R Einführung

In diesem Markdown-Dokument lernen wir die Grundlegenden Funktionen der Statistiksoftware R kennen. Hierbei werden wir die folgenden Kapitel behandeln:

1. Datentypen
2. Packages
3. Funktionen
4. Schleifen und Bedingungen
5. Zufallszahlen
6. Plotten
7. Data Frames

Am besten öffnest du ebenfalls ein R Skript und wandelst die aufgeführten Beispiele nach Lust und Laune ab.

Da es laut Wikipedia Tradition ist, starten wir auch diese Einführung mit einem "Hello World!" Programm.

```
print("Hello World!")
```

```
## [1] "Hello World!"
```

1. Datentypen

In R gibt es die üblichen Datentypen:

- Integer
- Double
- Complex
- Character (String)
- Logical (Boolean)
- Raw

Wobei jedoch Raw sehr selten benutzt wird und deshlab hier nicht wirklich behandelt wird.

Grundlegendes

Mit `typeOf()` oder `classOf()` kann der Datentyp eines Objektes ausgegeben werden. Per default sind alle "Zahlen" als double gespeichert, können jedoch mit einem "L" zu einem Integer umgewandelt werden. Strings werden mit `" "` oder `' '` erstellt.

```
typeof(42)
```

```
## [1] "double"
```

```
typeof(42L)
```

```
## [1] "integer"
```

```
class(42L)
```

```
## [1] "integer"
```

```
typeof(42.0)
```

```
## [1] "double"
```

```
typeof("Zweiundvierzig")
```

```
## [1] "character"
```

```
typeof('Zweiundvierzig')
```

```
## [1] "character"
```

```
typeof(TRUE)
```

```
## [1] "logical"
```

```
typeof(4+2i)
```

```
## [1] "complex"
```

```
charToRaw("Zwei und Vierzig")
```

```
##  [1] 5a 77 65 69 20 75 6e 64 20 56 69 65 72 7a 69 67
```

```
typeof(charToRaw("Zwei und Vierzig"))
```

```
## [1] "raw"
```

Mit `#` wird ein Kommentar markiert, d.h. der Interpreter ignoeriert was nach # kommt.

```
# wichtige Inforamtion zum Verstehen des Codes
```

Selbstverständlich können Werte auch in Variablen gespeichert werden. Dies geschieht mit `=` oder `<-`, wobei letzteres geläufiger ist.

Durch `x<-3` wird eine Variable x mit dem Wert 3 erstellt. Durch einfassen in Klammern, `(x<-3)` wird die Variable direkt ausgegeben. Dies ist äquivalent zu `x` in einem Codeblock. Mit `print(x)` werden ebenfalls Konsolenausgaben erzeugt.

```
x<-3        # keine Ausgabe
(x<-3)      # Ausgabe
```

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

```
x           # Ausgabe
```

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

```
print(x)    # Ausgabe
```

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

Für numerische Datentypen funktioneren die Grundrechenarten wie zu erwarten. Der Modulo-Operator wird in R mit `%%` aufgerufen und mit `%/%` wird ganzzahlig geteilt.

```
3+3
```

```
## [1] 6
```

```
16-4
```

```
## [1] 12
```

```
7*9.1
```

```
## [1] 63.7
```

```
4/2
```

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

```
5%%2
```

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

```
3%/%2
```

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

```
3%/%4
```

```
## [1] 0
```

Logicals nehmen die Werte `True` und `FALSE`an und können mit `T` bzw. `F` abgekürzt werden. Wie üblich werden sie mit `x & y`, `x | y`, `! x` sowie `xor(x,y)` verarbeitet. Mit `isTRUE()` bzw. `isFALSE` lässt sich der Zustand des Boolens abfragen.

Hier ist jedoch die Besonderheit zu beachten, dass bei der Verwendung von Vektoren (nächster Abschnitt) im Falle von `&` und `|` wird der Operator elementweise interpretiert wird.

```
x<-TRUE
y<-FALSE
x&y
```

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

```
x|y
```

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

```
!x
```

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

```
xor(x,y)
```

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

```
isTRUE(x)
```

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

```
isFALSE(x)
```

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

```
c(T,T,T) & c(F,T,T)
```

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

```
c(T,T,T) & c(T,T,T)
```

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

Vergleichsoperatoren vergleichen zwei Objekte und liefern einen Boolen zurück. Für Skalare Zahlen sind in R die folgenden Operatoren definiert.

```
1==1      #gleich
```

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

```
1!=1      #ungleich
```

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

```
1<2       #kleiner als
```

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

```
1>2       #größer als
```

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

```
1<=2      #kleiner oder gleich
```

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

```
1>=2      #größer oder gleich
```

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

Vektoren und Matrizen

Vektoren

Wie gerade schon angedeuted ist es selbstverständlich auch möglich Werte in Arrays zu speichern. Arrays sind ein, zwei oder n- dimensionale Objekte in denen sich mehrere Werte speichern lassen. Aufgrund des mathematischen Schwerpunkts benutzen wir hier in dieser Einfühung die Worte Array und Vektor, bzw. 2D Array und Matrix synonym. Mit der Funktion `c( )` können wir einen Vektor erstellen

```
x<-c(1,2,3)
c(14,23,4)->z     #"Pfeilschreibweise" funktioniert auch in die andere Richtung
c(x,0,0,0,0,z)
```

```
##  [1]  1  2  3  0  0  0  0 14 23  4
```

Die oben beschriebenen Grundrechenarten können auch auch Vektoren angewendet werden und werden dabei elementweise verstanden.

```
x+3
```

```
## [1] 4 5 6
```

```
x+x
```

```
## [1] 2 4 6
```

```
x+c(1,2) #c(1,2) ist kürzer als x, deshalb wird mit dem ersten Element wieder begonnen
```

```
## Warning in x + c(1, 2): Länge des längeren Objektes
##       ist kein Vielfaches der Länge des kürzeren Objektes
```

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

Die grundlegenden Funktionen wie `log(),exp(),min(),sin(),..` sind in der R Basisversion bereits implementiert und lassen sich einfach auf Vektoren anwenden. Besonders nützlich ist die Funktion `length()`, welche die Länge des Vektors ausgibt. Inforamtionen zu Funktionnen können mit `help(sqrt), ?sqrt, ??sqrt` oder allgemein `help.start()` ausgegeben werden.

```
sin(x)
```

```
## [1] 0.8414710 0.9092974 0.1411200
```

```
log(x)
```

```
## [1] 0.0000000 0.6931472 1.0986123
```

```
max(x)
```

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

```
length(x)
```

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

```
help(sqrt)
```

```
## starte den http Server für die Hilfe fertig
```

