V01 – Crashkurs

12. April 2021

Contents

Ziele der Lektion

Dieser Crashkurs soll eine Einführung in den Umgang mit R darstellen und eine grobe Übersicht über die Fähigkeiten von R geben. Dabei werden die meiste Themen nur oberflächlich behandelt und erst in späteren Lektionen im Detail ausgeführt.

1 R im Terminal

R ist eine freie (*open source*) Programmiersprache für statistische Berechnungen und Grafiken.

Außerdem ist `R` das Programm, welches diese Programmiersprache ausführt. Die aktuelle Version von `R` kann unter https://www.r-project.org heruntergeladen werden.

Wir beginnen mit der Nutzung von R im Terminal und gehen später zu einer grafischen Benutzeroberfläche über.

Starten wir das Programm `R`, werden einige allgemeine Informationen anzeigt – unter anderem die Versionsnummer.

```
R version 3.6.3 (2020-02-29) -- "Holding the Windsock"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Für diese Lektion gehen wir von einer Version ≥ 3.6 aus.

Die Ausgabe endet mit der **Eingabeaufforderung** (*command prompt*) `>`. Dort können wir nun R-Befehle eingeben und ausführen (`Enter`-Taste).

2 Einfache Operationen

2.1 Rechnen

Wir können in der Eingabeaufforderung `>` verschiedene mathematische Terme eingeben, die `R` für uns auswertet.

Im Folgenden sind Eingaben in die Eingabeaufforderung zu sehen. Darunter steht mit `##` markiert die Ausgabe.

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

```
3*4
## [1] 12
5^6
## [1] 15625
(-3.5 + 7) * 2 + 0.5 * 16^0.5
## [1] 9
cos(pi)
## [1] -1
```

Hier sind einige mathematische Funktionen die zum Grundwortschatz von R gehören:

- Grundrechenarten: `+`, `-`, `*`, `/`,
- Potenz `^`, Wurzel `sqrt()`
- Division mit Rest: `%%` (Rest, "modulo"), `%/%` (ohne Rest)
- trigonometrische Funktionen `sin()`, `cos()`, `tan()`; Arkusfunktionen `asin()`, `acos()`, `atan()`; analog `sinh()`, `asinh()`, ... für Hyperbelfunktionen
- `exp()`, `log()` (Basis e), `log10()` (Basis 10), `log2()` (Basis 2)
- Betrag `abs()`, Vorzeichen `sign()`
- Aufrunden `ceiling()`, Abrunden `floor()`, Nachkommastellen abschneiden `trunc()`, kaufmännisches Runden `round()`

Außerdem ist die Konstante `pi` (π) implementiert.

Reelle Zahlen können mit der sogenannten **E-Notation** eingegeben werden: `7.6e-5` ($7{,}6 \cdot 10^{-5}$), `-1.23e4` ($-1{,}23 \cdot 10^4$). Je nach Größe des Wertes entscheidet sich `R` auch bei der Ausgabe für dieses Format.

```
1e2
## [1] 100
1e20
## [1] 1e+20
```

Da ein Computer nur eine endliche Teilmenge der reellen Zahlen darstellen kann (**Maschinengenauigkeit**), kann es zu Rechenfehlern kommen.

```
sin(pi)
## [1] 1.224606e-16
```

Reelle Zahlen haben den Typ `double`. Der Typ eines Objektes wird mit der Funktion `typeof()` abgefragt.

```
typeof(1)
## [1] "double"
typeof(pi)
## [1] "double"
```

Zum Typ `double` gehören auch `Inf` (unendlich), `-Inf` und `NaN` (*not a number*).

```
1/0
## [1] Inf
-1/0
## [1] -Inf
0/0
## [1] NaN
```

2.2 Logik

`R` wertet Vergleiche und aussagenlogische Formeln aus.

```
1 > 2
## [1] FALSE
```

```
2 <= 2
## [1] TRUE
(!(4 > 5) & (2+1 == 1+2)) | (0 > 1)
## [1] TRUE
```

`R` kennt die Wahrheitswerte `TRUE`, `FALSE` und `NA` (*not available*, dazu später mehr). Wahrheitswerte sind vom Typ `logical`.

```
typeof(TRUE)
## [1] "logical"
```

Operatoren auf dem Typ `logical` sind: `|` (oder), `&` (und), `!` (nicht). Die Operatoren `<`, `>`, `<=`, `>=`, `!=` (ungleich) `==` (gleich) ergeben einen Wahrheitswert durch den Vergleich von Zahlen.

Wegen beschränkter Maschinengenauigkeit ist bei Gleichheitstests für `double`-Werte besondere Vorsicht geboten.

```
sin(pi) == 0 # ergibt nicht das, was man vielleicht erwartet
## [1] FALSE
abs(sin(pi)) < 1e-14 # besser
## [1] TRUE
```

2.3 Hilfe

Mit dem **Hilfsoperator** `?` rufen wir die R-Dokumentation auf. Für mehr Infos zu mathematischen Funktionen siehe zB `?Arithmetic`, `?sin`, `?log`, `?sinh`, `?factorial`, `?sqrt`.

2.4 Operatorpriorität

Die **Operatorpriorität** (*operator precedence*) gibt an, in welcher Reihenfolge Operatoren ausgewertet werden. Zum Beispiel gilt "Punkt vor Strich" und "Arithmetik vor Logik".

```
5 + 4 * 3^2
## [1] 41
1 + 3 < 2 * 4
## [1] TRUE
```

Siehe `?Syntax` für eine Liste der Operatoren geordnet nach Prioritäten.

3 Variablen

Der Zuweisungsoperator `<-` verknüpft **Werte** mit **Namen** zu **Variablen**: `name <- value`. Mit den so definierten Variablen können wir rechnen.

```
variable_name <- 5
asdf <- 7
variable_name
## [1] 5
asdf
## [1] 7
variable_name * asdf
## [1] 35
asdf <- asdf + 1
asdf
## [1] 8
```

Um den Wert einer Variablen zu erhalten, geben wir ihren Namen nach der Eingabeaufforderung ein.

Variablen haben **Typen**. Diese werden automatisch zugewiesen (**implizite Typisierung**, *implicit typing*). Im Gegensatz dazu muss in einer Programmiersprache mit expliziter Typisierung (*explicit typing*) wie zB `C` der Typ beim Erstellen der Variable angeben werden (R: `x <- 1.23`, C: `double x = 1.23;`).

Den **Typ** einer Variable finden wir mit der Funktion `typeof()` heraus:

```
b <- TRUE
typeof(b)
## [1] "logical"
i <- 1
typeof(i)
## [1] "double"
x <- 3.14
typeof(x)
## [1] "double"
word <- "hallo"
typeof(word)
## [1] "character"
```

Gültige **Namen** bestehen aus Buchstaben, Zahlen, `.`, oder `_` und beginnen mit einem Buchstaben oder einem Punkt nicht gefolgt von einer Zahl. `.8sam`, `8sam` sind keine gültigen Namen , `..8sam`, `._.` schon. Außerdem dürfen **reservierte Wörter** nicht als Variablennamen verwendet werden. Die reservierten Wörter sind:

