Wolfgang Bischof

Technische Hochschule Augsburg

Stochastik

Kapitel 0 R Grundlagen

Stand: 20. März 2024

Definition und Zuweisung

- Das Herzstück von R bilden die Vektoren.
- Alle Elemente eines Vektors müssen den gleichen Datentyp besitzen.
- Ein Vektor kann also entweder aus Zahlen (Datentyp: numeric) oder aber aus Zeichen(ketten) (Datentyp: character) oder aber aus Wahrheitswerten (Datentyp: logical) bestehen. Mischungen von Zahlen, Zeichenketten und Wahrheitswerten sind nicht erlaubt.
- Die einfachste Variante, einen Vektor zu definieren, ist die Aufzählung aller Elemente.
- Dabei steht c für *concatenate* oder *combine* und <- ist der Zuweisungoperator.

```
x <- c(2,3,5)
```

- Genau genommen fügt c drei einelementige Vektoren zu einem neuen Vektor zusammen, da jede Zahl als einelementiger Vektor betrachtet wird.
- Da mit c also Vektoren verknüpft werden, können wir mit

```
y <- c(x,4,x)
```

einen neuen Vektor

```
y
## [1] 2 3 5 4 2 3 5
```

definieren, der sieben Elemente enthält.

[ABC 1]

Folgen

```
-3:7                        # Vektor in 1er Schritten
##  [1] -3 -2 -1  0  1  2  3  4  5  6  7
1.3:6
## [1] 1.3 2.3 3.3 4.3 5.3
seq(0, 5, 0.5)              # Vektor in 0.5er Schritten
##  [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
seq(2,6, length.out = 12) # Vektor der Länge 12 (äquidistant, d.h. Abstand=4/11)
##  [1] 2.00 2.36 2.73 3.09 3.45 3.82 4.18 4.55 4.91 5.27 5.64 6.00
rep(1:3, 4)                 # Wiederholt den Vektor 1, 2, 3 viermal
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3
rep(1:3, each = 4)          # Wiederholt 1, 2, 3 je viermal
##  [1] 1 1 1 1 2 2 2 2 3 3 3 3
```

Operatoren und elementweise Funktionen

Die Operatoren und Funktionen in der folgenden Tabelle werden alle elementweise ausgeführt.

Operator, Funktion	Bedeutung
`+` / `-` / `*` / `/`	Addition, Subtraktion, Multiplikation, Division
`%%` / `%/%`	Modulo / Ganzzahlige Division
`^` / `**`	Potenzieren: `a^b` / `a**b` berechnet a^b
`<` / `<=` / `==` / `>` / `>`	Vergleichsoperatoren
`abs()` / `sign()`	Betragsfunktion / Algebraisches Vorzeichen
`sqrt()`	Quadratwurzel
`sin()` / `cos()` / `tan()`	Trigonometrische Funktionen
`asin()` / `acos()` / `atan()`	Arkusfunktionen (Umkehrfunktionen)
`exp()`	Exponentialfunktion
`log()` / `log10()` / `log2()`	Logarithmus zur Basis e / 10 / 2
`factorial()`	Fakultät (eigentlich: $\Gamma(x+1)$)
`round(`$\langle n \rangle$`, digits=`$\langle k \rangle$`)`	Rundet kaufmännisch $\langle n \rangle$ auf $\langle k \rangle$ Stellen nach dem Komma
`floor()` / `ceiling()`	Abrunden / Aufrunden

```
# Operatoren werden elementweise ausgeführt
c(1,2,3) + c(4,5,6)

## [1] 5 7 9

c(1,2,3)^2

## [1] 1 4 9

c(1,2,3) > c(2,2,2)

## [1] FALSE FALSE  TRUE

# die elementweisen Funktionen ebenso
sqrt(c(1,4,9))

## [1] 1 2 3

round(c(0.5,1.5,2.5,3.5,3.6,4.1,4.4))

## [1] 0 2 2 4 4 4 4
```

Recycling-Regel

- Ein praktisches Konzept im Umgang mit Vektoren ist die Recycling-Regel: Wenn ein Operator zwei Vektoren miteinander verknüpft und diese Verknüpfung nur für Vektoren gleicher Länge erlaubt ist, werden die Elemente des kürzeren Vektors automatisch solange wiederholt, bis die Länge des ersten Vektors erreicht ist:

```
c(1, 4) + c(20, 30, 40, 50)
```

```
## [1] 21 34 41 54
```

```
# ist damit das Gleiche wie
c(1, 4, 1, 4) + c(20, 30, 40, 50)
```

```
## [1] 21 34 41 54
```