Vektoren mit Zahlenfolgen können so erzeugt werden

```
1:10        #ganzzahlig von 1 bis 10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(from=0,to=5.7,by=0.3)    #von 0 bis 5,7 in Schritten der Größe 0,3
```

```
##  [1] 0.0 0.3 0.6 0.9 1.2 1.5 1.8 2.1 2.4 2.7 3.0 3.3 3.6 3.9 4.2 4.5 4.8 5.1 5.4
## [20] 5.7
```

```
seq(0,0.5,0.1)               #analog ohne "from","to" und "by
```

```
## [1] 0.0 0.1 0.2 0.3 0.4 0.5
```

```
rep(0,10)                    # 0 wird 10 mal wiederholt
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
rep(c(1,2),3)
```

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

```
rep(c(1,2,3),each=3)
```

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

```
rep(c(1,2,3), times=c(2,1,3))
```

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

Es gibt verschiedene Arten auf die Einträge von Vektoren zuzugreifen, z.B. über den Index, bzw. über Vektoren mit Indizes (R indiziert ab 1, d.h. auf das erste Element eines Vektors wird mit 1 zugegriffen)

```
x<-1:10
x[1]
```

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

```
x[c(1,3)]
```

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

Mit negativen Indizes lassen sich Werte aus Vektoren entfernen

```
x<-1:10
x[-1]
```

```
## [1]  2  3  4  5  6  7  8  9 10
```

```
x[-c(1,3,5)]
```

```
## [1]  2  4  6  7  8  9 10
```

Auch mithilfe von logischen Vektoren lässt sich auf Elemente eines Vektors zugreifen. Dies vereinfacht den Zugriff auf Einträge mit gewissen Eigenschaften, z.B. alle durch zwei teilbaren Einträge. Hierbei wird nur auf Elemente zugegriffen bei denen der Indexvektor TRUE ist.

```
x<-1:10
b<-rep(c(T,T,F,F,T),2)
x[b]
```

```
## [1]  1  2  5  6  7 10
```

```
durchZweiTeilbar<-c(x%%2==0)
x[durchZweiTeilbar]=42
x
```

```
##  [1]  1 42  3 42  5 42  7 42  9 42
```

Matrizen

Wie bereits erwähnt sind Matrizen zweidimensionale Arrays.

```
(M<-matrix(1:9,nrow=3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
matrix(10:18,nrow=3,byrow=TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]   10   11   12
## [2,]   13   14   15
## [3,]   16   17   18
```

Und der Zugriff erfolgt entweder elementweise, oder spalten/zeilenweise

```
M[1,2]
```

```
## [1] 4
```

```
M[1,]
```

```
## [1] 1 4 7
```

```
M[,1]
```

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

```
M[,c(1,3)]
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
```

```
M[c(1,2),c(1,3)]
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
```

Auch für Matrizen gibt es unzählige Rechenoperationen. Hier eine kleine Auswahl

```r
t(M) # Transponieren
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```r
M%*%M #Matrixmultiplikation
```

```
##      [,1] [,2] [,3]
## [1,]   30   66  102
## [2,]   36   81  126
## [3,]   42   96  150
```

```r
A<-matrix(c(1,2,1,2,5,-1,4,8,5),nrow=3)
b<-c(1,1,1)

solve(A) #Invertieren
```

```
##      [,1] [,2] [,3]
## [1,]   33  -14   -4
## [2,]   -2    1    0
## [3,]   -7    3    1
```

```r
solve(A,b) #Löse: Ax=b
```

```
## [1] 15 -1 -3
```

```r
eigen(A) #Eigenwertzerlegung
```

```
## eigen() decomposition
## $values
## [1] 5.485584+2.142906i 5.485584-2.142906i 0.028832+0.000000i
##
## $vectors
##                         [,1]                   [,2]           [,3]
## [1,]  0.3768163+0.0276553i  0.3768163-0.0276553i -0.97639359+0i
## [2,]  0.8952211+0.0000000i  0.8952211+0.0000000i  0.05797428+0i
## [3,] -0.0398659+0.2328830i -0.0398659-0.2328830i  0.20807341+0i
```

2. Packages

Bis jetzt haben wir nur mit der Basisfunktion von R gearbeitet, jedoch gibt es unzählige Packages, die Funktionen und Datensätze enthalten. Diese müssen einmalig mit dem Befehl `install.packages()` installiert werden und mit `library()` im Skript geladen werden. Im Laufe dieser Einführung werden wir noch einige nützlichen Packages kennenlernen.

```r
#install.packages("data.table")
library(data.table)
```

3. Funktionen

Durch Funktionen lassen sich gewisse Operationen leichter an verscheidenen Stellen ausführen und der Code wird dadurch überichtlicher. Definiert werden Funktionen in einer Funktionsumgebeung und können dann an einer beliebeigen Stelle im Skript aufgerufen werden.

Innerhalb einer Funktion können Variablen definiert werden, diese überschreiben keine gleichnamigen Variablen außerhalb der Funktion, außer dies wird mit einem `<<-` erzwungen.

Falls es lokal, innerhalb einer Funktion eine Variable nicht gibt, wird auf die globale Variable mit dem gleichen Namen zugegriffen.

Mit `return()` wird ein Wert zurück gegeben. Ohne das Stichwort return wird die letzte Zeile inerhalb der Funktion zurückgegeben.

```
addieren <- function(a,b,c){
    a=a+b+c                    #ohne globales a zu überschreiben
    return (a) # mit return
}

addieren2 <- function(a,b,c){
    a+b+c #ohne return
}

ueberschreiben <- function(b){
    a<<-3 # überschreiben
    return ("Überschrieben")
}

globalerZugriff <- function (x){
    return( x+a ) # ohne, dass a übergeben oder in der Funktion definiert wurde
}

a<-1
addieren(a,2,3)
```

```
## [1] 6
```

```
addieren2(1,2,3)
```

```
## [1] 6
```

```
a
```

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

```
ueberschreiben(a)
```

```
## [1] "Überschrieben"
```

```
a
```

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

```
globalerZugriff (4)
```

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

Bei der Übergabe können default Parameter festgelegt werden, die benutzt werden, sofern nichts anderes angegeben wird.

```
sinus <- function (x,ampl=1,shift=0){
    return (ampl*(sin(x-shift)))
    }

sinus(3)
```

```
## [1] 0.14112
```

```
sinus(3,1,0)
```

```
## [1] 0.14112
```

Außerdem kann die Reihenfolge der Inputparamter vertauscht werden, wenn diese mit = übergeben werden

```
sinus2 <- function (x,ampl,shift) {
    return (ampl*(sin(x-shift)))
}

sinus2(3,2,1)
```

```
## [1] 1.818595
```

```
sinus2(3,shift=1,ampl=2)
```

```
## [1] 1.818595
```

Operatoren wie z.B. + sind auch Funktionen. Durch eine Verschachtelung des Funktionsnamens in "% %" lässt sich eine Funktion konstruieren, welcher eine Variable vor und eine nach dem Funktionsnamen übergeben wird.

```
`%addmult2%` <- function(x,y) {
  x+2*y
  }

1:4 %addmult2% 3:6
```

```
## [1]  7 10 13 16
```