```
if else repeat while function for in next break
TRUE FALSE NULL Inf NaN NA NA_integer_ NA_real_ NA_complex_ NA_character_
...
```

sowie `..1`, `..2`, `..3`, usw.

4 Skripte

Um eine Folge von Befehlen zu speichern, können diese in eine Textdatei geschrieben werden. Die Textdatei bekommt die Endung `.R`, um anzuzeigen, dass es sich um R-Code handelt. Wir bezeichnen diese Dateien als (R-)Skripte. Das Konsolenprogramm `Rscript` führt Skripte aus. In R wird mittels `source("Pfad/zum/Skript.R")` ein Skript ausgeführt.

Alle Zeichen nach und inklusive `#` sind **Kommentare** und werden bei der Ausführung ignoriert.

Befehle werden durch Zeilenumbrüche getrennt. Um mehrere Befehle in eine Zeile zu schreiben, müssen sie durch ein Semikolon `;` getrennt werden. Dadurch wird der Code jedoch unübersichtlich.

R wird (typischerweise) nicht *kompiliert* sondern *interpretiert*. Das heißt, wir erhalten keine ausführbare Datei (wie etwa bei C) sondern verwenden immer unseren Code (in Form einer Textdatei) und die Programme `R` oder `Rscript`, um den Code auszuführen.

5 RStudio

Für die meisten Anwender ist es komfortabler eine **integrierte Entwicklungsumgebung** (*integrated development environment*, IDE) zu verwenden als Terminal und Texteditor. Die verbreitetste IDE für R ist **RStudio** (https://www.rstudio.com/). Wie R selbst ist auch RStudio *open source*.

RStudio greift auf die lokale Instanz von R zu, um Code auszuführen. R muss also für die Nutzung von RStudio installiert sein.

Es ist hilfreich einige **Short-Cuts** von RStudio zu lernen.

- `Windows` (`Mac` falls abweichend): Beschreibung
- `Alt+-` (`Option+-`): füge `<-` ein

- `Ctrl+Z` (`Cmd+Z`), `Ctrl+Shift+Z` (`Cmd+Shift+Z`): undo bzw redo
- `Ctrl+1`, `Ctrl+2`: fokussiere Texteditor / Konsole
- `Ctrl+S` (`Cmd+S`): Datei speichern
- `Ctrl+F` (`Cmd+F`): Suchen und Ersetzen
- `Ctrl+Shift+Enter` (`Cmd+Shift+Return`): `source()` aktuelle Datei
- `Ctrl+Enter` (`Cmd+Return`): führe aktuelle Zeile (oder markierten Code) aus
- `Alt+Shift+K` (`Option+Shift+K`): Übersicht über Short-Cuts anzeigen
- `Tab`: Auto-Vervollständigung
- `Up-Arrow`: (nur in der Konsole) zuletzt ausgeführte Befehle aufrufen

Unter `Tools` → `Global Options` → `Appearance` kann ein Stil für das Syntax-Highlighting ausgewählt werden. Dieser sorgt dafür, dass in R-Skripten zB Strings in einer anderen Farbe als Zahlen dargestellt werden. Für differenzierteres Syntax-Highlighting sollte `Tools` → `Global Options` → `Code` → `Display` → `Highlight R function calls` aktiviert sein.

6 Atomare Vektoren

Daten können in R in unterschiedlichen *Datenstrukturen* gespeichert werden. Die einfachste Datenstruktur ist der **atomare Vektor**. Mit der Funktion `c()` (*combine*) können wir mehrere Werte (oder atomare Vektoren) gleichen Typs zu einem neuen atomaren Vektor zusammensetzen.

```
x <- c(1, 3, 5)
x
## [1] 1 3 5
y <- c(0, x, -42, x)
y
## [1]   0   1   3   5 -42   1   3   5
```

Die Anzahl der Elemente eines atomaren Vektors erhalten wir mit der Funktion `length()`.

```
length(x)
## [1] 3
length(y)
## [1] 8
```

Der Operator `:` erzeugt einen atomaren Vektor von auf- oder absteigenden Zahlen mit Differenz 1.

```
1:5
## [1] 1 2 3 4 5
3:-3
## [1]  3  2  1  0 -1 -2 -3
0.5:3.9
## [1] 0.5 1.5 2.5 3.5
```

Etwas mehr Kontrolle über die Sequenz der Zahlen haben wir mit der Funktion `seq()`.

```
seq(0, 3, by=0.5)
## [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0
seq(0, 3, length.out=5)
## [1] 0.00 0.75 1.50 2.25 3.00
```

Die Funktion `rep()` wiederholt einen Vektor.

```
rep(c(0,2), times=3)
## [1] 0 2 0 2 0 2
rep(c(0,2), each=3)
## [1] 0 0 0 2 2 2
```

Siehe dazu auch die entsprechenden Seiten der Dokumentation mittels `?':'`, `?seq`, `?rep`.

Die meisten Funktionen und Operatoren in R sind *vektorisiert*, dh, sie geben sinnvolle Ausgaben, wenn die Argumente nicht einzelne Werte sondern atomare Vektoren sind.

```
1:3 + 4:6
## [1] 5 7 9
2 * 1:3
## [1] 2 4 6
c(0,2,4) * 1:3
## [1]  0  4 12
sin(seq(0, 2*pi, length.out=6))
## [1]  0.000000e+00  9.510565e-01  5.877853e-01 -5.877853e-01 -9.510565e-01
## [6] -2.449213e-16
sum(1:99) # Summe aller Elemente des atomaren Vektors 1:99
## [1] 4950
x <- 1:5 < 5:1
x
## [1]  TRUE  TRUE FALSE FALSE FALSE
any(x) # Enthält das Argument x mindestens einmal TRUE?
## [1] TRUE
all(x) # Sind alle Elemente von x TRUE?
## [1] FALSE
```

6.1 Indizierung

Mit `x[i]` greifen wir auf das `i`-te Element des atomaren Vektors `x` zu. Wir können mit `[` auch mehrere Elemente auswählen.

```
x <- 11:15
x[2]
## [1] 12
x[c(1,4)]
## [1] 11 14
```

Achtung: Der erste Eintrag hat den Index 1 (nicht 0 wie in C).

Eine weitere Möglichkeit, Einträge eines Vektors auszuwählen, ist ein atomarer Vektor von Wahrheitswerten.

```
x[c(T, F, T, F, T)]
## [1] 11 13 15
```

Hierbei sind `T` und `F` Abkürzungen für `TRUE` und `FALSE`.

Dies ist besonders hilfreich in Zusammenhang mit vektorisierten Funktionen und Operatoren, deren Rückgabewert vom Typ `logical` ist.

```
x
## [1] 11 12 13 14 15
x %% 2
## [1] 1 0 1 0 1
x %% 2 == 0
## [1] FALSE  TRUE FALSE  TRUE FALSE
x[x %% 2 == 0] # wähle alle geraden Einträge
## [1] 12 14
```

Wir können beim Indizieren auch direkt neue Werte zuweisen.

```r
x[c(1,5)] <- c(-1, -2)
x
## [1] -1 12 13 14 -2
x[2:4] <- 0
x
## [1] -1  0  0  0 -2
```

6.2 Skalare

R kennt keine besonderen Typen oder Datenstrukturen für Skalare. Skalare sind einfach atomare Vektoren mit Länge 1.

```r
length(0)
## [1] 1
y <- 5
y[1]
## [1] 5
```

6.3 Copy on Modify

R setzt eine sogenannte **Copy-on-Modify**-Semantik um, dh eine Änderung einer Variablen führt zu einer Kopie des Wertes. Insbesondere gibt es keine Referenzen oder Pointer wie in C/C++.

```r
x <- 1:5
y <- x # Kopie
y[3] <- -10
y # wurde geändert
## [1]   1   2 -10   4   5
x # bleibt unverändert
## [1] 1 2 3 4 5
```

6.4 Fehlende Werte

Mit `NA` (*not available*) wird ein fehlender Wert markiert.

```r
x <- 1:5
x[3:7]
## [1]  3  4  5 NA NA
```

Sehr ähnlich zu `NA` ist der Wert `NaN`, der bei undefinierten Berechnungen zurückgegeben wird.

```r
0/0
## [1] NaN
log(-1)
## Warning in log(-1): NaNs produced
## [1] NaN
```

Achtung: Tendenziell liefern Funktionen in R eher Werte wie `NA` oder `NaN` anstatt eine explizite Fehlermeldung auszugeben. Dies kann es manchmal erschweren Fehler im Code zu finden.

Vorsicht ist geboten bei mathematischen und logischen Operationen mit `NA` und `NaN`!

```r
0 + NaN
## [1] NaN
NA | FALSE
## [1] NA
1:3 < c(0, NaN, 5)
```

```r
## [1] FALSE    NA  TRUE
NA != 1
## [1] NA
NA == NA
## [1] NA
```

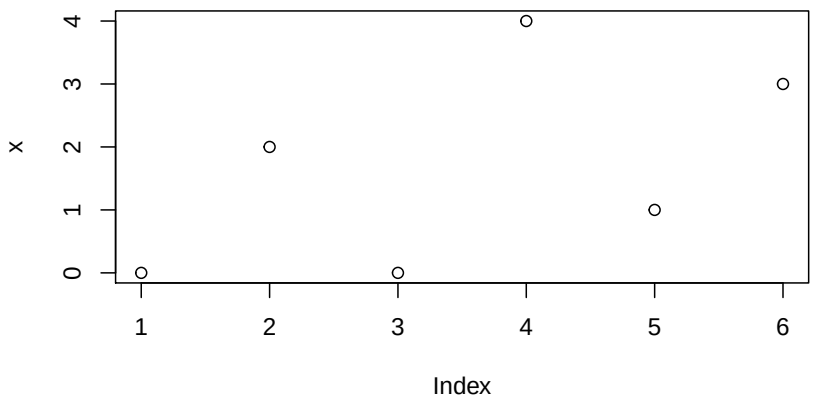
Um zu testen, ob eine Variable den Wert `NA` hat, kann `==` also nicht genutzt werden. Stattdessen verwenden wir die Funktion `is.na()`.