- Ist der längere Vektor kein ganzzahliges Vielfaches des kürzeren Vektors, so wird eine Warnung (kein Fehler) ausgegeben.
- Der resultierende Vektor hat immer die Länge des längeren Vektors.
- Die Recycling-Regel ist vor allem bei Vergleichen sehr nützlich.

```
# die 10 wird sechsmal genutzt
1:6 + 10
```

```
## [1] 11 12 13 14 15 16
```

```
# der Vektor (1, 100) wird dreimal genutzt
1:6 + c(1,100)
```

```
## [1]   2 102   4 104   6 106
```

```
# 6 ist kein Vielfaches von 4, daher Warnung
1:6 + c(1,10,100,1000)
```

```
## Warning in 1:6 + c(1, 10, 100, 1000): Länge des
längeren Objektes
##  ist kein Vielfaches der Länge des kürzeren
Objektes
```

```
## [1]    2   12  103 1004    6   16
```

```
# Wer ist mindestens 18 Jahre alt?
alter <- c(13, 24, 18, 45, 98, 8)
alter >= 18 # die 18 wird fünfmal recycelt
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
```

Vektorwertige Funktionen

R-Funktionen, welche für die Arbeit mit Vektoren konzipiert sind und einen Vektor als Eingabe erwarten, werden als Vektorfunktionen bezeichnet:

Vektorfunktion	Bedeutung
`sum()` / `prod()`	Summe / Produkt der Elemente eines Vektors (das klappt auch für Vektoren aus Wahrheitswerten, dabei wird TRUE in 1 und FALSE in 0 umgewandelt)
`cumsum()`	kumulierte Komponentensumme
`cumprod()`	kumuliertes Komponentenprodukt
`length()`	Länge des Vektors
`min()`; `max()`	Minimum/Maximum der Komponenten
`range()`	kleinster und größter Wert
`which.min()`	Index des kleinsten erst vorkommenden Wertes
`which.max()`	Index des größten erst vorkommenden Wertes
`which(Bedingung)`	gibt die Indizes aller Vektorelemente aus, welche Bedingung erfüllen
`which(x == min(x))`	alle Indizes der kleinsten Werte

```
(v <- c(3,4,5,1,1)) # print wg. äußerer Klammern

## [1] 3 4 5 1 1

sum(v); prod(v);

## [1] 14
## [1] 60

sum( v>3 ) # Anz. Elemente, die größer als 3 sind

## [1] 2

cumsum(v); length(v); range(v)

## [1]  3  7 12 13 14
## [1] 5
## [1] 1 5

which.min(v); which.max(v); which( v==min(v) )

## [1] 4
## [1] 3
## [1] 4 5
```

Vektorwertige Funktionen (Teil 2)

Vektorfkt.	Bedeutung
`diff()`	Differenz/Abstand zum folgenden Element („hinten minus vorne“); Die Option `lag` legt den Index-Abstand zur Bildung der Differenzen fest (Default: 1)
`sort()`	geordneter Vektor (Bsp.: geordnete Zeiten beim Wettlauf)
`order()`	Permutation der Indizes, so dass geordneter Vektor entsteht (Bsp.: Startnummern der nach Zeiten geordneten Läufer)
`rank()`	Ränge der Vektoreinträge (Bsp.: Ränge der nach Startnummern geordneten Läufer)
`lapply()`	elementweises Anwenden einer beliebigen Funktion, bei der das Ergebnis eine Liste ist

```
(v <- c(3,4,5,1,1))

## [1] 3 4 5 1 1

diff(v); diff(v, lag=2);

## [1]  1  1 -4  0
## [1]  2 -3 -4

sort(v); sort( v, decreasing = TRUE ) # absteigend

## [1] 1 1 3 4 5
## [1] 5 4 3 1 1
```

```
order(v); rank(v)

## [1] 4 5 1 2 3
## [1] 3.0 4.0 5.0 1.5 1.5

n <- c(2,1,3)
# rexp(m) liefert m zufällige Bediendauern
# Bediendauern von 2/1/3 Kunden
(bediendauern <- lapply(n, rexp))

## [[1]]
## [1] 0.336 0.963
##
## [[2]]
## [1] 0.344
##
## [[3]]
## [1] 0.35 1.13 1.34

unlist(bediendauern) # ... als Vektor

## [1] 0.336 0.963 0.344 0.350 1.126 1.337
```

Vektorwertige Funktionen (Teil 3)