4. Schleifen und Bedingungen

4.1 If - Bedingungen

Eine If Bedigung hat eine Boolean als Input und führt je nach Zustand eine Operation aus

Die einfachste If- Bedingung hat die Gestalt

```
if (Bedingung){

Anweisung

}
```

Dies lässt sich zu einem “Entweder, oder” erweitern

```
if (Bedingung){

Anweisung

}else{

Anweisung alternativ

}
```

Mit `else if` lassen sich mehrere Fälle abfragen, wobei immer der erste Fall der zutrifft eintritt

```
if (Bedingung){

Anweisung

}else if (Bedinung 2){
```

```
Anweisung alternativ

}else{

Anweisung alternativ 2

}
```

```
x<-0

if (x<10){
    print("kleine Zahl")
}
```

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

```
if (x>10){
    print("große Zahl")
}else{
    print("kleine Zahl")
}
```

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

```
if (x>0){
    print("positiv")
}else if (x<0){
    print("negativ")
}else{
    print("Null")
}
```

```
## [1] "Null"
```

Diese Art von If-Bedinung ist jedoch nur skalarwertig. Wollen wir Vektoren abfragen, können wir dies folgendermaßen machen

```
x<- -2:10


ifelse(x>0,"positiv","Null oder negativ")
```

```
##  [1] "Null oder negativ" "Null oder negativ" "Null oder negativ"
##  [4] "positiv"           "positiv"           "positiv"
##  [7] "positiv"           "positiv"           "positiv"
## [10] "positiv"           "positiv"           "positiv"
## [13] "positiv"
```

4.2 For Schleife

In einer for-Schleife läuft ein Zähler über jedes Element eines Vektors. In der Regel sind dies Integer von 1 bis n, da man so auf andere Vektoren und Matrizen zugreifen kann, doch ist es auch möglich über z.B. Doubles oder Logicals zu iterieren.

```
frucht<-c('Apfel','Banane','Kiwi','Birne','Mango')

for (zähler in 1:length(frucht)){
    print(frucht[zähler])
}
```

```
## [1] "Apfel"
## [1] "Banane"
## [1] "Kiwi"
## [1] "Birne"
## [1] "Mango"
```

```
for (count in seq(from=0.1,to=1,by=0.1))
    {
    print(count)
}
```

```
## [1] 0.1
## [1] 0.2
## [1] 0.3
## [1] 0.4
## [1] 0.5
## [1] 0.6
## [1] 0.7
## [1] 0.8
## [1] 0.9
## [1] 1
```

```
for (bool in c(T,T,T)){
    print("Iteration")
}
```

```
## [1] "Iteration"
## [1] "Iteration"
## [1] "Iteration"
```

4.3 While-Schleife

Prinzipiell kann jede for-Schleife durch eine while-Schleife ersetzt werden und umgekehrt. Jedoch gibt es durchaus Anwendungen bei dennen das eine sinnvoller ist als das andere.

Eine while-Schleife wiederholt so lange einen Prozess, bis eine Bedingung erfüllt ist.

```
n<-0
while(n<5){
    print(n)
    n<-n+1
}
```

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

```
c<-1
bool<-T
while(bool){
    print("Yap")
    if (c==5){
        bool<-F
        }
    c<-c+1
}
```

```
## [1] "Yap"
## [1] "Yap"
## [1] "Yap"
## [1] "Yap"
## [1] "Yap"
```

4.4 Repeat-Schleife

Sehr ähnlich wie eine while-Schleife, ist eine repeat-Schleife. Diese wird mit dem Stichwort `break` beendet.

```
n<-0

repeat{
    n<-n+1
    if(n==7) break
    if(n %% 2 == 0) print(n)
}
```

```
## [1] 2
## [1] 4
## [1] 6
```

4.5 Exkurs: Vektorisieren

Schleifen sind oft sehr ineffektiv und langsam, deshalb werden Operationen vektorisiert um die Performance des Codes zu erhöhen. Z.B. wendet die Funktion `apply(M,dim,fun)` die function `fun` zeilen- oder spaltenweise (dim=1,2) auf `M` an.

```
(M<-matrix(1:9,nrow=3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
apply(M,1,mean)
```

```
## [1] 4 5 6
```

```
apply(M,2,sum)
```

```
## [1]  6 15 24
```

Dieses Prinzip kann auch mit `lapply(x,fun)` auf Listen übertragen werden.

```
liste=list(Spaß=c("Fahrrad","Mathe","Party"),kleineZahlen=1:9,großeZahlen=12:18)
lapply(liste,length)
```

```
## $Spaß
## [1] 3
##
## $kleineZahlen
## [1] 9
##
## $großeZahlen
## [1] 7
```

Kompliziertere Funktionen lassen sich entweder vorab definieren, oder mit sog. anonymen Funktionen auf Listen/ Matrizen/ ... anwenden.

Anonyme Funktionen wenden die Funktion auf jedes Element `x` der List/ Matrix/ `...` an. Und werden entweder mit `function(x) ...` oder `\ (x) ...` initialisiert.

Mit der Funktion `sapply(I,fun)` kann auf auf jedes Element von I die Funktion angewandt werden.

Der Unterscheid zwischen den Funktionen ist, dass `apply(I,dim,fun)` die Funktion spalten- oder zeilenweise anwendet, `lapply(I,fun)` elementweise anwendet, aber Listen zurückgibt und `sapply(I,fun)` ebenfalls elementweise anwendet, aber Vetoren/ Matrizen/ Listen zurückgibt.

```
SehrNuetzlicheFunktion<-function(x){
    if (any(x>5)){
        return(x)
    }else{
        return("klein")
    }
}

print("apply()")
```

```
## [1] "apply()"
```

```
apply(M,2,SehrNuetzlicheFunktion) # Vordefinierte Funktion
```

```
## [[1]]
## [1] "klein"
##
## [[2]]
## [1] 4 5 6
##
## [[3]]
## [1] 7 8 9
```

```
print("sapply()")
```

```
## [1] "sapply()"
```

```
sapply(M,SehrNuetzlicheFunktion)
```

```
## [1] "klein" "klein" "klein" "klein" "klein" "6"     "7"     "8"     "9"
```

```
print("lapply()")
```

```
## [1] "lapply()"
```

```
lapply(M,SehrNuetzlicheFunktion)
```

```
## [[1]]
## [1] "klein"
##
## [[2]]
## [1] "klein"
##
## [[3]]
## [1] "klein"
##
## [[4]]
## [1] "klein"
##
## [[5]]
```

```
## [1] "klein"
##
## [[6]]
## [1] 6
##
## [[7]]
## [1] 7
##
## [[8]]
## [1] 8
##
## [[9]]
## [1] 9
```

```
print("Anonyme Funktionen")
```

```
## [1] "Anonyme Funktionen"
```

```
lapply(liste,function(x) length(x)<4) # Anonyme Funktion
```

```
## $Spaß
## [1] TRUE
##
## $kleineZahlen
## [1] FALSE
##
## $großeZahlen
## [1] FALSE
```

```
apply(M,2,\(x){if (any(x>5)){x}else{"klein"}}) # Das selbe wie oben, nur anonym
```

```
## [[1]]
## [1] "klein"
##
## [[2]]
## [1] 4 5 6
##
## [[3]]
## [1] 7 8 9
```

5 Zufallszahlen

Um Plots wie Histogramme oder Boxplots zu erklären, führen wir noch kurz die Generierung von Zufallszahlen ein.