```r
c(is.na(NA), is.na(NaN), is.na(0))
## [1]  TRUE  TRUE FALSE
# is.na() ist vektorisiert:
x <- c(1, NA, 2, NaN, 3)
x
## [1]   1  NA   2 NaN   3
is.na(x)
## [1] FALSE  TRUE FALSE  TRUE FALSE
x[!is.na(x)] # x ohne NA-Werte
## [1] 1 2 3
x[is.na(x)] <- 0
x
## [1] 1 0 2 0 3
```

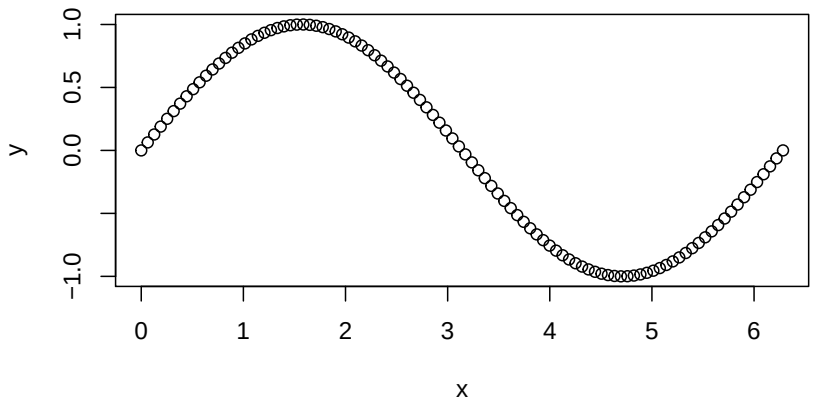
6.5 Plots

Mit der Funktion `plot()` werden die Werte eines numerischen atomaren Vektors grafisch dargestellt.

```r
x <- c(0,2,0,4,1,3)
plot(x)
```




```r
# Funktion plotten, Funktionsgraph zeichnen:
x <- seq(0, 2*pi, length.out = 100)
y <- sin(x)
plot(x, y)
```

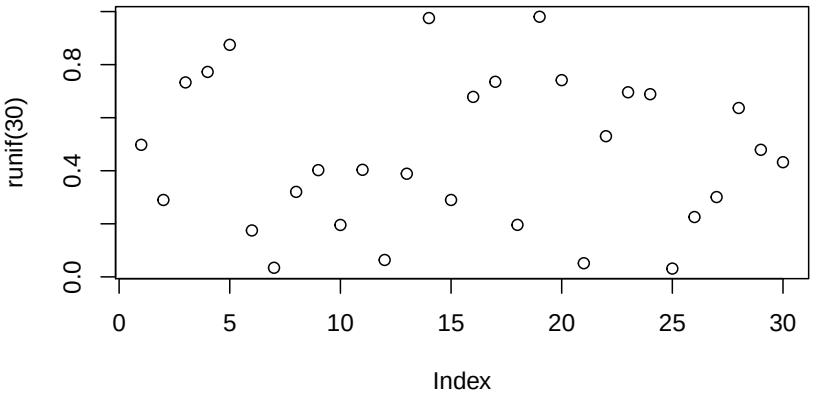


Die `plot()`-Funktion wird später noch ausführlicher vorgestellt.

6.6 Simulation von Zufallsexperimenten

R zeichnet sich durch einige nützliche Funktionen zum Simulieren von Zufallsexperimenten aus. Zum Beispiel erzeugen wir mit `runif()` auf dem Intervall $[0, 1]$ gleichverteilte Zufallszahlen.

```
runif(3)
## [1] 0.08075014 0.83433304 0.60076089
runif(3)
## [1] 0.157208442 0.007399441 0.466393497
plot(runif(30))
```



Weitere Funktionen im Zusammenhang mit Zufallszahlen und Verteilungen werden später noch ausführlicher vorgestellt.

6.7 Typen

Wie bereits erwähnt, kann R mit atomare Vektoren verschiedenen Typs umgehen. Der Typ kann mit der Funktion `typeof()` abgefragt werden. Wir haben bereits mit atomaren Vektoren vom Typ `double` und

`logical` gearbeitet. Der Typ für Text heißt `character`.

```
typeof(c(3, pi, exp(1)))
## [1] "double"
typeof(c(T, F, T))
## [1] "logical"
typeof(c("asdf", "Hallo Welt!"))
## [1] "character"
```

6.8 Text

Die Elemente eines atomaren Vektors vom Typ `character` sind Zeichenketten (und nicht einzelne Buchstaben), genannt **Strings**. Mit `length()` erhalten wir die Anzahl der Strings eines `character`-Vektors. Die Anzahl der Symbole in einem String wird von `nchar()` zurückgegeben.

```
length("ein String")
## [1] 1
nchar("ein String")
## [1] 10
x <- c("zwei", "Strings")
length(x)
## [1] 2
nchar(x)
## [1] 4 7
```

`cat()` und `print()` geben Werte auf der Konsole aus.

```
x <- 1:10
x
##  [1]  1  2  3  4  5  6  7  8  9 10
print(x)
##  [1]  1  2  3  4  5  6  7  8  9 10
cat(x)
## 1 2 3 4 5 6 7 8 9 10
```

Hier ist `print(x)` gleich der Eingabe von `x` in der Konsole. Arbeitet man mit Skripten und dem Befehl `source()` oder definiert man selbst Funktionen (nächster Abschnitt), wird die Ausgabe bei `x` unterdrückt und muss explizit mit `print(x)` gefordert werden.

Im Unterschied zu `print()` nimmt `cat()` eine beliebige Anzahl an Argumenten. Diese werden mit einem Leerzeichen getrennt zusammengefügt. Das Trennsymbol kann mit dem Argument `sep` bestimmt werden. `cat()` erzeugt automatisch keinen Zeilenumbruch. Dieser kann mit `\n` eingefügt werden.

```
cat(1);cat("A")
## 1A
```

```
cat(1, "\n");cat(2)
## 1
## 2
cat(1:3, "A\nB\n\nC\n", sep="_")
## 1_2_3_A
## B
##
## C
```

Für die Umwandlung von Zahlen in den Typ `character` ohne direkte Ausgabe auf der Konsole, siehe `?sprintf`, `?as.character`, `?paste`.

```r
str1 <- as.character(pi)
str1
## [1] "3.14159265358979"
str2 <- sprintf("Pi ist ungefähr %.2f und e ist ca. %.2f", pi, exp(1))
str2
## [1] "Pi ist ungefähr 3.14 und e ist ca. 2.72"
str3 <- paste0("pi = ", pi, " und e = ", exp(1))
str3
## [1] "pi = 3.14159265358979 und e = 2.71828182845905"
```

7 Funktionen definieren

Mit dem reservierten Wort `function` können wir eigene Funktionen definieren. Die Syntax dafür folgt dem Schema `fun_name <- function(arg1, arg2, arg3, ...) expression`.

`expression` ist hierbei ein einzelner Befehl oder eine Folge von Befehlen (getrennt durch Zeilenumbruch oder `;`), die von `{ }` umschlossen ist.

```r
myfun <- function(x, y) {
  thesum <- x+y
  2*thesum
}
myfun(1, 2)
## [1] 6

f <- function(x, y) x + 10*y
f(1,2)
## [1] 21
```

Der Rückgabewert der Funktion ist der letzte ausgewertete Ausdruck. Alternativ können wir den Rückgabewert explizit mit `return()` bestimmen. Befehle nach `return()` werden nicht ausgeführt.

```r
myfun <- function(x, y) {
  thesum <- x+y
  x # führt nicht zu Ausgabe auf Konsole
  print(y) # Ausgabe auf Konsole
  return(2*thesum)
  print(123) # ignoiert, da nach return()
}
myfun(1, 2)
## [1] 2
## [1] 6
```

Die Argumente einer Funktion können **Default-Werte** haben. Schema: `function(arg1=default1, arg2=default2, ...)`. Argumente mit Default-Werten müssen beim Aufruf nicht angegeben werden. Sie sind daher **optionale Argumente**.

```r
f <- function(x, y=0) {
  x + 10*y
}
f(1, 2)
## [1] 21
f(3)
## [1] 3
```

Beim Funktionsaufruf können wir die Namen der Argumente explizit benennen.

```r
f <- function(x, y=0, z=0) {
  x + 10*y + 100*z
}
f(1, z=2)
## [1] 201
f(3, 4, 5)
## [1] 543
f(z=6, y=7, x=8)
## [1] 678
```

Beim Aufruf `f(x)` wird der Wert von `x` kopiert und der Funktion `f()` übergeben (und keine Referenz auf `x`). Dieses Verhalten wird mit **call by value** bezeichnet (im Gegensatz zu **call by reference**).

