(3,4,7)]`
  `x[3:5]`
- **Elemente unterdrücken**
  `x[c(-3,-4,-7)]`
  `x[-c(3,4,7)]` Syntaxfehler: `x[c(+2,-3)]`
- **Logische Indizierung**
  `(1:5) [c(FALSE,TRUE,TRUE,FALSE,TRUE)]` ⇝ `2 3 5`
  `(1:50) [c(FALSE,TRUE)]` ⇝ `2 4 6 8 10 12 14 ...`
  `gehalt [gehalt > 78000]`

Funktionen und Operatoren

- **Funktionen** von $\mathbb{R}^n$ nach $\mathbb{R}$

Summe/Produkt	`sum()`, `prod()`
Extremwerte	`max()`, `min()`
Statistik	`mean()`, `median()`, `var()`, `sd()`

- **Funktionen** von $\mathbb{R}^n$ nach $\mathbb{R}^n$

Logarithmen	`log()`, `log10()`, `log2()`
Rundung	`round()`, `signif()`, `trunc()`, `floor()`, `ceiling()`
Trigonometrie	`a/sin()`, `a/cos()`, `a/tan/2()`
Sonstige	`exp()`, `sqrt()`, `abs()`
Kumulative Berechnungen	`cum{sum,prod,max,min}()`

- **Operatoren** *(zweistellige Funktionen in Infixschreibweise)*

Strich- und Punktrechnung	`x+y`, `x-y`, `x*y`, `x/y`
Potenzbildung	`x^y` oder `x**y`
Ganzzahldivision/Rest	`x %/% y` bzw. `x %% y`

Beispiel: `'+' (1:2, 4:5)` ⇝ `5 7`

Vektorkomponenten mit Namen

- **Konstruktion benannter Vektoren**
  `x <- c (karl=6, heinz=28, mandy=17, .....)`
- **Namenlisten sind Vektorattribute**
  `namelist <- names(x)` *(Typ `character`)*
  `namelist <- attr (x, 'names')` *(dto.)*
- **Namen können geändert oder gelöscht werden**
  `names(x)[2] <- 'osama'` *(einzeln)*
  `names(x) <- NULL` *(alle)*
- **Komponenten lassen sich durch Namen indizieren**
  `x['mandy'] == x[3]` ⇝ `TRUE`
  `x[rep('karl',3)]` ⇝ `6 6 6`

Komplexwertige Vektoren

Konstruktor `complex()` und Projektionen `Re()`, `Im()`, `Mod()`, `Arg()` und `Conj()`

- **Konstruktion komplexer Vektoren**

```
complex (3)                                  0+0i 0+0i 0+0i
c (3i+2, 1-5i)                                    2+3i 1-5i
sqrt (-1:+1 + 0i)                            0+1i 0+0i 1+0i
is.complex (-08.15)                                  FALSE
```

- **Projektion komplexer Zahlen**

```
Re (3.0-4.0i)                                            3
Im (3.0-4.0i)                                           -4
Mod (3.0-4.0i)                                           5
Arg (3.0-4.0i)                                  -0.9272952
all.equal (sin (Arg (3.0-4.0i)), -4/5)                TRUE
Conj (3.0-4.0i)                                       3+4i
```

Wahrheitswerte

Vektoren vom Typ `logical`

- **Dreiwertige Logik in 'R'**
  `TRUE`, `FALSE` und `NA` — Literale
- **Vergleichsoperatoren**
  `==`, `!=`, `<=`, `>=`, `<`, `>` — arithmetisch oder lexikographisch
- **Logische Verknüpfungen**
  `!b` — einstellig
  `a&b`, `a|b`, `xor(a,b)` — zweistellig; *nicht* strikt!
  `all(b)`, `any(b)` — ∀/∃-Quantor
- **Bedingte Ausdrücke**
  `if (b) x else y` — skalare Wertverzweigung
  `ifelse (b, x, y)` — vektorielle Wertverzweigung
- **Selektion von Unterfeldern**

```
(1:8)[c(TRUE,FALSE)] == c(1,3,5,7)                TRUE
lebensabendspanne <- mean (alter[alter>65]) - 65
```


Vektorisierte Suchoperationen

Trefferpositionen und -meldungen für Vergleichsanfragen

- **Extremalposition (min/max)**

```
which.max (c(4,7,1,1))                    2
which.min ((-69:+96)^2)                  70
```

- **Logische Trefferpositionen**

```
which (LETTERS=='H')                      8
which (11:20 > 17)                  8 9 10
```

- **Test auf Wertegleichheit — Trefferpositionen**

```
match (x=4, table=abs (-5:+5))            2
match (x=c(8,8,3,0), table=12:1)  5 5 10 NA
```

- **Test auf Wertegleichheit — Treffermeldungen**

```
c(8,8,3,0) %in% 12:1          TRUE TRUE TRUE FALSE
c('S','RTFM') %in% LETTERS[1:20]      TRUE FALSE
```


Matrizen

Klasse `matrix` — Vektoren mit Dimensionsattribut

- **Konstruktoren**

```
A <- diag (5)                          Einheitsmatrix in ℝ^{5×5}
A <- diag (c(4,7,1,1))                 Diagonalmatrix in ℝ^{4×4}
matrix (data=NA, nrow=1, ncol=1, byrow=FALSE)
A <- matrix (1:12, ncol=3)             Matrix in ℝ^{4×3}
A <- matrix (1:3, 4, 3, byrow=TRUE)    Matrix in ℝ^{4×3}
```