Für die deskriptive Statistik stehen eine Vielzahl von Funktionen zur Verfügung, wie z.B.:

Vektorfunktion	Bedeutung
`mean()`	Arithmetisches Mittel
`median()`	Median
`sd()`	Stichproben-Standardabweichung
`var()`	Stichproben-Varianz
`quantile()`	Quantile
`IQR()`	Interquartilsabstand
`mad()`	Schätzer für den Median der absoluten Abweichung vom Median

```
v <- c(1,2,3,4,5)
mean(v); median(v)

## [1] 3
## [1] 3

sd(v); var(v);

## [1] 1.58
## [1] 2.5

quantile(v)

##   0%  25%  50%  75% 100%
##    1    2    3    4    5

IQR(v); mad(v)

## [1] 2
## [1] 1.48
```

Pipe-Operator |>

- Der Pipe-Operator |> ermöglicht es, das erste Argument in intuitiver Weise zun übergeben.
- Was links vom Operator |> steht, wird als erstes Argument in der Funktion rechts verwendet.
- Formal entspricht `x |> f()` also dem Aufruf `f(x)`.
- Alle weiteren Argumente können der Funktion auf die übliche Weise hinzugefügt werden (siehe Beispiel `round()`).
- Das Ergebnis kann dann wiederum mit der Pipe |> in eine nächste Funktion geschoben werden, etc.
- Shortcut für Pipe-Operator in R-Studio: Strg + Shift + M (Windows) bzw. Shift + CMD + M (Mac)

```
  sqrt(9); 9 |> sqrt() # ist beides das gleiche

## [1] 3
## [1] 3

  log(2.25)

## [1] 0.811

  round(log(2.25), digits = 2)

## [1] 0.81

# letzteres kann geschrieben werden als:
  2.25 |> log() |> round( digits = 2 )

## [1] 0.81

  betraege <- c(10.89, 3,99, 4.98)
  betraege |> sum()

## [1] 118
```

Fehlende Werte: NAs

- Anders als z.B. bei Excel-Tabellen, in denen Zellen leer sein können, gibt es in de R-Datentabellen keine Leereinträge.
- Statt dessen gibt es den Eintrag `NA` (Not Available).
- Mit der Funktion `is.na()` kann nach `NA`-Einträgen gesucht werden. Die Funktion liefert `TRUE` falls ein Eintrag `NA` ist und `FALSE` falls dies nicht der Fall ist.
- Wenn bei einer Operation (`+,-,*,/,^,<,<=,==,>=,>`) ein Partner den Wert NA hat, ist das Ergebnis auch NA.
- Bei vielen Vektorfunktionen (wie z.B. `sum` oder `mean`) kann über die Option `na.rm` eingestellt werden, ob NAs vor der Ausführung gelöscht werden sollen. Wenn nicht gelöscht wird, ist das Ergebnis NA, wennn mindestens ein Verktoreingtrag `NA` ist.

```
NA+1; NA >= -1; NA == NA
```

```
## [1] NA
## [1] NA
## [1] NA
```

```
(x <- c(2, NA, 5, NA))
```

```
## [1]  2 NA  5 NA
```

```
is.na(x)
```

```
## [1] FALSE  TRUE FALSE  TRUE
```

```
# Summation über einen Vektor mit NAs:
sum(x)
```

```
## [1] NA
```

```
sum(x, na.rm = TRUE)  # ignoriert NAs
```

```
## [1] 7
```

Indizierung von Vektoren

- Einzelne Elemente eines Vektors können mittels eines zugehörigen Index angesprochen werden, indem dem Namen des Vektors der entsprechende Index in eckigen Klammern [] nachgestellt wird. Dabei beginnt die Zählung, anderes als bei den Arrays in gängigen Programmiersprachen wie C++ und Java bei 1 und nicht bei 0.
- Der Index muss keine einzelne natürliche Zahl sein, er kann aus einem Vektor von Indizes bestehen.
- Mittels eines vorangestellten Minuszeichens kann der Ausschluss eines oder mehrerer Elemente erreicht werden (sog. inverse Indizierung).
- Eine logische Indizierung ist möglich, wobei TRUE für ausgewählte und FALSE für abgewählte Elemente verwendet wird (Selektion durch Index). Hat ein Element den Wert NA, so wird es mit dem Wert NA ausgewählt, was beim Zählen zu Verwirrungen führen kann.

```
x <- c(12, 5, 1, 9, 11)
x[4]

## [1] 9
```

```
x[2:5]

## [1]  5  1  9 11

x[c(-1, -5)]

## [1] 5 1 9

x[x>7]

## [1] 12  9 11

alter <- c(12,34,19,NA,17)
(ü18 <- alter[alter>18])

## [1] 34 19 NA

length(ü18) # liefert NICHT die Anz. d. über 18-jähr.

## [1] 3

sum(alter>18, na.rm = TRUE) # Anz. d. über 18-jähr.

## [1] 2
```