```r
swapFirstTwo <- function(x) {
  tmp <- x[2]
  x[2] <- x[1]
  x[1] <- tmp
}
x <- 1:3
x
## [1] 1 2 3
swapFirstTwo(x) # kein Effekt
x
## [1] 1 2 3

# Stattdessen müssten wir wie folgt vorgehen:
swapFirstTwo <- function(x) {
  tmp <- x[2]
  x[2] <- x[1]
  x[1] <- tmp
  return(x)
}
x <- 1:3
x <- swapFirstTwo(x)
x
## [1] 2 1 3
```

Bemerkung: Da `swapFirstTwo()` mit einem Befehl realisierbar ist, lohnt es sich nicht eine eigene Funktion dafür zu definieren.

```r
x <- 1:3
x[1:2] <- x[2:1] #swapFirstTwo
x
## [1] 2 1 3
```

Bei längeren Skripten ist es jedoch äußerst hilfreich, den Code dadurch zu **strukturieren**, dass zusammengehörende oder wiederkehrende Folgen von Befehlen zu Funktionen zusammengefasst werden.

8 Listen

Die wichtigste Datenstruktur in R neben atomaren Vektoren sind **Listen**. Dies sind geordnete Sammlungen von R-Objekten beliebigen Typs. Sie können mit `list()` erzeugt werden.

```r
lst1 <- list(c(1,5), c("hallo", "wie geht's"))
lst1
```

```
## [[1]]
## [1] 1 5
##
## [[2]]
## [1] "hallo"      "wie geht's"
lst2 <- list(1, 2, list("a", 3))
lst2
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [[3]][[1]]
## [1] "a"
##
## [[3]][[2]]
## [1] 3
```

Vektor ist der Überbegriff für **atomare Vektoren** und **Listen**.

`length()` gibt die Anzahl der Element einer Liste aus.

```
length(lst1)
## [1] 2
length(lst2)
## [1] 3
```

Listen können geschachtelt werden, atomare Vektoren nicht.

```
list(1, list(2, list(3)))
## [[1]]
## [1] 1
##
## [[2]]
## [[2]][[1]]
## [1] 2
##
## [[2]][[2]]
## [[2]][[2]][[1]]
## [1] 3
c(1, c(2, c(3)))
## [1] 1 2 3
```

Auf einzelne Elemente einer Liste greifen wir mit `[[` zu.

```
lst1[[1]]
## [1] 1 5
lst1[[2]]
## [1] "hallo"      "wie geht's"
```

Wenn wir mehrere Elemente auswählen wollen, benutzen wir `[`. Dann erhalten wir eine Liste der ausgewählten Elemente.

```
lst2[1:2]
## [[1]]
```

```
## [1] 1
##
## [[2]]
## [1] 2
lst2[1] # auch Liste!
## [[1]]
## [1] 1
```

Listen können Elemente beliebigen Typs enthalten, sogar Listen und Funktionen. Im Gensatz dazu sind die Elemente atomarer Vektoren alle vom selben Typ und dieser Typ muss ein **Basistyp** sein (zB `logical`, `double` oder `character`). Es gibt keine atomaren Vektoren mit Elementen vom Typ "Liste" oder Elementen vom Typ "Funktion".

```
m <- list(max, min) # Liste der Funktionen max() und min()
m[[1]](1:3)
## [1] 3
m[[2]](1:3)
## [1] 1
```

Um Listen etwas kompakter auf der Konsole auszugeben, verwenden wir die Funktion `str()`.

```
str(lst1)
## List of 2
##  $ : num [1:2] 1 5
##  $ : chr [1:2] "hallo" "wie geht's"
```

Zwei oder mehrere Listen können mit `c()` zusammengehängt werden.

```
str(c(lst1, lst2))
## List of 5
##  $ : num [1:2] 1 5
##  $ : chr [1:2] "hallo" "wie geht's"
##  $ : num 1
##  $ : num 2
##  $ :List of 2
##   ..$ : chr "a"
##   ..$ : num 3
```

`unlist()` wandelt eine Liste in einen atomaren Vektor; `as.list()` wandelt einen atomaren Vektor in eine Liste.

```
x <- list(1, 2, c(3,4))
str(x)
## List of 3
##  $ : num 1
##  $ : num 2
##  $ : num [1:2] 3 4
unlist(x)
## [1] 1 2 3 4
str(as.list(c(5,6,7)))
## List of 3
##  $ : num 5
##  $ : num 6
##  $ : num 7
```

9 Kontrollstrukturen

Wie aus anderen Programmiersprachen bekannt, gibt es auch in R **Kontrollstrukturen** (*Control Flow*). Wir gehen hier davon aus, dass ihre Funktionsweise bereits bekannt ist und erklären nur die Syntax.

9.1 Bedingte Anweisungen

Die Syntax für bedingte Anweisungen entspricht dem Schema `if (condition) expression` bzw `if (condition) expression_TRUE else expression_FALSE`

`condition` ist ein Ausdruck, der evaluiert entweder `TRUE` oder `FALSE` ergibt.

```
x <- 1
y <- FALSE

if (x > 0) {
  cat("x größer 0\n")
}
## x größer 0

if (y) {
  cat("y ist wahr\n")
} else {
  cat("y ist falsch\n")
}
## y ist falsch
```

Falls der bedingt auszuführende Code nur aus einer Anweisung besteht, können die geschweiften Klammern weggelassen werden. Jedoch ist Vorsicht bei einem darauffolgenden `else` geboten.

```
if (TRUE) print("Das ist immer wahr!")
## [1] "Das ist immer wahr!"
```

```
if (FALSE) 1 # if-Ausdruck endet hier
else 2 # ERROR
```

```
if (FALSE) 1 else 2
## [1] 2
if (FALSE) 1 else
  2
## [1] 2
if (FALSE) {
  1
} else 2
## [1] 2
```

Iterative `if else`- Ausdrücke erlauben die Unterscheidung mehrerer Werte.

```
w <- "blue"
if (w == "red") {
  cat("roses are red\n")
} else if (w == "blue") {
  cat("violets are blue\n")
} else if (w == "green") {
  cat("gras is green\n")
} else {
  cat("color unknown\n")
```

```
}
## violets are blue
```

9.2 while-Schleife

`while`-Schleifen haben in R die Form `while(condition) expression`.

```
x <- 3
while (x > 0) {
  print(x)
  x <- x - 1
}
## [1] 3
## [1] 2
## [1] 1
x
## [1] 0
```

Schleifen können mit `break` abgebrochen werden. Mit `next` wird nur der aktuelle Schleifendurchlauf beendet und mit dem nächsten angefangen.

```
x <- 10
while (TRUE) {
  x <- x - 1
  if (x %% 2 != 0) next
  cat(x)
  if (x <= 0) break
}
## 86420
```

9.3 for-Schleife

Mit einer `for`-Schleife können wir über einen Vektor (atomarer oder Liste) iterieren: `for (iterator in vector) expression`. `next` und `break` können hier analog zu `while`-Schleifen benutzt werden.

```
x <- 0
for (i in 1:10) x <- x + i
x
## [1] 55
lst <- list(FALSE, 1, "two")
for (y in lst) {
  print(length(y))
  cat(y, "\n")
}
## [1] 1
## FALSE
## [1] 1
## 1
## [1] 1
## two
```

In diesem Zusammenhang ist die Funktion `seq_along()` nützlich. Sie erzeugt einen Index-Vektor ihres Argumentes.

```
seq_along(lst)
## [1] 1 2 3
```

```r
for (i in seq_along(lst))
  cat(letters[i], ":", lst[[i]], "\n")
## a : FALSE
## b : 1
## c : two
```

Bemerkung: Der Vektor `letters` enthält die Buchstaben des Alphabets.

```r
letters
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

10 Zufall

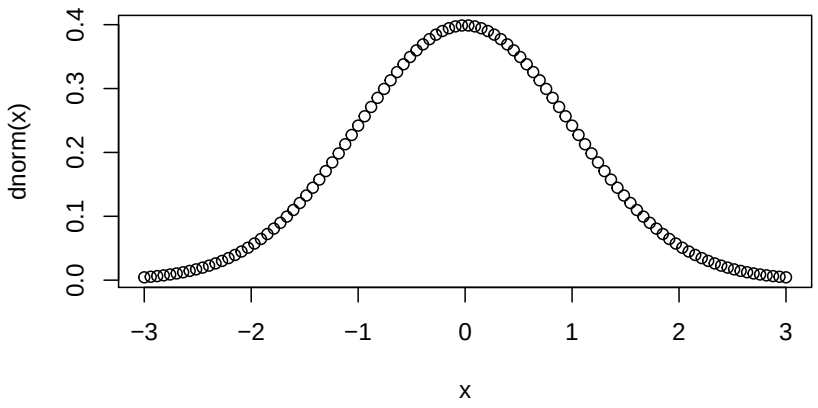
Mit R lassen sich Zufallszahlen aus verschiedenen Verteilungen ziehen. Die Benennung der entsprechenden Funktion folgt dem Schema `rDISTRI()`, wobei `DISTRI` durch ein Kürzel einer Verteilung ersetzt werden muss.