- **Dimensionsattribut**

```
dim (A)                                [1] 4 3
nrow (A)                               [1] 4
ncol (A)                               [1] 3
length (A)                            [1] 12
dim(A) <- c(3,4)                       Todesstrafe!
```

Matrizen

Typ · Klasse · Transposition · Konversion

- **Typ und Klasse**

```
mode (A)                                  [1] 'numeric'
class (A)                                 [1] 'matrix'
is.matrix (A)                                 [1] TRUE
is.matrix (1:12)                             [1] FALSE
```

- **Transponieren einer Matrix**

```
dim (A)                                       [1] 4 3
dim (t(A))                                    [1] 3 4
```

- **Explizite und implizite Typkonversion**

```
dim (as.matrix (1:12))                        [1] 12 1
dim (t (1:12))                                [1] 1 12
as.vector (diag (4))                spaltenweise angeordnet
all (c (matrix (1:12,4,3)) == 1:12)           [1] TRUE
```


Matrizen

Selektion von Elementen, Zeilen, Spalten, Blöcken

- **Selektion eines Matrixelements**

```
A [i,j]                    Komponente A_ij (Zeile/Spalte)
A [k]                      k-tes Vektorelement nach Konversion
```

- **Selektion eines Matrixblocks**

```
A [i1:i2,j1:j2]            Block [A_ij]_{i1≤i≤i2}^{j1≤j≤j2}
A [i1:i2,]                 alle Spalten
A [,j1:j2]                 alle Zeilen
```

- **Automatische Dimensionsreduktion**

```
A [i,]                     (!) Vektor [A_ij]_{j=1..nc}
A [,j]                     (!) Vektor [A_ij]_{i=1..nr}
A [i,,drop=FALSE]          einzeilige Matrix
A [,j,drop=FALSE]          einspaltige Matrix
A [i,j,drop=FALSE]         einelementige Matrix
```


Matrizen

Selektion für Fortgeschrittene

- **Selektion einer beliebigen Matrixprojektion**

```
is.matrix (A [idx,jdx])                       [1] TRUE
```

$$(\boldsymbol{A}[\mathrm{idx},\mathrm{jdx}])\,[n,m] \;=\; \boldsymbol{A}[\mathrm{idx}[n],\mathrm{jdx}[m]]$$

passende Indizes; nicht notwendig aufsteigend; evt. Wdh.

- **Selektion mit Wahrheitswertmatrix**

```
is.matrix (A)                                 [1] TRUE
is.matrix (B)                                 [1] TRUE
is.logical (B)                                [1] TRUE
all (dim(A) == dim(B))                        [1] TRUE
```

Vektor aller TRUE-markierten Matrixelemente:

```
is.vector (A[B])                              [1] TRUE
```


Matrizen

Zeilen · Spalten · Diagonale

- **Rechnen mit Zeilen- und Spaltenindizes**

```
all (row(A)[i,] == i)                  TRUE für jedes i
all (col(A)[,j] == j)                  TRUE für jedes j
all (col(A) == t(row(t(A))))                       TRUE
A <- matrix(1:9,3); A[row(A)<col(A)]          [1] 4 7 8
```

- **Matrixdiagonale**

```
is.vector (diag (A))                               TRUE
all (x == diag(diag(x)))               TRUE für Vektoren
```

- **Zeilen untereinander stellen**

```
rbind (...)                Vektoren und/oder Matrizen
rbind (A,B)                Matrizen gleicher Spaltenzahl
rbind (x,y)                kürzerer Vektor wird wiederholt
rbind (A,y)                Matrix bestimmt Spaltenzahl
```

- **Spalten nebeneinander stellen**

```
cbind (...)                                (analog)
```

Matrizen

Multiplikation von Matrizen

- **Komponentenweise Matrixoperationen**
  `A+B`, `A*B`, ... *(siehe vector)*
- **Matrixmultiplikation**
  `ncol(A) == nrow(B)` konforme Matrixdimensionen
  `C <- A %*% B` liefert $C = A \cdot B$
  `nrow(C)==nrow(A)` TRUE
  `ncol(C)==ncol(B)` TRUE
- **Inversenbildung**
  `X <- solve (A)` A quadratisch, invertierbar
  `all (A %*% X == diag(ncol(A)))` i.a. nicht TRUE
- **Lösung linearer Gleichungssysteme**
  `X <- solve (A,B)` löst das LGS $A \cdot X = B$
  `all (A %*% X == B)` i.a. nicht TRUE

Matrizen

Multiplikation von Vektoren und Matrizen

- **Operator %*% berechnet immer eine Matrix**
  `class (1 %*% 1)` 'matrix'
- **Lineare Vektorabbildung**
  `A %*% y` $A \cdot y$ einspaltig
  `x %*% B` $x^\top \cdot B$ einzeilig
- **Inneres Vektorprodukt**
  `x %*% y` $x^\top y$ einelementig (!)
- **Äußeres Vektorprodukt**
  `x %o% y` $xy^\top$ dyadische Produktmatrix
  `outer (X, Y, FUN='*', ...)` Defaultfall = dto.
  `x %*% t(y)` weil `t(y)` eine Matrix ist

Lineare Algebra

QR-, Eigen- und Singulärzerlegung mit den Funktionen `qr()`, `eigen()` und `svd()`