Indizierung von Vektoren (Teil 2)

- Steht die Indizierung auf der linken Seite einer Zuweisung, so können damit Elemente des Vektors ersetzt werden; insbesondere bewirkt [] (also die leere eckige Klammer) ein Ersetzen aller Elemente eines Vektors.
- Damit können bequem auch alle fehlende Werte ersetzt werden.

```
x <- c(12, 5, 1, 9, 11, 5)
x[1] <- 33; x

## [1] 33  5  1  9 11  5

x[x==5] <- 6; x

## [1] 33  6  1  9 11  6

x[] <- 22
x <- c(12, 5, NA, 9, 11, NA)
# Ersetze alle NAs durch 0
x[is.na(x)] <- 0; x

## [1] 12  5  0  9 11  0
```

Wann sollte man eine Funktion erstellen?

```r
# a bis d enthalten jeweils 100 mal 1 bis 10 Punkte
set.seed(1907) # für Reproduzierbarkeit
a = sample( 0:10, 100, replace = TRUE )
b = sample( 0:10, 100, replace = TRUE )
c = sample( 0:10, 100, replace = TRUE )
d = sample( 0:10, 100, replace = TRUE )
```

```r
# nun sollen die Punkte so skaliert werden, dass sie
# zwischen 0 und 100% liegen, wobei die schlechteste
# Leistung 0% und die beste Leistung 100% sein soll
a <- (a - min(a, na.rm = TRUE)) /
  (max(a, na.rm = TRUE) - min(a, na.rm = TRUE))

b <- (b - min(b, na.rm = TRUE)) /
  (max(b, na.rm = TRUE) - min(a, na.rm = TRUE))

c <- (c - min(c, na.rm = TRUE)) /
  (max(c, na.rm = TRUE) - min(c, na.rm = TRUE))

d <- (d - min(d, na.rm = TRUE)) /
  (max(d, na.rm = TRUE) - min(d, na.rm = TRUE))
```

- Die Funktion skaliert die Punkte so, dass sie immer zwischen 0 und 100% liegen.
- Beim Kopieren & Einfügen ist aber ein Fehler passiert (Wo?).
- Regel: Wenn man mehr als zweimal dasselbe ausführt (meist per Copy&Paste), sollte man eine Funktion schreiben.
- Dies hat den weiteren Vorteil, dass Änderungen an der Funktion nur einmal ausgeführt werden müssen.
- Es ist einfacher, mit einer funktionierenden Befehlsfolge zu starten und diese in eine Funktion umzuwandeln, als eine Funktion auf einem leeren Blatt zu entwerfen und diese dann zum Laufen zu bringen.

Bestandteile einer Funktion

```
skaliere_Werte_01 <- function( pkte )
{
  punkte_skaliert <-
    (pkte - min(pkte, na.rm = TRUE)) /
     (max(pkte, na.rm = TRUE) - min(pkte, na.rm = TRUE))

  return(punkte_skaliert)
}
a_skaliert <- format( round(skaliere_Werte_01(a[1:10]),2),
                      digits=2 )
writeLines( paste( a[1], "(", a_skaliert[1], "); ",
                   a[2], "(", a_skaliert[2], "); ",
                   a[3], "(", a_skaliert[3], "); ",
                   a[4], "(", a_skaliert[4], "); ",
                   a[5], "(", a_skaliert[5], "); \n",
                   a[6], "(", a_skaliert[6], "); ",
                   a[7], "(", a_skaliert[7], "); ",
                   a[8], "(", a_skaliert[8], "); ",
                   a[9], "(", a_skaliert[9], "); ",
                   a[10], "(", a_skaliert[10], ")", sep="" ) )

## 4(0.44); 8(0.89); 7(0.78); 6(0.67); 6(0.67);
## 0(0.00); 0(0.00); 9(1.00); 6(0.67); 5(0.56)
```

- Eine Funktion besteht aus einem Funktionsnamen (im Bsp.: `skaliere_Werte_01`), den Argumenten (im Bsp.: `pkte`) und dem Funktionskörper (im Bsp.: `{ … }`).
- Ein guter Funktionsname beginnt (meist) mit einem Verb und erklärt, was die Funktion tut. Als Trenner zwischen den Worten im Funktionsnamen wird üblicherweise ein Unterstrich (wie im Bsp.), ein Punkt oder gar kein Zeichen („camelCase“) verwendet. Welche Variante man wählt, ist egal, aber man sollte sich für eine Variante entscheiden.
- Mit dem Befehl `return` wird festgelegt, welchen Wert die Funktion zurückgibt.