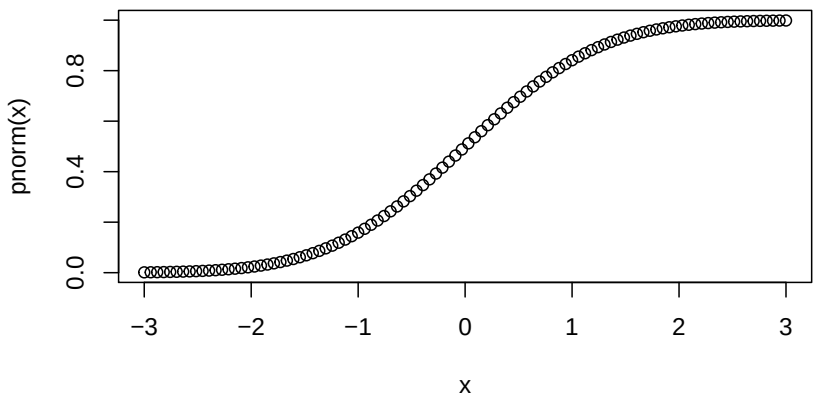
```r
runif(8) # uniform distribution
## [1] 0.7064338 0.9485766 0.1803388 0.2168999 0.6801629 0.4988456 0.6416793
## [8] 0.6602843
rbinom(8, size=5, prob=0.5) # binomial distribution
## [1] 1 3 3 5 5 2 2 2
rnorm(8, mean=0, sd=1) # normal distribution
## [1] -0.93584735 -0.01595031 -0.82678895 -1.51239965  0.93536319  0.17648861
## [7]  0.24368546  1.62354888
```

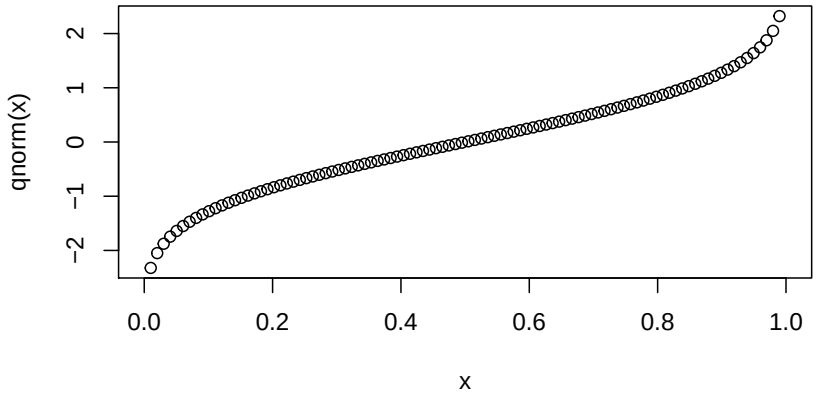
Siehe `?Distributions` für eine Übersicht und eine Beschreibung der Parameter.

Dichte, Verteilungsfunktion und Quantilfunktion folgen dem Schema `dDISTRI()`, `pDISTRI()`, bzw `qDISTRI()`.

```r
x <- seq(-3, 3, length.out=100)
plot(x, dnorm(x))
plot(x, pnorm(x))
x <- seq(0, 1, length.out=100)
plot(x, qnorm(x))
```







Für die `rDISTRI()`-Funktionen benutzt R einen **Pseudo-Zufallszahlengenerator**. Da ein Computer eine deterministische Maschine ist, kann R keine "echten" Zufallswerte erzeugen. Ein Pseudo-Zufallszahlengenerator startet mit einem Initialwert und erzeugt darauf aufbauend eine Folge von Zahlen die "zufällig aussieht", aber vollkommen deterministisch ist. Der Initialwert für den R-internen Pseudo-Zufallszahlengenerator kann mit der Funktion `set.seed()` gesetzt werden. Dies ist sehr nützlich um reproduzierbare "Zufalls-Simulationen" zu erstellen.

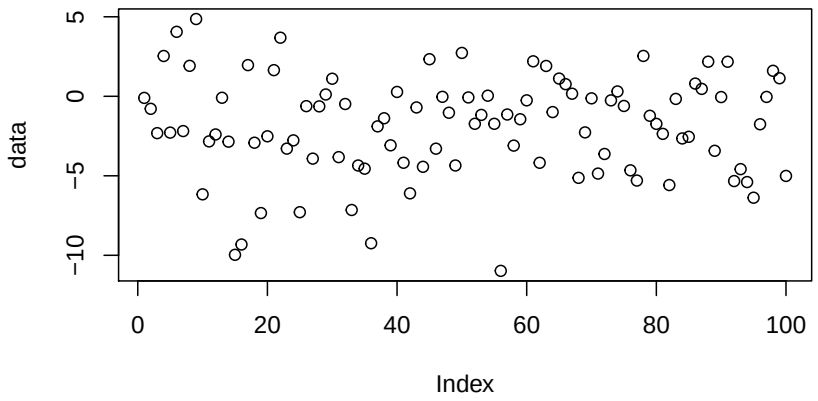
```
runif(3) # 1. Aufruf
## [1] 0.5446034 0.2785972 0.4467025
runif(3) # 2. Aufruf ergibt neue Werte
## [1] 0.37151118 0.02806097 0.46598719
set.seed(42)
runif(3) # 3.
## [1] 0.9148060 0.9370754 0.2861395
runif(3) # 4.
## [1] 0.8304476 0.6417455 0.5190959
set.seed(42)
runif(3) # gleiche Werte wie bei 3.
## [1] 0.9148060 0.9370754 0.2861395
runif(3) # gleiche Werte wie bei 4.
```

```
## [1] 0.8304476 0.6417455 0.5190959
```

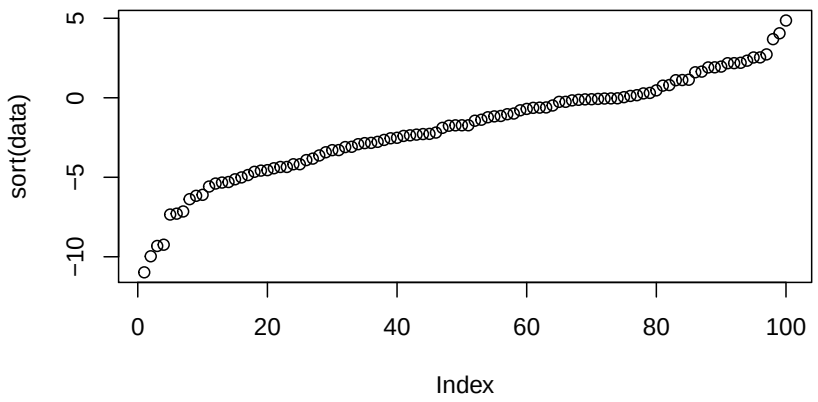
11 Summary Statistics

Wir listen ein paar Funktionen auf, die einfache Zusammenfassungen numerischer Daten erlauben.

```
data <- rnorm(100, mean=-2, sd=3) # erzeuge Daten
plot(data)
```




```
mean(data) # arithmetisches Mittel
## [1] -1.900262
median(data) # Median
## [1] -1.73061
var(data) # empirische Varianz
## [1] 9.86938
sd(data) # empirische Standardabweichung
## [1] 3.141557
min(data) # Minimum
## [1] -10.97927
max(data) # Maximum
## [1] 4.859936
which.min(data) # Argmin
## [1] 56
which.max(data) # Argmax
## [1] 9
summary(data) # mehrere Statistiken, inklusive Quartile
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -10.97927  -3.98502  -1.73061  -1.90026   0.05703   4.85994
plot(sort(data))
```



12 Matrizen

R kann mit Matrizen rechnen. Diese werden mit der Funktion `matrix()` erzeugt.

```
matrix(1:12, nrow=3)
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
matrix(1:12, ncol=3)
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
matrix(1:12, ncol=3, byrow=TRUE)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
```

Das erste Argument ist ein Vektor aller Einträge der Matrix. Mit der Angabe von `nrow` oder `ncol` wird die Zeilen- bzw Spaltenzahl festgelegt. In welcher Reihenfolge die Daten in die Matrix gefüllt werden, kann mit dem Argument `byrow` beeinflusst werden. Standardmäßig wird Spalte für Spalte von oben nach unten befüllt (*column-major order*).