- **QR-Zerlegung** $X=QR$, $Q^\top Q=E$, R ist OD-Matrix
  ```
  B <- matrix (1:12, nrow=3)
  o <- qr (B)
  all (qr.Q(o) %*% qr.R(o) == B)                TRUE
  ```
- **(Symmetrische) Eigenwertaufgabe** $S=U\Lambda U^\top$, $U^\top U=E$, $\Lambda=\mathrm{diag}(\lambda)$
  ```
  o <- eigen (S <- B %*% t(B))
  all (o$vec %*% diag(o$val) %*% t(o$vec) == S)  TRUE
  ```
  (auch Rechts- und Linkseigenvektoren nichtsymmetrischer Matrizen)
- **Singulärwertaufgabe** $X=VDU^\top$, $V^\top V=E$, $U^\top U=E$, $D=\mathrm{diag}(s)$
  ```
  o <- svd (B)
  all (o$v %*% diag(o$d) %*% t(o$u) == B)       TRUE
  ```
- **Determinante (quadratische Matrix)**
  ```
  det (diag(1:5)) == prod (1:5)                 TRUE
  determinant (diag(1:64), log=TRUE)   sign=1 modulus=205.1
  ```
  $\det(A) \equiv \mathtt{prod(eigen(A)\$val)} \equiv \mathtt{prod(diag(qr.R(qr(A))))}$

Distanzmatrizen

Klasse `dist` — symmetrische Matrix mit Nulldiagonale

- **Konstruktor**
  ```
  dist (x, method='euclidean', diag=F, upper=F)
  ```
  $$D[i,j] = d_{\text{method}}(x[i,], x[j,])$$
  'euclidean' $\|\cdot\|_2$, 'maximum' $\|\cdot\|_\infty$, 'manhattan' $\|\cdot\|_1$, 'canberra'
- **Klasse und Typ**
  ```
  all (dist(diag(5)) == sqrt (2))          [1] TRUE
  class (dist(diag(5)))                    [1] 'dist'
  mode (dist(diag(5)))                     [1] 'numeric'
  ```
- **Konversion zwischen Matrix und Distanz**
  `as.matrix (x)` redundante Quadratmatrix
  `as.vector (x)` das untere Dreieck seriell
  `as.dist (m, diag=F, upper=F)` unteres Dreieck

Felder beliebiger Dimension

Klasse `array (data=NA, dim=length(data), dimnames=NULL)`

- **Konstruktor**

```
a1 <- array (1:24, c(24))          1D-Feld
a2 <- array (1:24, c(6,4))         2D-Feld
a3 <- array (1:24, c(2,3,4))       3D-Feld
```

- **Typprüfung**

```
is.array (a3) etc.                 TRUE
is.vector (a1)                     FALSE
is.matrix (a2)                     TRUE
is.array (diag (5))                TRUE
```

- **Verallgemeinerte Transposition**

```
all (aperm (a2, c(2,1)) == t (a2))   TRUE
a3p <- aperm (a3, c(2,3,1))
a3[i,j,k] == a3p[j,k,i]              TRUE
dim (UCBAdmissions) (Datenbeispiel)  2 2 6
```


Listen

Klasse `list` — enthält benannte Komponenten unterschiedlichen Typs

- **Konstruktoren**

```
L <- list ()                       die leere Liste
L <- list (a=2,b=3,c=5)            Liste von numeric-Vektoren
```

- **Gemischt und geschachtelt**

```
L <- list (83.5, TRUE, c('hello','world'), diag(4))
M <- list (IQ=83.5, above=L, lily=list(17,TRUE))
```

- **Typprüfung**

```
is.list (list (cottbus=0:3))       ergibt TRUE
is.list (0:3)                      ergibt FALSE
```

- **Typwandlung**

```
as.list (c(2,3,5,7,11))            Vektoren elementweise
as.list (diag(5))                  Matrizen elementweise
as.list (plot.vector)              Funktionen: , Argliste, Rumpf
```


Listen

Selektion von Komponenten mit `'[[....]]'`

- **Zugriff mit Index**

```
L <- list (IQ=83.5, debil=T, lily=list(z=17,F))
L[[1]]                             [1] 83.5
```

- **Zugriff mit Namen**

```
L[['IQ']]                          [1] 83.5
L$IQ                               dto., eleganter
```

- **Zugriff wiederholt**

```
L$lily$z                             [1] 17
L$lily[[2]]                        [1] FALSE
L[['lily']][[2]]                   [1] FALSE
L[[3]][[2]]                        [1] FALSE
```

- **Zugriff wiederholt mit Indexvektor**

```
L[[c(3,2)]]                        [1] FALSE
L[[c('lily','z')]]                   [1] 17
```

Listen

Selektion von Teillisten mit '[....]'

- **Zugriff mit Indexfolge**

```
L <- list (IQ=83.5, debil=T, lily=list(z=17,F))
L[1:2]                    wie list (IQ=83.5, debil=T)
L[c(2,1)]                 wie list (debil=T, IQ=83.5)
```

- **Vorsicht: Einerlisten**

```
L[1]                              wie list (IQ=83.5)
L[3]                   wie list (lily=list(z=17,F))
```

- **Löschen von Listenelementen**

```
M <- list (a=1,b=2,c=3)
M$c <- NULL                       ergibt list (a=1,b=2)
M$b <- NULL                       ergibt list (a=1,c=3)
```

(Index bleibt fortlaufend ununterbrochen!)