Einer der wichtigsten Befehle ist hierbei `runif(n)`, welcher einen Vektor mit n Zufallszahlen auf [0,1] erzeugt. Flexibler ist der Befehl `sample(x,n,replace,prob)`, welcher n Elemente aus x zieht. Mit dem Boolean replace kann Zurücklegen aktiviert, bzw. deaktiviert werden. Mit dem Vektor prob kann die Wahrscheinlichkeit, dass ein Element gezogen wird gewichtet werden.

Alle am PC generierten Zufallszahlen sind nicht echt zufällig, siehe Wikipedia. Sie werden mit einem Algorithmus berechnet und können nicht vom echten Zufall unterscheiden werden, jedoch können sie reproduziert werden, indem der selbe Startwert, der sog. Seed übergeben wird. Wird kein Seed übergeben wird bei R die Systemzeit als Seed benutzt.

```
# Bei jeder Ausführung verscheidene zufällige Ergebnisse
(uniform<-runif(1))
```

```
## [1] 0.9460303
```

```
(uniform2d<-runif(3))
```

```
## [1] 0.7475954 0.7114957 0.3679320
```

```
(integer<-sample(1:10,10,replace=T))                # Ganzzahlig zwischen 1:100, mit zurücklegen,
```

```
##  [1]  9  2  5 10  1  2  4  2 10  4
```

```
                                                    # ACHTUNG: defaul=F

(zufall<-sample(c("apfel","birne","melone"),1))     # Funktioniert auch für beliebige Mengen
```

```
## [1] "melone"
```

```
(normalverteilt<-rnorm(n=3,mean=1,sd=2))
```

```
## [1] 0.4827016 1.7808507 1.6315440
```

```
# Bei jeder Ausführung die selben Ergebnisse
set.seed(43)
sample(1:10,1)
```

```
## [1] 8
```

Es können auch Zufallszahlen nach bestimmten Verteilungen erzeugt werden

Diskret: - `rpois(), rbinom(), ...`

Stetig: - `rbeta(), rexp(), rnorm(), runif(),..`

Hierbei wird immer erst die Anzahl der Zufallsvariablen übergeben und dann optional die jeweiligen Parameter der Verteilung.

```
rpois(10,1000)
```

```
##  [1]  950 1003 1000 1035  968  998  978  961  969  968
```

```
rexp(5,3)
```

```
## [1] 0.142942236 0.446388951 0.008337294 1.116924182 0.113532445
```

6. Plotten

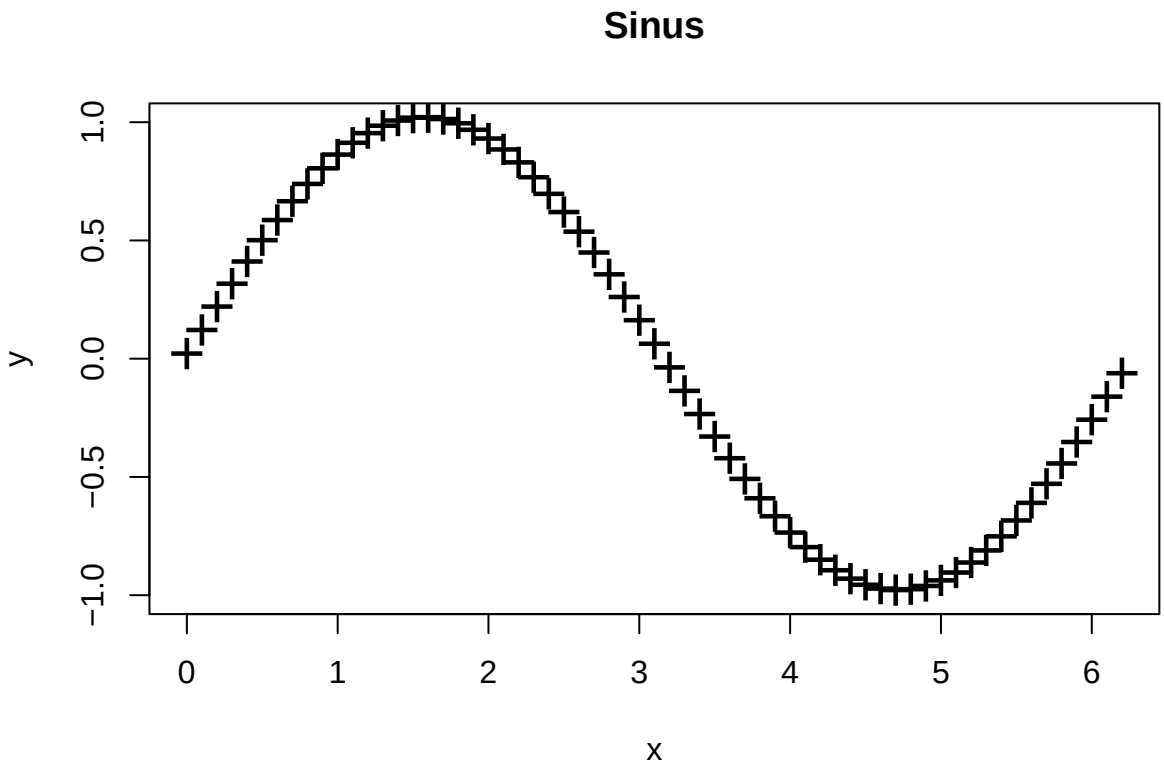
6.1 Lineplots, Scatterplots

In R gibt es nur einen Befehl zum plotten, nämlich plot(x,y,...) . Es werden sämtliche (optionale) Parameter direkt mit übergeben:

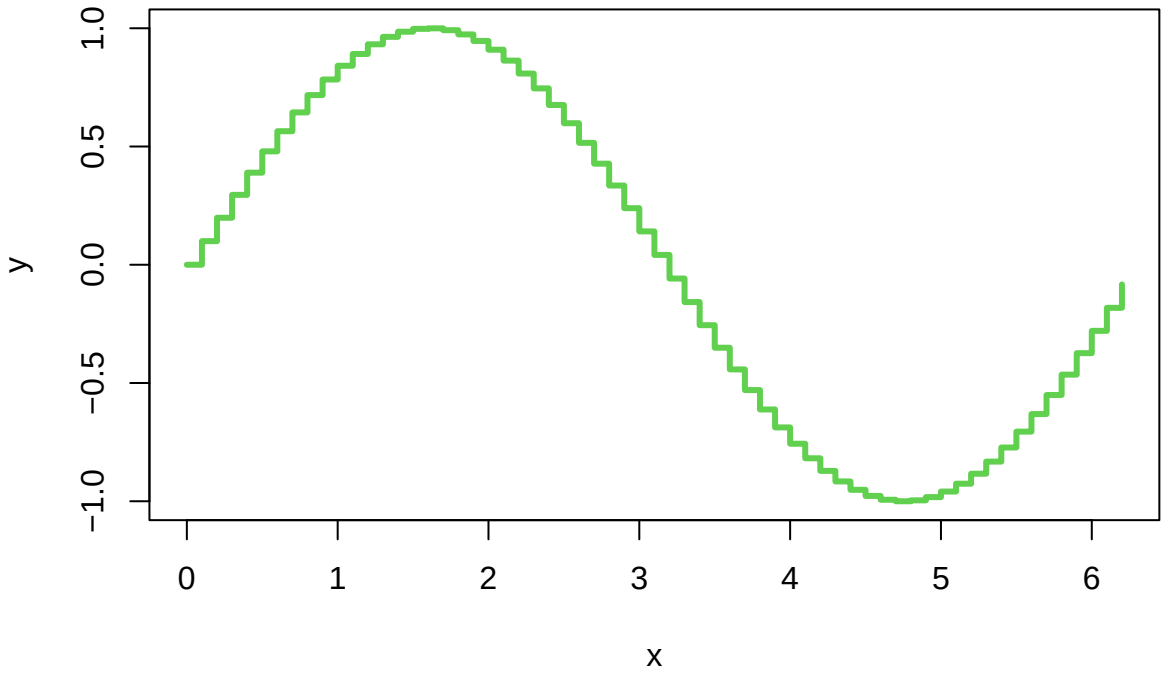
- type: (als String): 'n' - none, 'l' - line, 's' - stepfunction, 'p' - points (default)
- xlim, ylim (als Vektor): Achsenabschnitte z.B. c(0,4.2)
- main: (als String): Titel
- xlab, ylab (als String): Achsenbeschriftungen
- col: (als String) siehe. colours() oder (als Zahl) z.B: 2 = rot
- pch (Zahl oder String): Markerymbol
- cex (Zahl): Markergröße (character expansion)
- lty (Zahl): Linetype - 1 = durchgezogen, 2 = gestrichelt...
- lwd (Zahl): Lininedicke

```
x<-seq(from=0,to=2*pi,by=0.1)
y<-sin(x)
```

```
plot(x,y,main='Sinus',col=1,pch='+',cex=2)
```

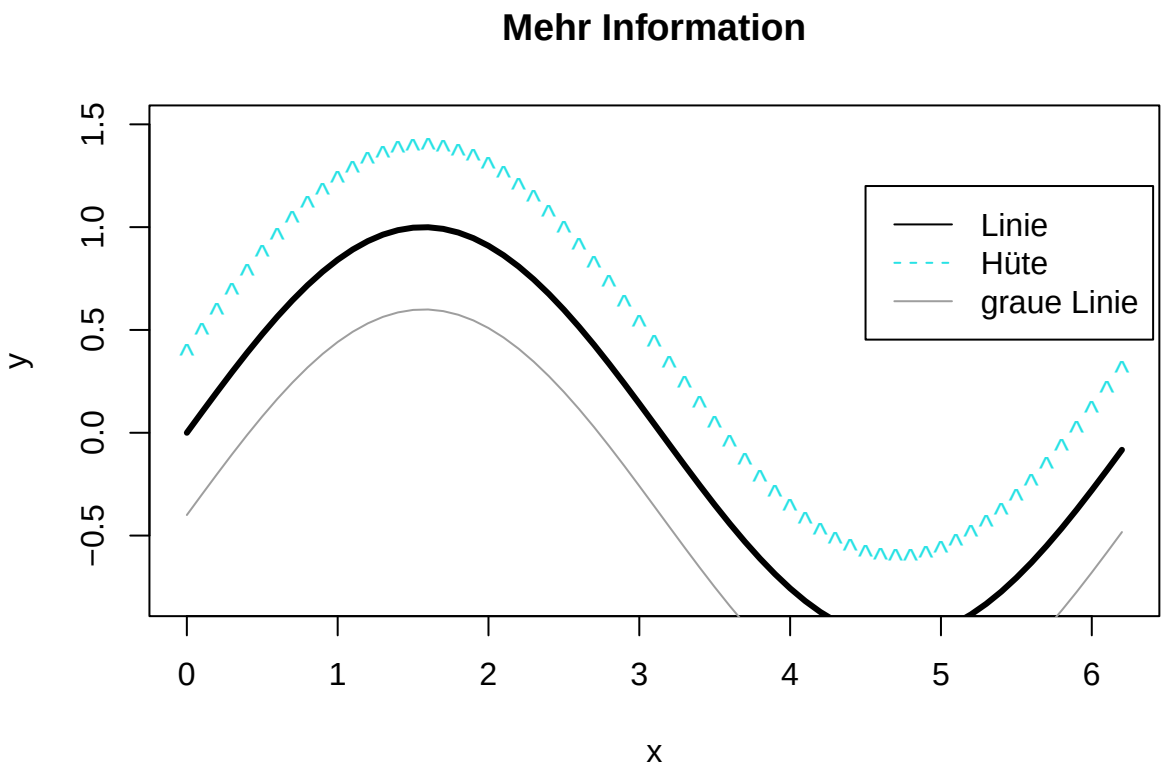



```
plot(x,y,type='s',col=3,lwd=3)
```