Die Abfrage der Größe einer Matrix wird in folgenden Beispielen gezeigt.

```
m <- matrix(1:12, nrow=3)
nrow(m) # Zeilenzahl
## [1] 3
ncol(m) # Spaltenzahl
## [1] 4
length(m) # Anzahl Einträge gesamt
## [1] 12
dim(m) # Vektor der Zeilen- und Spaltenzahl
```

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

Auf Elemente von Matrizen kann mit der Angabe von zwei Indizes (für Zeile und Spalte) zugegriffen werden. Die Indizes werden durch ein Komma getrennt. Mehrfachauswahl ist analog zu atomaren Vektoren möglich.

```
m
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
m[3,2] # Zeile zuerst (3), Spalte später (2)
## [1] 6
m[2:3,1]
## [1] 2 3
m[c(1,2),c(1,3)] # 2x2 Untermatrix der Elemente aus Zeile 1 und 2,
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
# die gleichzeitig auch in Spalte 1 oder 3 stehen;
# entspricht Streichung von Zeile 3 und Spalten 2 und 4
```

Wird der Zeilen- oder Spaltenindex freigelassen, wird jede Zeile bzw Spalte ausgewählt, dh ganze Spalten bzw Zeilen.

```
m[2, ] # 2. Zeile
## [1]  2  5  8 11
m[ ,c(4,1)] # Spalte 4 und 1
##      [,1] [,2]
## [1,]   10    1
## [2,]   11    2
## [3,]   12    3
```

Wir können mit Matrizen rechnen:

```
m1 <- matrix(0:5, nrow=2)
m2 <- matrix(-5:0, nrow=2)
m1
##      [,1] [,2] [,3]
## [1,]    0    2    4
## [2,]    1    3    5
m2
##      [,1] [,2] [,3]
## [1,]   -5   -3   -1
## [2,]   -4   -2    0
m1 + m2 # elementweise Addition
##      [,1] [,2] [,3]
## [1,]   -5   -1    3
## [2,]   -3    1    5
2 * m1 # skalare Multiplikation
##      [,1] [,2] [,3]
## [1,]    0    4    8
## [2,]    2    6   10
m1 * m2 # elementweise Multiplikation
##      [,1] [,2] [,3]
## [1,]    0   -6   -4
## [2,]   -4   -6    0
```

```
t(m2) # transponierte Matrix
##      [,1] [,2]
## [1,]   -5   -4
## [2,]   -3   -2
## [3,]   -1    0
m1 %*% t(m2) # Matrixmultiplikation
##      [,1] [,2]
## [1,]  -10   -4
## [2,]  -19  -10
```

Mit `cbind()` und `rbind()` werden Vektoren oder Matrizen an den Spalten bzw Zeilen zu neuen Matrizen verbunden.

```
cbind(m1, c(10,20))
##      [,1] [,2] [,3] [,4]
## [1,]    0    2    4   10
## [2,]    1    3    5   20
rbind(m2, m1)
##      [,1] [,2] [,3]
## [1,]   -5   -3   -1
## [2,]   -4   -2    0
## [3,]    0    2    4
## [4,]    1    3    5
```

Weitere Funktionen auf Matrizen:

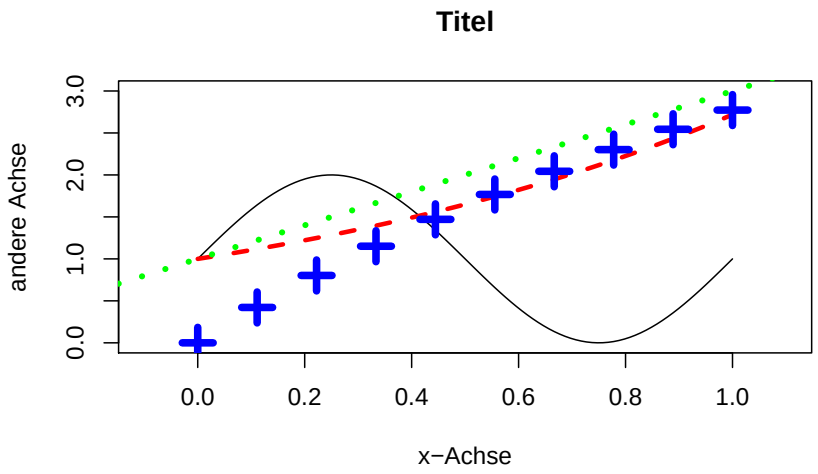
```
m <- matrix(1:9, nrow=3)
m[2,2] <- 0
m[3,3] <- 0
m
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    0    8
## [3,]    3    6    0
det(m) # Determinante
## [1] 132
y <- solve(m, rep(1,3)) # löse Gleichungssystem
y
## [1] 0.19696970 0.06818182 0.07575758
m %*% y
##      [,1]
## [1,]    1
## [2,]    1
## [3,]    1
```

13 Plots

Die Funktion `plot()` hat einige optionale Argumente, mit denen die Gestalt des Plots beeinflusst werden kann.

```
x <- seq(0, 1, length.out=100)
y1 <- sin(2*pi*x)+1
y2 <- exp(x)
x2 <- seq(0, 1, length.out=10)
y3 <- log(x2+1)*4
```

```r
plot(x, y1, # Erstelle neuen Plot
     type="l", # Zeichne Plot vom Typ "l": Linie
     xlim=c(-0.1,1.1), ylim=c(0, 3), # c(min,max) der jeweiligen Achse
     main="Titel", # Überschrift
     xlab="x-Achse", ylab="andere Achse") # Achsenbeschriftung
lines(x, y2, # Füge zum bestehenden Plot eine neue Kurve vom Typ "l" (Linie) hinzu
      lty=2, # Linientyp 2: gestrichelt
      lwd=3,
      col="red") # Farbe rot
points(x2, y3, # Füge zum bestehenden Plot eine neue Kurve vom Typ "p" (Punkte) hinzu
       pch=3, # Symbol Nummer 3 (plus) für die Punkte verwenden
       col="#0000FF", # Farbe im Format "#RRGGBB" R: rot, G: grün, B: blau, im Hexadezimalsystem
       cex=2, # Größe des Symbols
       lwd=5) # Liniendicke für das Malen des Symbols
abline(1,2, # Füge zum bestehenden Plot eine neue Linie mit intercept 1 und Steigung 2 hinzu
       col="green",
       lwd=4,
       lty=3) # Linientyp 3: gepunktet
```

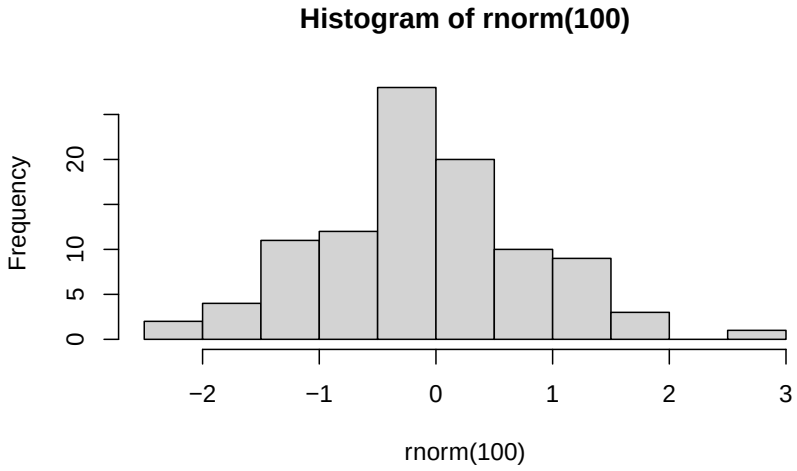


Für eine Dokumentation der Plotfunktion siehe `?plot.default`.

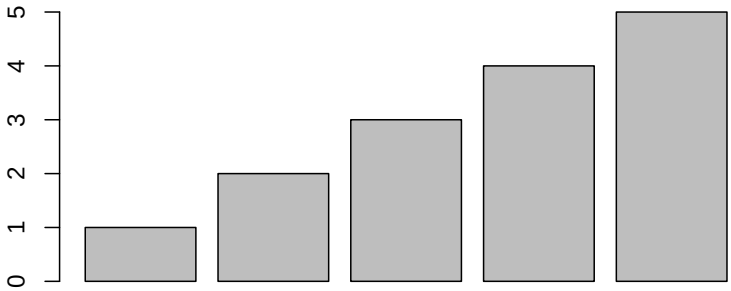
Die Funktion `plot()` erzeugt standardmäßig einen neuen Plot, wohingegen `points()`, `lines()` und `abline()` in den zuletzt erzeugten Plot zeichnen.