Dataframes

Beispiel: 'http://stat.ethz.ch/Teaching/Datasets/NDK/sport.dat'

- Leseroutine akzeptiert Dateinamen, Eingabeströme und URLs

```
d.sport <- read.table ('sport.dat')
```

- Datensätze mit benannten Mustern & Merkmalen:

```
                weit kugel hoch disc stab speer punkte
OBRIEN          7.57 15.66 207 48.78 500 66.90 8824
BUSEMANN        8.07 13.60 204 45.04 480 66.86 8706
DVORAK          7.60 15.82 198 46.28 470 70.16 8664
FRITZ           7.77 15.31 204 49.84 510 65.70 8644
HAMALAINEN      7.48 16.32 198 49.62 500 57.66 8613
NOOL            7.88 14.01 201 42.98 540 65.48 8543
ZMELIK          7.64 13.53 195 43.44 540 67.20 8422
GANIYEV         7.61 14.71 213 44.86 520 53.70 8318
PENALVER        7.27 16.91 207 48.92 470 57.08 8307
HUFFINS         7.49 15.57 204 48.72 470 60.62 8300
PLAZIAT         7.82 14.85 204 45.34 490 52.18 8282
MAGNUSSON       7.28 15.52 195 43.78 480 61.10 8274
SMITH           7.47 16.97 195 49.54 500 64.34 8271
MUELLER         7.25 14.69 195 45.90 510 66.10 8253
CHMARA          7.75 14.51 210 42.60 490 54.84 8249
```

- Auswahl von Zeilen und Spalten, z.B. `d.sport[,'kugel']`

```
 [1] 15.66 13.60 15.82 15.31 16.32 14.01 13.53 14.71 16.91 15.57 14.85 15.52
[13] 16.97 14.69 14.51
```


Dataframes

Klasse `data.frame` — enthält Vektoren gleicher Länge

- **Klasse und Dimensionen**

```
data (iris)                                  der Iris-Datensatz
class (iris)                                   'data.frame'
dim (iris)                                       [1] 150 5
ncol (iris)                                        [1] 5
nrow (iris)                                      [1] 150
```

- **Spalten- und Zeilennamen**

```
names (iris)     [1] 'Sepal.Length' 'Sepal.Width' ... 'Species'
colnames (iris)                                            dto.
rownames (iris)                        [1] 1 2 3 4 5 6 7 ... 150
```

- **Konstruktor**

```
daf <- data.frame (x=1:3,
ch=c('shoo','bee','doo'), cpl=rep(2i,3))
```

Dataframes

Attributvektoren gleicher Länge, aber unterschiedlichen Typs

- **Selektion der Attributvektoren**

```
iris$Sepal.Length          [1] 5.1 4.9 4.7 4.6 5.0 ... 5.9
iris[[1]]                  [1] 5.1 4.9 4.7 4.6 5.0 ... 5.9
```

- **Attributvektoren vom Typ 'factor'**

```
iris$Species          [1] setosa setosa setosa ... virginica
```

- **Attributnamen als lokale Variable**

```
attach (iris)                                  Namen zuordnen
Sepal.Length               [1] 5.1 4.9 4.7 4.6 5.0 ... 5.9
detach (iris)                                  Namen entfernen
```

- **Selektion von Teildatensätzen wie 'matrix'**

```
iris[,3]                   nicht wie iris$Petal.Length
iris[17,]                          das Muster der 17. Zeile
iris[11:20,3:5]      Datensatz mit 10 Zeilen, 3 Merkmalen
iris[3:5]                          Vorsicht: Listenselektion!
```


Faktoren (Klasse factor)

Speicherökonomische Darstellung *kategorialer* Variablen

- **Faktor ≙ integercodierter Wertevektor**

```
data (iris); attach (iris);                    der Iris-Datensatz
print (Species)          [1] setosa setosa setosa ... virginica
class (Species)                                  [1] 'factor'
as.vector (Species)      [1] 'setosa' 'setosa' ... 'virginica'
unclass (Species)              [1] 1 1 1 1 ... 2 2 ... 3 3 3
as.integer (Species)           [1] 1 1 1 1 ... 2 2 ... 3 3 3
```

- **Codebuch eines Faktors**

```
levels (Species)         [1] 'setosa' 'versicolor' 'virginica'
length (levels (Species))                              [1] 3
class (levels (Species))                   [1] 'character'
```


Faktoren (Klasse factor)

Speicherökonomische Darstellung *kategorialer* Variablen

- **Konstruktor**

```
factor (c(T,T,T,F,F,T))  [1] TRUE TRUE TRUE FALSE FALSE TRUE
factor (c(5,4,7,5,4,7))                      [1] 5 4 7 5 4 7
unclass (factor (c(5,4,7,5,4,7)))            [1] 2 1 3 2 1 3
factor (c('a','b','b','a'))                     [1] a b b a
```

- **Typcheck und Konversion**

```
is.factor (Species)                                [1] TRUE
as.factor (5:8)                          wie factor (5:8)
```

- **Manipulation des Codebuchs: Reihenfolge**

```
factor (c(T,F,F), levels=c(T,F))     [1] TRUE FALSE FALSE
unclass (factor (c(T,F,F)), levels=c(T,F))   [1] 1 2 2
unclass (factor (c(T,F,F)), levels=c(F,T))   [1] 2 1 1
```