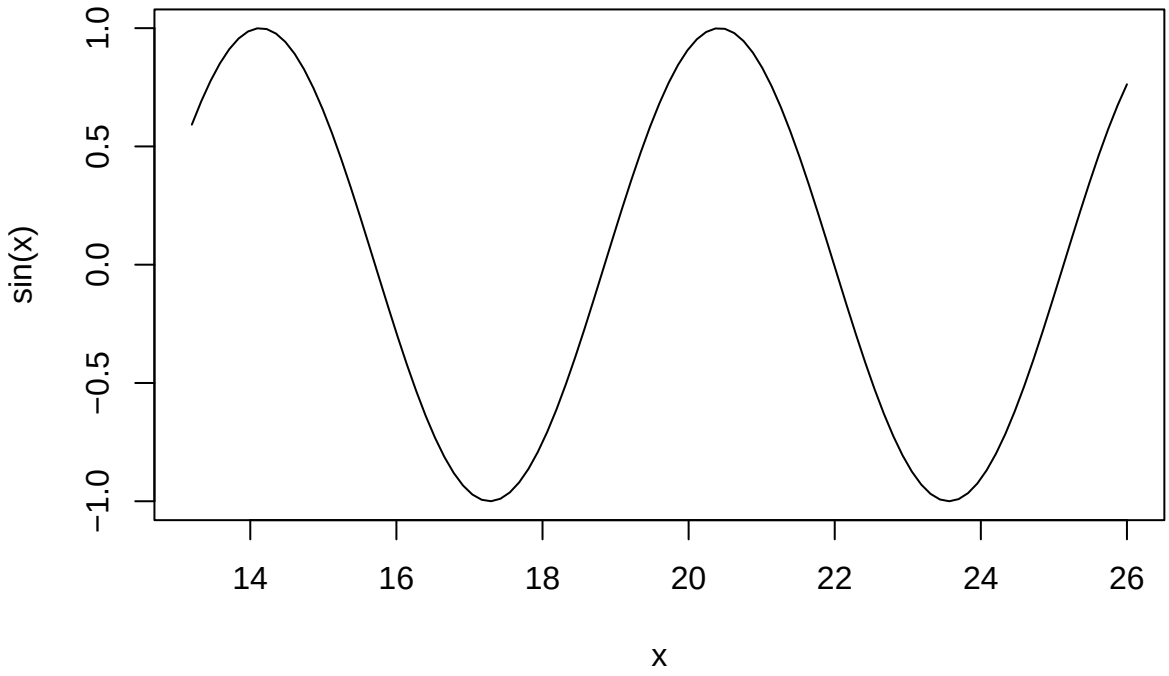
Mit lines() oder points() lassen sich mehrere Muster in das selbe Bild einfügen.

```
plot(x,y,main="Mehr Information",type='l',lwd=3,ylim=c(-0.8,1.5))
points(x,y+0.4,col=5,pch='^')
lines(x,y-0.4,col=8)
legend(4.5,1.2,c('Linie','Hüte','graue Linie'),col=c(1,5,8),lty=c(1,2,1))
```



Funktionen können auch direkte mit dem Befehl curve geplottet werden

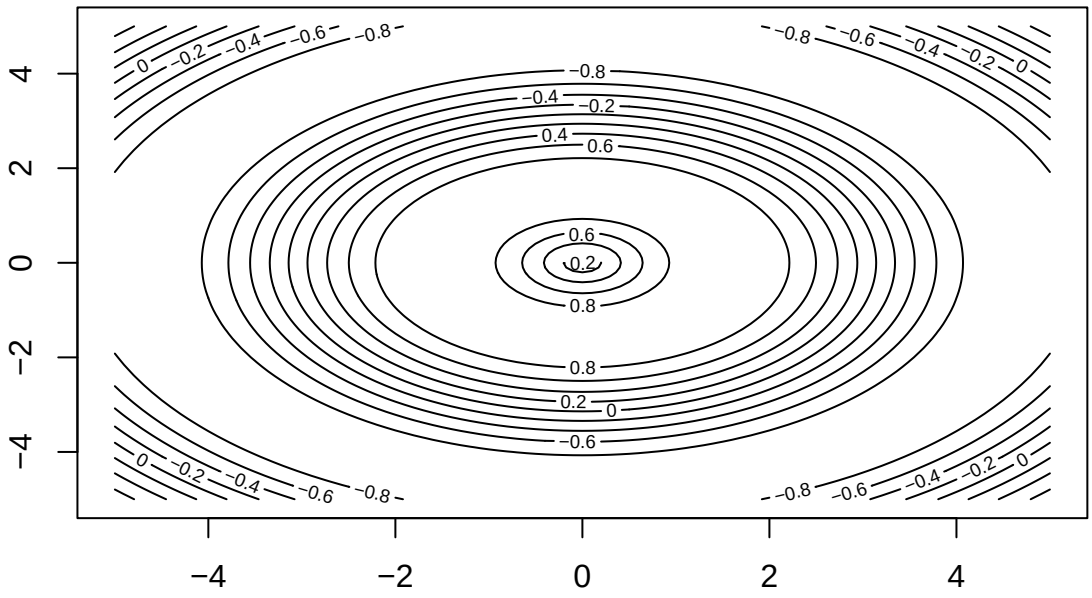
```
curve(sin,from=13.2,to=26)
```



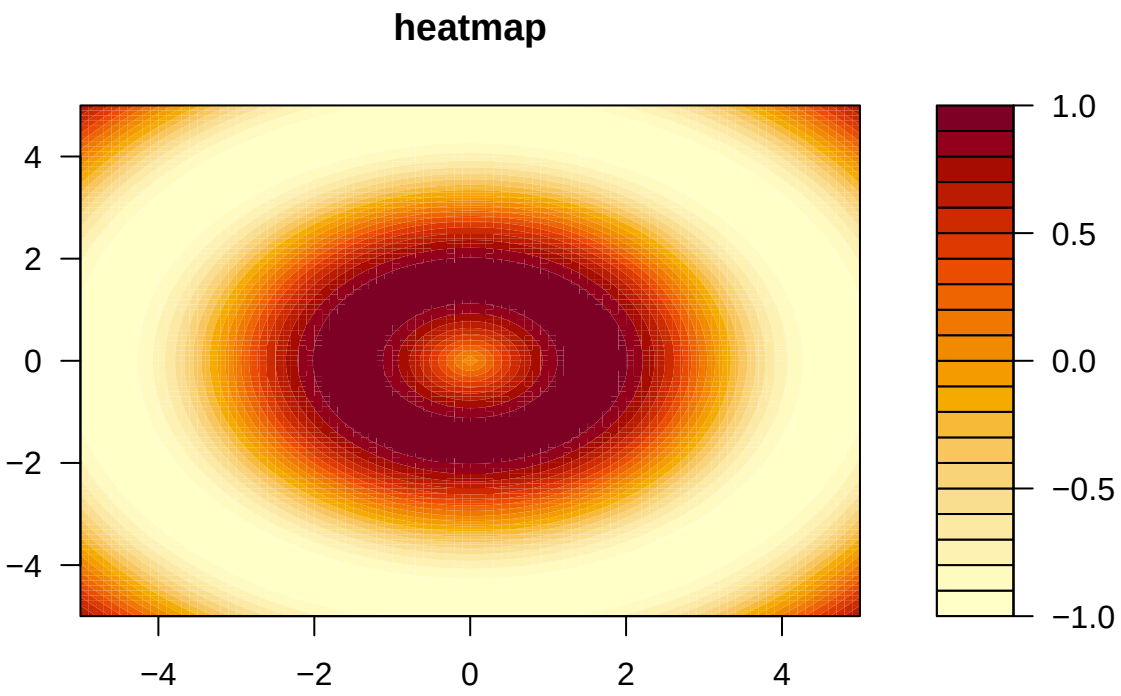
Höhenlinien lassen sich mit contour(x,y,z) erstellen

```
xs=seq(from=-5,to=5,by=0.1)
ys=seq(from=-5,to=5,by=0.1)
zs=sin(sqrt(outer(xs^2,ys^2,'+')))  # outer erstellt eine Matrix mit den passenden
                                    # Funktionswerten zu sqrt(x^2+y^2)

contour(xs,ys,zs,main='Höhenlinien')
```




```
filled.contour(xs,ys,zs,main='heatmap')
```