Bestandteile einer Funktion (2)

```
werfe_muenze <- function( Anzahl, Wsk_Kopf = 0.5 )
{
  return( sample( c("Kopf","Zahl"), Anzahl, replace = TRUE,
                  prob = c(Wsk_Kopf, 1-Wsk_Kopf )))
}
werfe_muenze( Wsk_Kopf = 0.7, Anzahl = 5 ) # unfaire Münze!

## [1] "Kopf" "Kopf" "Kopf" "Kopf" "Kopf"

werfe_muenze( 5, 0.7 ) # ohne Namen Reihenfolge einhalten

## [1] "Zahl" "Kopf" "Kopf" "Zahl" "Zahl"

werfe_muenze( 5 )      # fairer Münzwurf, da Wsk_Kopf std.mäßig 0.5

## [1] "Zahl" "Zahl" "Zahl" "Zahl" "Zahl"
```

- Wenn die Namen der Argumente mit angegeben werden, ist die Reihenfolge der Argumente egal.
- Ansonsten muss die Reihenfolge eingehalten werden.
- Wenn bei der Definition einer Funktion nach dem Argument ein Gleichheitszeichen und ein Standardwert angegeben wird, muss dieses Argument beim Aufruf nicht übergeben werden. Es wird dann der Standardwert verwendet.

Rückgabewert

Eine Rückgabe von mehr als einem Wert ist dadurch möglich, dass man vor der eigentlichen Rückgabe die zurückzugebenden Werte in einem geeigneten Objekt, meist einer Liste (dazu später mehr), zusammenfasst und eben dieses Objekt dann zurückliefert.

```
# Paket-Manager (muss install. sein)

library(pacman)
# Zahlentheorie-Bibliothek
p_load(numbers)
```

```
Quadrat_Reziprok_Primfaktoren <- function(x)
  { return (list( qu=x^2, rezi=1/x, prim_fct=primeFactors(x) )) }
(erg <- Quadrat_Reziprok_Primfaktoren(6))
```

```
## $qu
## [1] 36
##
## $rezi
## [1] 0.167
##
## $prim_fct
## [1] 2 3
```

```
str(erg); erg$prim_fct  # Zugriff auf Listenelement
```

```
## List of 3
##  $ qu      : num 36
##  $ rezi    : num 0.167
##  $ prim_fct: num [1:2] 2 3
## [1] 2 3
```

```
unlist(erg) # macht aus der Liste einen Vektor
```

```
##        qu      rezi prim_fct1 prim_fct2
##    36.000     0.167     2.000     3.000
```

Definition und Indizierung

- Wie bei den Vektoren müssen alle Elemente einer Matrix denselben Datentyp aufweisen.
- Mittels matrix() können Matrizen erzeugt werden. Argumente der Funktion matrix() sind:
  - die eigentlichen Daten, aus welchen sich die Matrix zusammensetzt, also die nach Spalten oder Zeilen geordneten (sortierten) Elemente der zu erzeugenden Matrix in Form eines Vektors,
  - eines oder beide der folgenden Argumente: `nrow` für die Zeilenanzahl sowie `ncol` für die Spaltenanzahl der zu erzeugenden Matrix und
  - `byrow`: `FALSE` entspricht einem spaltenweiser Aufbau der Matrix; `TRUE` führt zu einem zeilenweisen Aufbau (default ist `FALSE`).
- Die Indizierung von Matrizen erfolgt analog zur Indizierung von Vektoren, also per Index. Soll konkret das Element in der i.-te Zeile und j.-te Spalte einer Matrix X angesprochen werden, so ist hierfür X[i,j] zu schreiben.
- Wünscht man nur eine Ausgabe des i-ten Zeilenvektors, so ist hierfür die Spaltenangabe wegzulassen, also X[i,]; dementsprechend erhält man den j-ten Spaltenvektor mittels X[,j].
- Arrays sind eine Erweiterung der zweidimensionalen Matrizen auf beliebig große Dimensionen.

```
matrix(1:16, nrow=4, ncol=4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