Wir zeigen noch beispielhaft weitere Möglichkeiten Plots zu erzeugen. Um Details herauszufinden, können wir jederzeit die Dokumentation mittels `?` konsultieren.

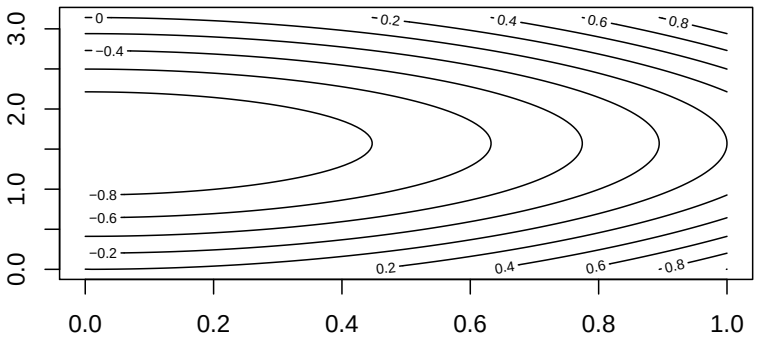
```r
hist(rnorm(100))
```




```r
barplot(1:5)
```




```r
x <- seq(0,1,length.out = 100)
y <- seq(0,pi,length.out = 100)
f <- function(x,y) {
  x^2-sin(y)
}
z <- matrix(nrow = length(x), ncol=length(y)) # erzeuge leere Matrix der angegebenen Dimensionen
for (ix in seq_along(x)) {
  for (iy in seq_along(y)) {
    z[ix, iy] <- f(x[ix], y[iy])
  }
}
contour(x, y, z)
```



14 Pakete

Pakete erweitern die Liste der in R verfügbaren Funktionen.

`library(package_name)` macht den Inhalt eines Paketes verfügbar.

Ist das Paket noch nicht lokal vorhanden, muss es zunächst heruntergeladen und installiert werden. Dies geschieht mittels `install.packages("package_name")`.

`install.packages()` muss nur einmal aufgerufen werden und sollte nicht Teil von R-Skripten sein. `library()` muss bei jeder neuen "R-Session" aufgerufen werden.

Ist ein Paket installiert aber noch nicht mit `library()` geladen, können Funktionen des Paketes auch mittels `package_name::function_name()` aufgerufen werden.

Das Paket `lubridate` stellt Funktionen zum Umgang mit Kalenderdaten zur Verfügung. ZB wandelt die Funktion `lubridate::dmy()` einen Text-String in ein Datums-Objekt. Mit `lubridate::day()`, `lubridate::month()`, `lubridate::year()` erhalten wir die entsprechenden Komponenten das Datums.

```
# install.packages("lubridate") bereits ausgeführt
x <- lubridate::dmy("20.4.20")
lubridate::day(x)
## [1] 20
lubridate::month(x)
## [1] 4
```

```
month(x) # ERROR: Funktion unbekannt
```

```
library(lubridate)
##
## Attaching package: 'lubridate'
## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
month(x)
## [1] 4
year(x)
## [1] 2020
```

15 Tibbles

Die gängigste Datenstruktur, mit der Datensätze in R verfügbar gemacht werden, sind **Tibbles**. Tibbles sind Tabellen. Sie haben eine Rechteckstruktur wie Matrizen. Jede Spalte steht für eine Beobachtungsvariable / ein Feature und jede Zeile für eine Beobachtung. Im Gegensatz zu Matrizen können verschiedene Spalten von unterschiedlichem Typ sein.

Tibbles werden vom Paket `tibble` bereitgestellt.

```
library(tibble)
tbl <- tibble( # Erzeugen eines Tibbles
  name = c("Alice", "Bob", "Claire", "Dave"), # Variable 1 / Spalte 1 hat die Bezeichnung 'name' und
  # als Wert den angegebenen atomaren Vektor vom Typ character
  age = c(21, 25, 37, 22), # Variable 2 / Spalte 2: Bezeichnung 'age', Typ: double
  left_handed = c(T, F, F, T) # Variable 3 / Spalte 3: Bezeichnung 'left_handed', Typ: logical
  # alle Spalten haben Länge 4
)
tbl
## # A tibble: 4 x 3
##   name     age left_handed
##   <chr>  <dbl> <lgl>
## 1 Alice     21 TRUE
## 2 Bob       25 FALSE
## 3 Claire    37 FALSE
## 4 Dave      22 TRUE
tbl[1,] # Indizierung analog zu Matrizen
## # A tibble: 1 x 3
##   name    age left_handed
##   <chr> <dbl> <lgl>
## 1 Alice    21 TRUE
tbl[,c(1,3)]
## # A tibble: 4 x 2
##   name   left_handed
##   <chr>  <lgl>
## 1 Alice  TRUE
## 2 Bob    FALSE
## 3 Claire FALSE
## 4 Dave   TRUE
tbl$age # Außerdem können benannte Spalten mir $ ausgewählt werden
## [1] 21 25 37 22
```

15.1 Daten einlesen

Typischerweise möchte man externe Daten einlesen und nicht Daten von Hand im R-Code erstellen. Daten können in den unterschiedlichsten Formaten vorliegen. Eines der einfachsten Formate ist `.csv` (*comma seperated values*). Dateien dieses Formats sind einfache Textdateien, die eine Tabellenstruktur haben, wobei Beobachtungen durch Zeilenumbrüche und Spalten durch `,` oder `;` getrennt werden. Um eine solche Datei als Tibble einzulesen, nutzen wir die Funktion `readr::read_csv()`:

```
library(readr)
cntrs <- read_csv("NV01_countries.csv")
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   Population = col_double(),
##   `Area (sq. mi.)` = col_double(),
```

```
##   `Infant mortality (per 1000 births)` = col_number(),
##   `GDP ($ per capita)` = col_double(),
##   `Literacy (%)` = col_number(),
##   `Other (%)` = col_number(),
##   Climate = col_number(),
##   Birthrate = col_number(),
##   Deathrate = col_number()
## )
## See spec(...) for full column specifications.
```

Die Datei `"01_countries.csv"` listet Länder mit verschiedenen Daten wie Fläche, Bevölkerungszahl, etc. `read_csv()` gibt einige Informationen aus, wie die Datei eingelesen wurde. Diese ignorieren wir für den Moment.

Der Datensatz ist zu groß, um ihn vollständig in der Konsole auszugeben. Solch große Tibbles werden automatisch verkürzt ausgegeben.

```
nrow(cntrs) # Zeilenzahl
## [1] 227
ncol(cntrs) # Spaltenzahl
## [1] 20
cntrs
## # A tibble: 227 x 20
##    Country Region Population `Area (sq. mi.)` `Pop. Density (~ `Coastline (coa~
##    <chr>   <chr>       <dbl>            <dbl> <chr>            <chr>
##  1 Afghan~ ASIA ~   31056997           647500 48,0             0,00
##  2 Albania EASTE~    3581655            28748 124,6            1,26
##  3 Algeria NORTH~   32930091          2381740 13,8             0,04
##  4 Americ~ OCEAN~      57794              199 290,4            58,29
##  5 Andorra WESTE~      71201              468 152,1            0,00
##  6 Angola  SUB-S~   12127071          1246700 9,7              0,13
##  7 Anguil~ LATIN~      13477              102 132,1            59,80
##  8 Antigu~ LATIN~      69108              443 156,0            34,54
##  9 Argent~ LATIN~   39921833          2766890 14,4             0,18
## 10 Armenia C.W. ~    2976372            29800 99,9             0,00
## # ... with 217 more rows, and 14 more variables: `Net migration` <chr>, `Infant
## #   mortality (per 1000 births)` <dbl>, `GDP ($ per capita)` <dbl>, `Literacy
## #   (%)` <dbl>, `Phones (per 1000)` <chr>, `Arable (%)` <chr>, `Crops
## #   (%)` <chr>, `Other (%)` <dbl>, Climate <dbl>, Birthrate <dbl>,
## #   Deathrate <dbl>, Agriculture <chr>, Industry <chr>, Service <chr>
```

Mit `View()` wird ein Datensatz im RStudio-Data-Viewer gezeigt. Alternativ kann dieser mit einem Links-klick auf die Variable im *Environment-Pane* in RStudio geöffnet werden.