- **Manipulation des Codebuchs: Wertebereich**

```
unclass (factor (3:5))                             [1] 1 2 3
unclass (factor (3:5, levels=1:5))                 [1] 3 4 5
```

Faktoren ≙ Musterklassen

Etikettierte Stichprobe ≙ Dataframe ($N\times$ `numerical` & $1\times$ `factor`)

- **Klasseninformation als letzte Variable**
  `iris [[length(iris)]]` der Faktor
  `iris [length(iris)]` nicht der Faktor
  `iris [-length(iris)]` die Merkmale
  `as.matrix (iris [-length(iris)])` die Merkmalmatrix
- **Selektion von Teilstichproben**
  `iris [47:11,]` die Muster 11–47 rückwärts
  `iris [Species=='setosa',]` die Muster 1–50
- **Berechnung klassenweiser Statistiken**
  `mean (iris [Species=='setosa','Petal.Width'])` `[1] 0.246`
  `var (iris [Species=='setosa','Petal.Width'])` `[1] 0.011`
- **Faktoren und Volkszählung**
  `table (Species)` `[1] 50 50 50`

Nicht vektorisierte Kontrollstrukturen

Nur für kleine, äußere Schleifen verwendbar !!!

- **Einseitige Wertverzweigung**
  `if (COND) EXPR` `logical[1]`, `expression[1]`
- **Zweiseitige Wertverzweigung**
  `if (COND) EXPR.1 else EXPR.2`
- **Gezählte Wiederholung**
  `for (VAR in SEQ) EXPR` Liste/Vektor $1\times$ zu Beginn ausgewertet
  Weiterschaltung mit `next`
- **Abweisende Wiederholung**
  `while (COND) EXPR` `logical[1]`, `expression[1]`
- **Unbedingte Wiederholung**
  `repeat EXPR` Ausbruch mit `break`

Bemerkung

Klammern `{` und `}` um zusammengesetzte Ausdrücke nicht vergessen!
Kein Zeilenvorschub vor `else` einfügen (syntaktisch ambig)!
Kontrollstrukturen besitzen einen Wert ➪ letzter Ausdruck in letztem Durchlauf

Vektorisierte Kontrollstrukturen

Fallunterscheidung und komponentenweise Wertverzweigung

- **Vektorisierte zweiseitige Wertverzweigung**
  `x <- ifelse (test, yes, no)` $1\times$ `logical`, $2\times$ `expression`
  drei Vektoren gleicher [...] Länge
- **Fallunterscheidung nach Positionen**
  `x <- switch (EXPR, exp1, exp2, exp3, .....)` `integer[1]`
- **Fallunterscheidung nach Namen**
  `x <- switch (EXPR, nam1=exp1, nam2=exp2, nam3=exp3, .....)`
  `character[1]`
- **Fallunterscheidung mit Voreinstellung**
  (nur Namen; ohne Voreinstellung ist `NULL` Default)
  `x <- switch (EXPR, nam1=exp1, nam2=exp2, ....., exp.def)`
- **Fallunterscheidung mit Mehrfachklauseln**
  `x <- switch (EXPR, ....., nam1=, nam2=, nam3=exp, .....)`

Vektorisierte Iteration über Listen und Vektoren

Iterationsrumpf wird nur 1× geparst!

- **Mehrfache ($n\times$) Auswertung eines Ausdrucks**
  `replicate (n, expr, simplify=TRUE)`
  Resultat ≙ Liste oder Vektor/Matrix (`simplify=F/T`)
- **Funktionsanwendung auf Listenelemente**
  `lapply (X, FUN, ...)` die Argumente `...` werden der Fkt. `FUN` serviert
  Resultat ≙ Liste der `FUN(X[[i]])`
- `lapply (list (4,9,16), sqrt)` `list (2,3,4)`
  `lapply (list (4,9,16), '-', 3)` `list (1,6,13)`
- **Funktionsanwendung auf Listen- oder Vektorelementen**
  `sapply (X, FUN, ..., simplify=TRUE, USE.NAMES=TRUE)`
  Resultat je nach `simplify`; ggf. werden die `X`-Namen eingebunden
- `sapply (1:5, sqrt)` Vektor mit Wurzeln
  `sapply (1:5, rep, 3)` (3×5)-Matrix
  `sapply (iris[,-5], mean)` Mittelwertvektor

'R'-Syntax ?vector ?matrix ?list ?data.frame ?factor ?Control function() class()

Vektorisierte Iteration über mehrdimensionale Felder

`apply (X, MARGIN, FUN, ...)`

- **Funktionsanwendung auf Matrixzeilen**
  `apply (iris[,-5], MARGIN=1, FUN=max)` 150 Zeilenmaxima
- **Funktionsanwendung auf Matrixspalten**
  `apply (iris[,-5], MARGIN=2, FUN=mean)` 4 Spaltenmittel
  `apply (iris[,-5], MARGIN=2, FUN=range)` (2×4)-Matrix
- **Funktionsanwendung auf Matrixelemente**
  `apply (iris[,-5], MARGIN=c(1,2), FUN='/', 100)` Umrechnung [cm] in [m]
  `apply (iris[,-5], MARGIN=c(1,2), FUN=rep, 3)` $(3 \times 150 \times 4)$-Kubus
- **Hinausrechnen von Statistiken**
  `sweep (x, MARGIN=1, STATS=a, FUN='-')` subtrahiert a_i von Zeile i
  `sweep (x, MARGIN=2, STATS=s, FUN='/')` dividiert Spalte j durch s_j
- **Spezialwerkzeug:** $\mu = 0$ **und/oder** $\sigma = 1$
  `scale (x, center=TRUE, scale=TRUE)`