```
(A <- matrix(1:16, nrow=4, byrow = TRUE))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
```

```
A[2,3]; A[3,]; A[,2]
```

```
## [1] 7
## [1]  9 10 11 12
## [1]  2  6 10 14
```

Matrizenfunktionen

Alle elementweisen Operationen und Funktionen werden auch bei Matrizen elementweise ausgeführt. Spezielle Matrixfunktionen sind:

Matrixfunktion	Bedeutung
`dim()`; `nrow()`; `col()`	Anzahl der Zeilen und Spalten; (nur) Anzahl der Zeilen; (nur) Anzahl der Spalten
`t()`	Transponieren
`%*%`	Matrixmultiplikation
`solve()`	Invertieren einer Matrix, Auflösen des Gleichungssystems $AX = B$ nach X
`eigen()`	Eigenwerte und Eigenvektoren
`apply()`	führt eine beliebige, auch benutzerdefinierte, Funktion je Zeile oder Spalte aus

```
A <- matrix(1:6, nrow=3, byrow = TRUE)
dim(A); nrow(A); ncol(A)

## [1] 3 2
## [1] 3
## [1] 2

t(A)

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
(B <- matrix(1:4, nrow=2))

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

(B_inv <- solve(B))

##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5

B %*% B_inv

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1

apply(B, 1, mean) # arith.Mittel je Zeile

## [1] 2 3

apply(B, 2, mean) # arith.Mittel je Spalte

## [1] 1.5 3.5
```

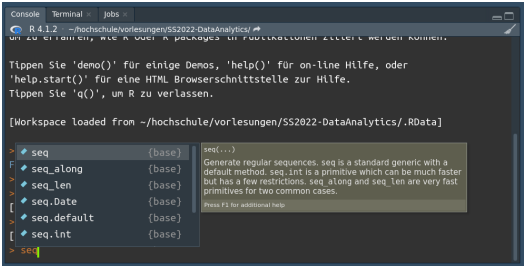
Definition

- Listen sind die flexibelste Datenstruktur in R und können als Elemente Objekte unterschiedlichen Datentyps und unterschiedlicher Länge enthalten.
- Häufig tauchen Listen als Rückgabewert von R-Funktionen auf. So kann z.B. auf Details von Signifikanztests oder auf deas Ergebnis einer linearen Regression zugegriffen werden. Dazu verwendet man den Auswahloperator $.
- Datentabellen (data frames) und tibbles sind Spezialfälle von Listen, bei denen die Elemente zwar einen unterschiedlichen Datenty, aber immer die gleiche Länge haben (rechteckige Datenstrukturen).
- Wie bei den benutzerdefinierten Funktionen gesehen, treten Listen auch als Rückgabewert von eigenen Funktioen auf, wenn man komplexere Daten zurückgeben möchte.

```
x <- rnorm(n=100,mean=0,sd=5) # 100 normalvtlt. Zufallszahlen
# Ist der Erwartungswert signifikant von 1 verschieden?
(erg <- t.test( x, mu=1 ))

##
## ^^IOne Sample t-test
##
## data:  x
## t = -1, df = 99, p-value = 0.2
## alternative hypothesis: true mean is not equal to 1
## 95 percent confidence interval:
##  -0.824  1.374
## sample estimates:
## mean of x
##     0.275

erg$conf.int # num. Vektor der Länge 2

## [1] -0.824  1.374
## attr(,"conf.level")
## [1] 0.95

erg$alternative # chr. Vektor der Länge 1

## [1] "two.sided"
```

R Lifehacks

- Fängt man an, einen Befehl in die Konsole einzugeben, so erscheint ein Fenster mit einer Liste der dazu passenden Befehle.
- Wählt man einen der gelisteten Befehle an, so wird ein Tooltip mit Informationen zu der Funktion eingeblendet.



- Mit △ (Pfeil nach oben) können die zuletzt aufgerufenen Befehle angezeigt und bei Bedarf verändert und nochmal ausgeführt werden.
- Mit `Alt` + `−` kann der Zuordnungsoperator hübsch formatiert eingefügt werden (Mac: `Option` + `−`).
- Mit `Strg` + `Return` kann die aktuelle Zeile im Skripteditor ausgeführt werden (Mac: `Cmd` + `Return`).
- Mit `Alt` + `Shift` + `K` erhält man eine Liste aller Keyboard-Shortcuts (Mac: `Option` + `Shift` + `K`).

[ABC 3; Moodle 0; Ü 1]