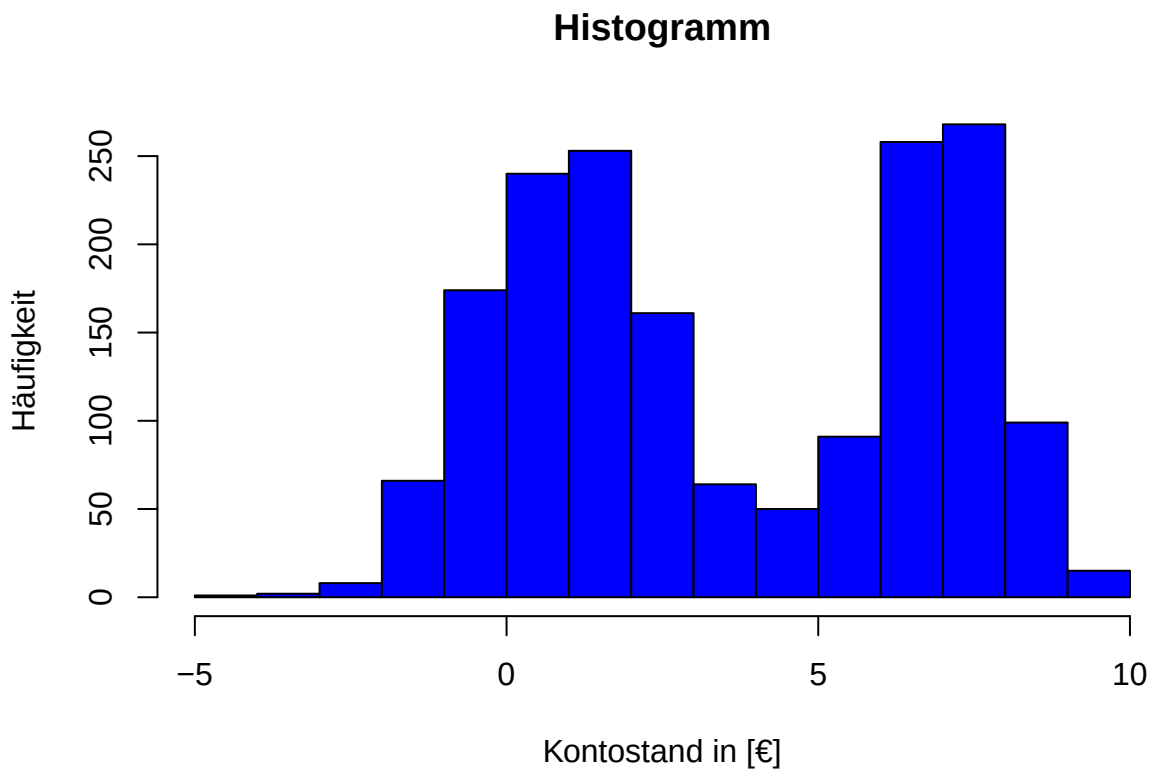


6.2 “Statistische” plots

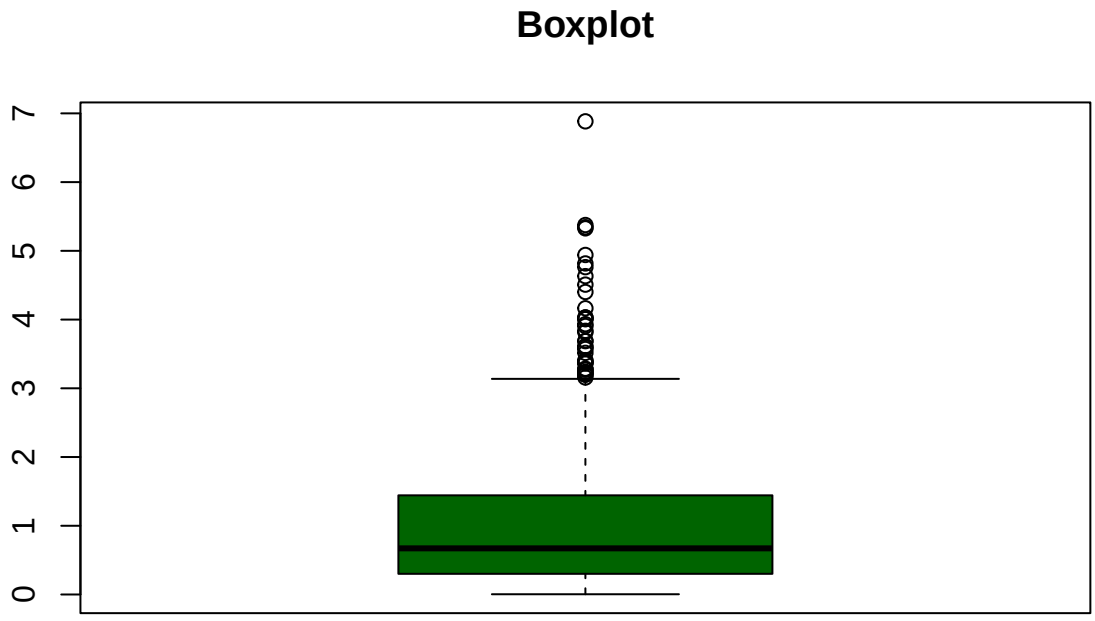
Ohne viel zu erklären, hier ein paar Möglichkeiten Datensätze zu visualisieren.

```
h<-c(rnorm(n=1000,mean=1,sd=1.5),rnorm(n=750,mean=7,sd=1))

hist(h,breaks=20,ylab='Häufigkeit',xlab='Kontostand in [€]',main='Histogramm',col='blue')
```

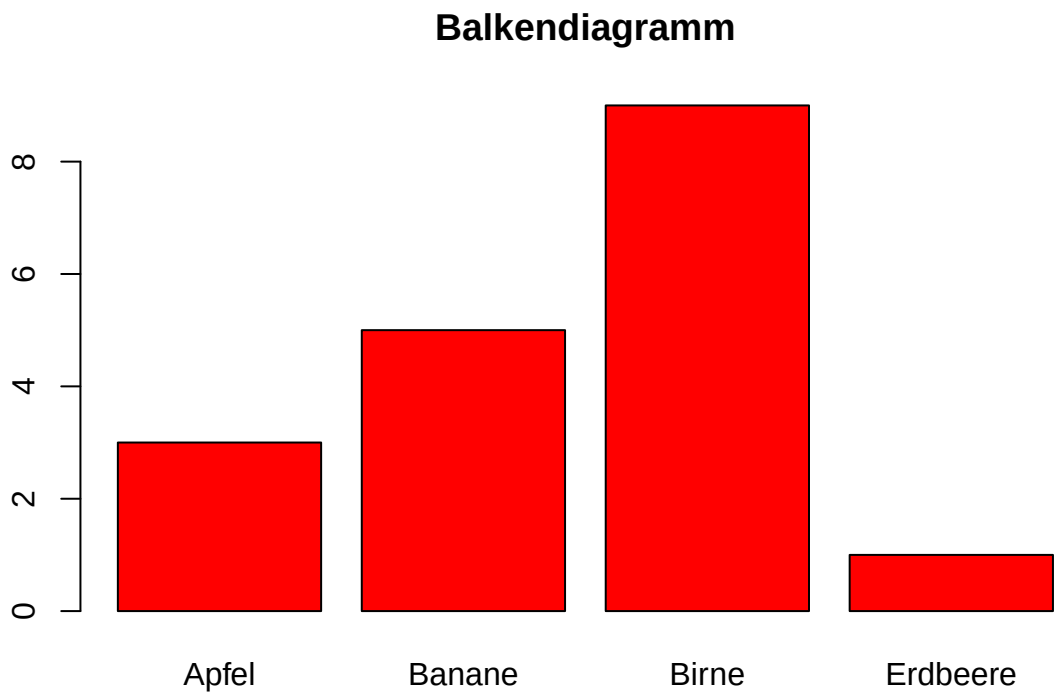



```
boxplot(rexp(1000),main='Boxplot',col='darkgreen')
```




```
i<-c(3,5,9,1)  # Zu Gruppe 1-4 gehören jeweils 3,5,9,1 Datenpunkte
label<-c('Apfel', 'Banane', 'Birne', 'Erdbeere') # Labels initialisieren

barplot(i,names.arg=label,col='red',main="Balkendiagramm")
```