'R'-Syntax ?vector ?matrix ?list ?data.frame ?factor ?Control function() class()

Iterierte Funktionsanwendung mit mehreren Variablen

`mapply (FUN, ..., MoreArgs=NULL)`

- **Verallgemeinert** `sapply`
  `mapply (FUN=sqrt, 1:8)` wie `sapply (X=1:8, FUN=sqrt)`
- **Funktionen mit zwei und mehr Argumenten**
  (alle Argumentvektoren sind von gleicher Länge)
  `mapply (FUN='+', 1:4, 4:1)` ergibt `[1] 5 5 5 5`
  `mapply (FUN=rep, 1:4, 4:1)` $\{\{1,1,1,1\},\{2,2,2\},\{3,3\},\{4\}\}$
- **Benannte Argumente können adressiert werden**
  `mapply (FUN=rep, times=1:4, x=4:1)`
  $\{\{4\},\{3,3\},\{2,2,2\},\{1,1,1,1\}\}$
- **Weitere benannte Argumente mit Konstanten belegen**
  `mapply (FUN=rep, times=1:4, MoreArgs=list(x=8))`
  $\{\{8\},\{8,8\},\{8,8,8\},\{8,8,8,8\}\}$

'R'-Syntax ?vector ?matrix ?list ?data.frame ?factor ?Control function() class()

Funktionsanwendung auf faktorgruppierte Datenvektoren

`tapply (X, INDEX, FUN=NULL, ..., simplify=TRUE)`

- **Klassenweise Mittelwertbildung**
  `tapply (iris[[2]], iris[[5]], mean)` nur 1 Faktor
- **Auch Wahrheitswerte werden Faktoren**
  `z <- runif(100); tapply (z, z>0.5, sum)` 2 Levels
- **Warum nicht auch mehrere Faktoren?**
  `tapply (iris[[2]], list (iris[[5]], iris[[3]]<5), length)` 2D-Tabelle
- **Spezialwerkzeuge für Dataframes**
  berechnet Merkmalstatistiken für alle Gruppen
  `aggregate (x, by, FUN, ...)` `x` Datensatz, `by` Faktorliste
  `by (data, INDICES, FUN, ..., simplify=T)` ⇝ `by`-Objekt
- **Spezialwerkzeug für Vektoren**
  `ave (x, ..., FUN=mean)` `x` Vektor, `...` Faktoren

Sonstige Schleifenersatzfunktionen

- **Summenbildung**
  `rowSums (x)` entspricht `apply (x, MARGIN=1, FUN=sum)`
  `colSums (x)` entspricht `apply (x, MARGIN=2, FUN=sum)`
- **Mittelwertbildung**
  `rowMeans (x)` entspricht `apply (x, MARGIN=1, FUN=mean)`
  `colMeans (x)` entspricht `apply (x, MARGIN=2, FUN=mean)`

  Argument `na.rm=F` zur `NA`-Feinsteuerung
  Argument `dim=1` für höherdimensionale Felder
- **Kumulative Arithmetik**

`cumsum (x)`	$y_k := \sum_{i=1}^{k} x_i$
`cumprod (x)`	$y_k := \prod_{i=1}^{k} x_i$
`cummin (c(3:1,2:0,4:2))`	3 2 1 1 1 0 0 0 0
`cummax (c(3:1,2:0,4:2))`	3 3 3 3 3 3 4 4 4
`diff ((0:8)^2)`	1 3 5 7 9 11 13 15

- **Äußeres Produkt**, z.B. Vandermonde-Matrix:
  `outer (X=z, Y=seq(along=z)-1, FUN="^")` $V_{ij} := z_i^{j-1}$

Vektorisierte Feldkomponentenselektion

Schleifenfreier Zugriff auf Elementesequenz nach Agendamatrix

- **(Konter)diagonale einer Matrix**

```
(A <- matrix (1:25, 5, 5))
```

$$\begin{bmatrix} 1 & 6 & 11 & 16 & 21 \\ 2 & 7 & 12 & 17 & 22 \\ 3 & 8 & 13 & 18 & 23 \\ 4 & 9 & 14 & 19 & 24 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$$

`diag (A)`	1 7 13 19 25
`A[cbind(1:5,1:5)]`	1 7 13 19 25
`A[cbind(5:1,5:1)]`	25 19 13 7 1
`A[cbind(1:5,5:1)]`	21 17 13 9 5
`A[cbind(5:1,1:5)]`	5 9 13 17 21

- **Anwendungsbeispiel: Travelling Florist Problem**

`D <- as.matrix (dist (iris[1:4]))`	150×150-Distanzmatrix
`p <- sample (nrow (iris))`	65 95 108 114 25 ... ... 141 126
`path <- cbind (p, c(p[-1],p[1]))`	65,95 95,108 ... 141,126 126,65
`sum (D[path])`	383.8539 *[kumulative Distanz]*