16 apply-Funktionen

Um den Code übersichtlich und besser verständlich zu machen, werden in R oft Funktionen wie `apply()`, `lapply()` und `sapply()` anstatt `for`-Schleifen eingesetzt.

Mit `lapply(x, f)` wird die Funktion `f()` auf jedes Element des Vektors `x` angewendet.

```
lst <- list(T, 42, "asdf")
fun <- function(x) {
  paste0("value: ", x, ", type: ", typeof(x))
}
```

```
res <- lapply(lst, fun)
res
## [[1]]
## [1] "value: TRUE, type: logical"
##
## [[2]]
## [1] "value: 42, type: double"
##
## [[3]]
## [1] "value: asdf, type: character"
```

Die Ausgabe von `lapply()` ist immer eine Liste. Sind die Ausgabewerte der Funktion immer vom selben Typ, kann stattdessen mit `sapply()` ein atomarer Vektor (oder eine Matrix) ausgegeben werden.

```
sapply(lst, fun)
## [1] "value: TRUE, type: logical"   "value: 42, type: double"
## [3] "value: asdf, type: character"
```

Mit einer `for`-Schleife würden wir statt `res <- sapply(lst, fun)` folgenden, etwas unübersichtlicheren Code schreiben.

```
res <- rep("", length(lst))
for (i in seq_along(lst)) {
  res[i] <- fun(lst[[i]])
}
```

Der Befehl `apply(mat, 1, f)` für eine Matrix `mat` und eine Funktion `f()` wendet `f()` auf jede Zeile von `mat` an. Dabei muss `f(x)` für atomare Vektoren `x` ein sinnvolles Ergebnis liefert.

```
mat <- matrix(1:12, nrow = 3)
mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
apply(mat, 1, mean) # Mittelwert für jede Zeile
## [1] 5.5 6.5 7.5
fun <- function(x) c(min(x), max(x))
apply(mat, 1, fun)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   10   11   12
```

Analog wendet man mit `apply(mat, 2, f)` die Funktion `f()` auf die Spalten der Matrix `mat` an.

17 Statistik

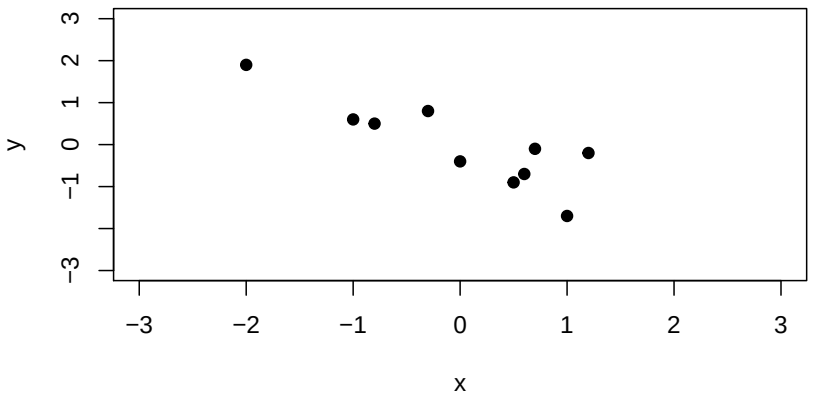
R ist in besonderem Maße dafür ausgelegt, statistische Auswertungen und Simulationen durchzuführen und Resultate ggf. grafisch darzustellen. Dies haben wir bereits in einigen Abschnitten bemerken können:

- erzeuge Zufallszahlen mit `rDISTRI()`
- Verteilungs-, Dichte- und Quantil-Funktionen: `pDISTRI()`, `dDISTRI()`, `qDISTRI()`
- Datensätze als Tibbles einlesen
- grafische Darstellung mit `plot()`, `hist()`, ...
- simple Statistiken der Daten berechnen: `mean()`, `var()`, ...

17.1 Lineare Regression

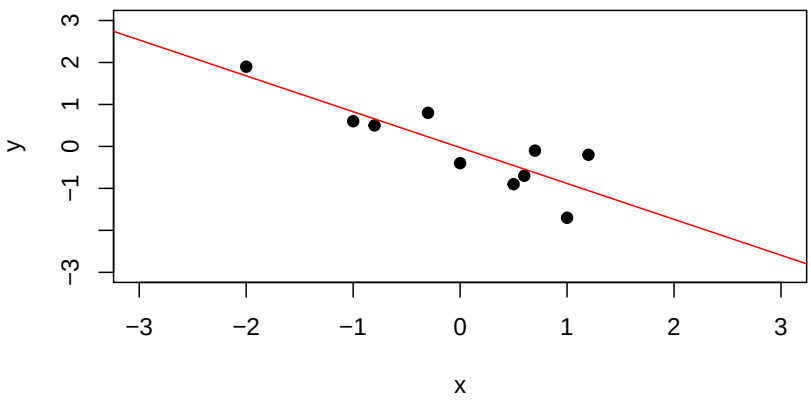
Gegeben seien folgende Daten:

```
x <- c(-2, -1, -0.8, -0.3, 0, 0.5, 0.6, 0.7, 1, 1.2)
y <- c(1.9, 0.6, 0.5, 0.8, -0.4, -0.9, -0.7, -0.1, -1.7, -0.2)
plot(x, y, xlim = c(-3, 3), ylim = c(-3, 3), pch = 19)
```



Wir suchen eine Gerade der Form `g(x) = intercept + slope * x`, wobei `intercept` und `slope` reelle Zahlen sind, sodass `g(x)` möglichst nahe an `y` ist (genauer: minimiere quadratischen Fehler `sum((g(x)-y)^2)`). In R lässt sich dies mit Hilfe der Funktion `lm()` (*linear model*) schnell umsetzen.

```
fit <- lm(y ~ x)
fit
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)            x
##    -0.02855     -0.85468
plot(x, y, xlim = c(-3, 3), ylim = c(-3, 3), pch = 19)
abline(fit, col = "red")
```



Der Rückgabewert der Funktion `lm()` ist eine Liste mit verschiedenen benannten Werten. Die Namen erhalten wir mit `names()`.

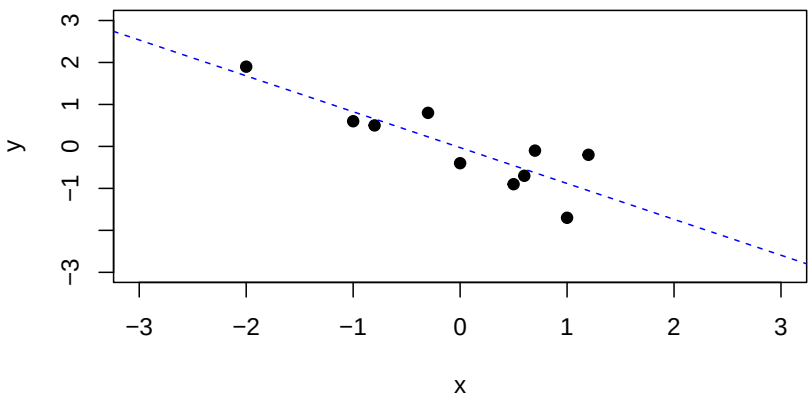
```
typeof(fit)
## [1] "list"
names(fit)
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

Die gesuchten Koeffizienten `intercept` und `slope` finden wir im ersten Eintrag (als atomarer Vektor vom Typ `double` mit Länge zwei). Dieser hat den Namen `"coefficients"`.

```
fit[[1]]
## (Intercept)           x
## -0.02854677 -0.85467688
fit$coefficients # benannte Listen-Einträge können mit '$' abgerufen werden
## (Intercept)           x
## -0.02854677 -0.85467688
```

Im obigen Plot findet R automatisch diesen Eintrag in `fit`, um die Gerade mittels `abline()` zu zeichnen. Wir können dies auch "von Hand" ausführen.

```
coeff <- fit$coefficients
plot(x, y, xlim = c(-3, 3), ylim = c(-3, 3), pch = 19)
abline(coeff[1], coeff[2], col = "blue", lty=2)
```



Die Funktion `lm()` macht mehr, als nur die Koeffizienten zu bestimmen. Eine Zusammenfassung der Resultate erhalten wir mit

```
summary(fit)
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.8168 -0.3351 -0.1569  0.4499  0.8542
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.02855    0.17626  -0.162  0.87536
## x           -0.85468    0.18307  -4.669  0.00161 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5574 on 8 degrees of freedom
## Multiple R-squared:  0.7315, Adjusted R-squared:  0.6979
## F-statistic: 21.79 on 1 and 8 DF,  p-value: 0.001606
```

Wir werden im weiteren Verlauf der Vorlesung noch genauer darauf eingehen, was diese Ausgaben bedeuten.