Abschließend hier noch eine Möglichkeit einen erzeugten Plot zu speichern. Analog lassen sich Plots auch als andere Dateiformate speichern.

```
jpeg(file="Tortendiagramm.jpeg")                # Speichert als .jpeg ins aktuelle Arbeitsverzeichnis
                                                # Analoge Befehle existieren für .pdf, .png ,...
pie(i,labels=label,main='Mhhhhh....Torte')      # Bild erstellen
pdf(file="Tortendiagramm.pdf")                  # als .pdf
pie(i,labels=label,main='Mhhhhh....Torte')      # Bild erstellen
```

7. Dataframes

Ein Data Frame ist eine Datenstruktur, die als Tabelle interpretiert werden kann, er besteht also aus Zeilen und Spalten. Alle Zeilen sind gleich lang und sind vom gleichen Datentyp. Alle Spalten sind gleich lang, aber verschiedene Spalten können unterschiedliche Typen haben. (siehe unten, z.B. Double und String)

Der Datansatz 'ToothGrowth' ist ein Beispiel Frame von R und enthält die Ergebnisse eines Experiments bei dem der Einfluss von Vitamin C auf das Zahnwachstum von Meerschweinchen untersucht wurde. Die Zeilen bilden Datenpunkte von 60 unterssuchten Meerschweinchen, während die Spalten die betrachteten Merkmale beschreiben: Länge der Zähne, Verabreichungsart (VC - ascorbine acid und OJ - Orangensaft), sowie die tägliche Vitamin C Dosis.

Hier ein paar Befehle, mit dennen ihr einen Data Frame untersuchen könnt

```
(dim(ToothGrowth)) # Dimension
```

```
## [1] 60  3
```

```
(head(ToothGrowth,3)) # Erste drei Zeilen des Datensatzes
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
```

```
(tail(ToothGrowth,3)) # Letze drei Zeilen des Datensatzes
```

```
##     len supp dose
## 58 27.3   OJ    2
## 59 29.4   OJ    2
## 60 23.0   OJ    2
```

```
(str(ToothGrowth))    # Aufbau
```

```
## 'data.frame':    60 obs. of  3 variables:
##  $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
##  $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
##  $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

```
## NULL
```

```
# Analog zu Matrizen, Zugriff auf bestimmte Einträge
(ToothGrowth[5,])     # 5. Zeile (-> Beobachtung)
```

```
##   len supp dose
## 5 6.4   VC  0.5
```

```
(ToothGrowth[,2])     # 2. Spalete (-> Merkmal)
```

```
##  [1] VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC
## [26] VC VC VC VC VC OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ
## [51] OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ
## Levels: OJ VC
```

```
(ToothGrowth[2:4,c(1,3)])
```

```
##    len dose
## 2 11.5  0.5
## 3  7.3  0.5
## 4  5.8  0.5
```

```
# Zugriff über Name des Merkmals mit dem Operator $

(ToothGrowth$len)     # gibt alles Längen zurück
```

```
##  [1]  4.2 11.5  7.3  5.8  6.4 10.0 11.2 11.2  5.2  7.0 16.5 16.5 15.2 17.3 22.5
## [16] 17.3 13.6 14.5 18.8 15.5 23.6 18.5 33.9 25.5 26.4 32.5 26.7 21.5 23.3 29.5
## [31] 15.2 21.5 17.6  9.7 14.5 10.0  8.2  9.4 16.5  9.7 19.7 23.3 23.6 26.4 20.0
## [46] 25.2 25.8 21.2 14.5 27.3 25.5 26.4 22.4 24.5 24.8 30.9 26.4 27.3 29.4 23.0
```

```
(ToothGrowth$len[1:5])   # Die ersten 5 Längen
```

```
## [1]  4.2 11.5  7.3  5.8  6.4
```

```
(ToothGrowth$dose==0.5)     # Booleanvektor -> True an den Stellen dose==0.5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [37]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
(ToothGrowth[ToothGrowth$dose==0.5,]) # Gibt alle Zeilen mit dose==0.5
```

```
##     len supp dose
## 1   4.2   VC  0.5
## 2  11.5   VC  0.5
## 3   7.3   VC  0.5
## 4   5.8   VC  0.5
## 5   6.4   VC  0.5
## 6  10.0   VC  0.5
## 7  11.2   VC  0.5
## 8  11.2   VC  0.5
## 9   5.2   VC  0.5
## 10  7.0   VC  0.5
## 31 15.2   OJ  0.5
## 32 21.5   OJ  0.5
## 33 17.6   OJ  0.5
## 34  9.7   OJ  0.5
## 35 14.5   OJ  0.5
## 36 10.0   OJ  0.5
## 37  8.2   OJ  0.5
## 38  9.4   OJ  0.5
## 39 16.5   OJ  0.5
## 40  9.7   OJ  0.5
```

Selbstverständlich wollen wir jetzt noch wissen, ob das Zahnwachsum mit der Vitamin C Dosis korreliert

```
(cor(ToothGrowth$len,ToothGrowth$dose, method="pearson")) # Pearson Korrellationskoeffizient
```

```
## [1] 0.8026913
```

```
(cor(ToothGrowth$len[ToothGrowth$supp=="OJ"],ToothGrowth$dose[ToothGrowth$supp=="OJ"],
     method="pearson")) # Bedingt auf Orangensaft
```

```
## [1] 0.7500585
```

```
(cor(ToothGrowth$len[ToothGrowth$supp=="VC"],ToothGrowth$dose[ToothGrowth$supp=="VC"],
     method="pearson")) # Bedingt auf Vitamin C Präparat
```

```
## [1] 0.8989722
```