- **Analoge Vorgehensweise für `array`-Objekte ...**

Deklaration und Aufruf von Funktionen

In 'R' sind Funktionen *„Objekte erster Klasse“*

- Funktionsdeklaration mit formalen Parametern:
  `funcname <- function (arglist) {body}`
  - Parameterbezeichner *`vname`*
  - Parameter mit Voreinstellung *`vname=default`*
  - Restparameterliste `...`
- Funktionsaufruf mit aktuellen Parametern:
  `funcname(arglist)`
  - Positionelle Übergabe *`expr`*
  - Namentliche Übergabe *`vname=expr`*
  - Restparameterübergabe `...`
- Beispiel:

```
zeichne <- function (x, y=NULL, title='Grafik', ...) {
           if (is.null (y))
                   { y <- x; x <- 1:length(x) }
           plot (y ~ x, main=title, ...)
           }

zeichne (sin(1:20), cos(1:20), title='Kreis', col='red')
```

Hilfsmittel zur Funktionsdeklaration

- **Konstruktor**
  `function («arglst») «bodyexpr»` Wert ≙ Funktionsobjekt
- **Rückgabewert** (`return` vs. `invisible`)
  `function (x) { x2 <- x^2; sqrt(sum(x2)) }` (der letzte Ausdruck)
  `function (x) return (sqrt(sum(x^2)))` (explizit: ± geschwätzig)
  `function (x) { y <- x%o%x; z <- y%o%y; invisible(z) }`
- **Abbruch, Warnung, Zusicherung**
  `stop ("Halt!","mich!","an!")` (Abbruch, Meldung, Position)
  `warning ("Das wird","teuer!")` (Meldungen werden akkumuliert)
  `stopifnot (is.matrix (S))` (bei Verstoß Abbruch und Report)
- **Benutzerdeklarierte Operatoren** z.B.: `%%`, `%/%`, `%*%`, `%o%`, `%in%`, ...
  `"%xor%" <- function (x,y) x != y`
  `c(T,T,F,F) %xor% c(T,F,T,F)` `FALSE TRUE TRUE FALSE`
- **Benutzerdeklarierte Zuweisungsoperatoren**
  Parametername immer `value` für RHS-Objekt!
  `"plus<-" <- function (x,value) x <- x+value`
  `z <- 1:5; plus(z) <- 10; print(z)` `11 12 13 14 15`

Informationen über Funktionsobjekte

Nur Psychopathen manipulieren einen Kantorovic-Baum!

- **Liste formaler Funktionsparameter**
  `formals (ls)` oder `formals ("ls")` (Argumentliste: Name/Default)
  `args (fun)` (dto., aber in Textform)
- **Funktionsrumpf als 'R'-Sprachobjekt**
  `body (fun)` (Objekt der Klasse `name`, `expression` oder `call`
  `is.language (body (fun))` `TRUE`
- **Online-Dokumentation abfragen**
  `help (fun)` oder `?fun` Hilfetext zu Funktion
  `help.search (pattern)` oder `??pattern` Hilfetext zu Stichwort
  `apropos (what=«pattern»)` Objektliste mit Treffern
  `example (fun)` Beispielaufrufe ausführen

Klasse und Datentyp

'R' kam nicht als objektorientierte Sprache auf die Welt

- **Atomarer Typ der Komponenten eines Feldes**
  `mode (x)` `character[1]`
- **Klasse eines 'R'-Objekts**
  `class (x)` `character[L]`
  (1) explizite Klasse: erste Komponente von `attr(x,'class')`
  (2) implizite Klasse: Matrix/Array; je nach `length(dim(x))`
  (3) implizite Klasse: `mode(x)` für Vektoren
- **Test auf Abstammung von einer Klassenauswahl**
  `inherits (x, what, which=FALSE)` `logical[1?L]`
- **Verleihen des Klassenattributs durch Zuweisung**
  `class(x) <- c('myofb','lol','imho')`
- **Reduktion auf elementaren Typ [...]**
  `y <- unclass (x)`

Klasse und Datentyp

Beispiele zur Orientierung in einer feindseligen Programmierumgebung

Objekt x	Typ mode(x)	Klasse class(x)	Reduktion class(unclass(x))
NA	logical	logical	logical
883	numeric	numeric	numeric
1618:1648	numeric	integer	integer
1618:1648/17	numeric	numeric	numeric
diag(7)	numeric	matrix	matrix
diag(7)%o%diag(7)	numeric	array	array
iris	list	data.frame	list
iris$Species	numeric	factor	integer
print	function	function	function
'+'	character	character	character

Betrachten von Objektinhalten

Methoden zur textuellen und graphischen Ausgabe

- **Textuelle Standardanzeige**
  `print (1:3)` oder `1:3` oder `(x <- 1:3)` `[1] 1 2 3`
  `all (print(iris) == iris)` `TRUE`
- **Kompakte, aber erschöpfende Inhaltsangabe**
  `str (iris)`

```
'data.frame': 150 obs. of 5 variables:
$ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
$ Sepal.Width:  num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
$ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
$ Petal.Width:  num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
$ Species:      Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

- **Semantische Objektzusammenfassung**
  `summary (rnorm (1000))` primär für Datenmodellierungsobjekte

Minimum	1st Quant.	Median val.	Mean val.	3rd Quant.	Maximum
-2.856	-0.634	-0.019	-0.026	0.601	2.408

- **Graphische Standardanzeige**
  `plot (iris)`

Objektattribute

Attribut $\hat{=}$ Name *(Zeichenkette)* + Wert *('R'-Objekt)*

- **Abfrage und Änderung**
  `attr (x, which="dim") <- 4:5` `<- NULL` um Attribut zu löschen
  `attr (x, which="dim")` oder `dim (x)` `4 5`
- **Alle Attribute auf einmal**
  `attributes (x) <- value` und `attributes (x)`
- **Allgemeine Objektkomposition**
  `structure (.Data, ...)` Zusatzargumente in `name=value`-Form
- **Standardattribute und ihre Abfrage/Zuweisung**

class	dim	dimnames	names	row.names	levels	comment
class()	dim()	dimnames()	names()	row.names()	levels()	comment()
	length()		colnames()	rownames()		
character	integer	character	character	character	character	?

Name, Wert und Namensraum

Ein Name ist eine Zeichenkette & ist doch keine Zeichenkette

- **Objektname versus Zeichenkette**
  `z <- levels(iris$Species); z == "z"` `FALSE FALSE FALSE`
- **Namensbindung eines 'R'-Objekts verfolgen**
  `get ("z")` `"setosa" "versicolor" "virginica"`
- **Neuen Namen an 'R'-Objekt binden**
  `assign (x="neu", value=z); neu[3]` `"virginica"`
- **Funktionsrümpfe bilden einen Namensraum**
  `setx <- function (val) x <- val`
  `x <- "mega-out"; setx (4711); print (x)` `"mega-out"`
- **Globaler Zuweisungsoperator '<<-'**
  `gsetx <- function (val) x <<- val`
  `x <- "mega-out"; gsetx (4711); print (x)` `4711`

Zuweisung — Bindung oder Kopie?

- **Zuweisungsoperator (LHS ≙ 'R'-Objekt)**
  `A <- matrix (rnorm(15), 5, 3)` Bindung der RHS an `A`
  `c(0,8,15) -> b` oder `b = c(0,8,15)`
  `A <- b` kein Problem! (Matrix hingerichtet)
- **Zuweisungsoperator (LHS ist ein 'fun<-'-Aufruf)**
  `A[2,] <- b` kopiert in zweite Zeile
  `A[,3] <- b` dritte Spalte zu lang!
  `A[6:8] <- b` Matrix ⇝ Vektor
  `A[] <- b` Glück gehabt! (3 teilt 15)
- **Nutzen klassenerhaltender Zuweisung**

```
C <- matrix (6:1, 2, 3)
sort (C)
C[] <- sort (C); print (C)
```

$$\begin{bmatrix} 6 & 4 & 2 \\ 5 & 3 & 1 \end{bmatrix}$$
1 2 3 4 5 6
$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Deklaration generischer Methoden

- **Deklaration einer generischen Funktion**
  `foo <- function (x, y, ...) UseMethod ('foo', x)`
- **Delegieren eines generischen Aufrufs**
  `foo (x, ...)` mit `class(x)` = `c(c1,c2,...,cn)`
  initiiert folgende Kette von Delegierungsversuchen:
  `foo.c1 (x, ...)` die Klasse von `x`
  `foo.c2 (x, ...)` eine Oberklasse von `x`
  ⋮
  `foo.cn (x, ...)` höchste Oberklasse von `x`
  `foo.default (x, ...)` letzte Chance; unbedingt deklarieren !!
- **Welche Methoden sind aktuell deklariert?**
  `methods ('print')` alle Methoden `print.classname`
  `methods (class='matrix')` alle Methoden `fctname.matrix`

Generische Methoden und polymorpher Aufruf

Generischer Aufruf
`plot (iris$Species)`

1. Delegieren an Funktion `plot.factor`?
2. Delegieren an Funktion `plot.integer`?
3. Delegieren an Funktion `plot.default`?

Wir basteln uns eine Graubildklasse

- **Warum? — Darum!**
  `plot (diag (69))` `plot.matrix` verwendet nur die Spalten 1 und 2
- **Konstruktor für die `BILD`-Klasse**

```
BILD <- function (x) {
     x <- matrix (as.integer(cut(x,256))-1, ncol=ncol(x))
     class(x) <- "BILD"
     invisible(x) }
```

- **Plotmethode für die `BILD`-Klasse**

```
plot.BILD <- function (x)
          image (x, col=gray(0:255/255))
```

- **Aufruf der Rasterplotmethode**
  `plot.BILD (diag (69))` oder `plot (BILD (diag (69)))`

Zusammenfassung (2)

1. 'R' ist eine **funktionale** Programmiersprache, besitzt aber zahlreiche **Infixoperatoren** als syntaktischen Zucker.
2. Atomare 'R'-Datentypen sind die **Vektoren** (numerisch, logisch, komplex, Strings); dazu gibt es flexible **Indexierungsmechanismen** und **komponentenweise** Arithmetik.
3. **Matrizen** und mehrdimensionale Felder sind als Vektoren mit **Dimensionsattribut** realisiert; Unterstützung der **linearen Algebra**.
4. **Listen** sind hierarchische Verbundobjekte. Wie auch die Felder unterstützen sie **Komponentennamen**.
5. **Datensätze** enthalten numerische, aber auch kategoriale (**Faktoren**) Merkmale.
6. Als **interpretative** Sprache gebietet 'R' den Gebrauch **vektorisierter** statt traditioneller **Kontrollstrukturen** (*„Schleifen“*).
7. In 'R' sind Funktionen **Datenobjekte** erster Klasse.
8. 'R' bietet rudimentäre **Objektorientierung** durch **Klassenattribute** und **polymorphen** Funktionsaufruf (Argument #1).