Wiederholung: Statistische Software (R)

Cornelius Fritz, Christian Scholbeck

17. April 2020

Kurzversion von Martin Binder

1 Intro

Kursziele

Einführung in R:

- Benutzen der Hilfsfunktion in R
- Vektoren
- Matrizen und Lineare Algebra
- Listen

Arbeiten mit Datensätzen:

- Data Frames und Daten einlesen
- Datensätze aufbereiten
- dplyr und tibble
- Operationen mit Zeichenketten

Visualisierung in R:

- Einfache Grafiken
- Fortgeschrittene Grafiken: ggplot2

HILFE gefunden!!!

Häufige Anwendungsfälle:

- Ich soll den Median ausrechnen, kenne aber den Befehl nicht. ⇒ Nachschlagen in Suchmaschinen/Büchern und ausprobieren.
- Ich denke, dass der Befehl `"med"` enthält. ⇒ `apropos()`.

```
> apropos("med")
```

- Ich weiß nicht mehr welche Argumente der Befehl `median` annimmt oder wie sie heißen. ⇒ `?median`

2 Datentypen

Vektoren in R

DAS Datenobjekt ist ein Vektor mit Elementen des Typs

- **numeric**: ganzzahlige oder Gleitkomma-Werte,
- **character**: beliebige Zeichen,
- **logical**: die Zustände `TRUE` und `FALSE`,
- **list**: ein Objekt beliebigen Typs (rekursive Datenstruktur!). Mehr dazu später.

Jeder Vektor besitzt Elemente **eines!!** Typs und hat eine Länge (`length()`).

Automatische Umwandlung

- Abfrage des Typs eines Vektors: `is.⟨Typ⟩()`

```
> is.numeric(numvec)
> is.character(numvec)
```

- Struktur eines R-Objekts erfragen: `str()`

```
> str(charvec)
```

- R wandelt den Typ eines Objektes *automatisch* um, wenn dies notwendig und möglich ist. Konvertierung immer logical → numeric → character → list:

```
> TRUE + 2  # convert TRUE to 1
> c("Hallo", sqrt(3))
```

Automatische Umwandlung

Umwandlung erzwingen mit `as.⟨neuerTyp⟩()`

```
> as.numeric(logicvec)
> as.logical(c(0, pi))
> as.character(c(1, 2, 4, TRUE))
> as.numeric(charvec)
Warning: NAs introduced by coercion
```

Sequenzen – `seq()`

Funktions-Schema: `seq(from, to, by, length)`

Wiederholungen – rep()

Funktions-Schema: `rep(times, each, length)`

Faktoren

Nominale / ordinale Merkmale werden in R als „Faktoren“ codiert

- Reihenfolge der Levels festlegen mit Argument `levels`:

```
> x <- factor(c("Saft", "Saft", "Limonade", "Saft", "Wasser"),
+             levels = c("Saft", "Wasser", "Limonade"))
> levels(x)
```

- Typumwandlung eines `character`-Vektors in Faktor mittels `as.factor()`:

```
> x <- as.factor(c("Apfel", "Birne", "Apfel",
+   "Traube", "Traube", "Kiwi"))
```

Rechnen mit Vektoren

- **Wichtig:** Die meisten Operationen von 2 Vektoren werden komponentenweise durchgeführt!!
- Addition, Subtraktion, Multiplikation, Division, ...

```
> x <- 1:4
> y <- c(4,10,2,0)
> x + y
> x * y
```

- Logische Vergleiche

```
> x < y
```

Recycling-Regel

R erlaubt auch Rechnen mit Vektoren unterschiedlicher Länge.

```
> x
> x + c(1, 2)
```

entspricht

```
> x + c(1, 2, 1, 2)
```

Fehlende Werte werden aus bestehenden „recycled“.

Zugriff auf Vektorelemente

4. Logischer Vektor

```
> x[(x > 5)]
> x[((x %% 2) == 0)]
```

5. Vektor von Zeichenketten

   Die Elemente eines Vektors kann man mit Namen versehen. Zugriff auf einzelen Elemente über diesen Namen möglich

```
> xn <- c(Wasser = 1, Saft = 2, Limonade = 3 )
> names(xn)
> xn["Saft"]
```

Grundlegende Operatoren und Funktionen

Aufruf der Hilfeseiten zu grundlegende Operatoren und Funktionen:

`?Arithmetic`	Operatoren für numerics
`?Logic`	Operatoren für logicals
`?log`	Logarithmus und Exponens
`?Trig`	Trigonnometrische Funktionen
`?Special`	Binomialkoeffizienten, Fakultät,...

Konstanten

Übersicht einiger Konstanten:

`pi`	Die Zahl π
`Inf`, `-Inf`	∞,$-\infty$
`NaN`	Not a Number: z.B. 0/0
`NA`	Not available: fehlende Werte
`NULL`	leere Menge
`letters`	Kleinbuchstaben von a bis z
`LETTERS`	Großbuchstaben von A bis Z

3 Matritzen

Matrizen – Erstellung

Eine Matrix in R ist ein Vektor mit Dimensions-Attribut!!!

Erzeugen einer Matrix:

```
> x <- matrix(nrow = 4, ncol = 2, byrow = TRUE,
+             data = c(1, 2, 3, 4, 5, 6, 7, 8))
> x
```

Was machen die Argumente in der oben angegebenen Funktion? `?matrix`

Matrizen – Eigenschaften

Abfrage bestimmter *Eigenschaften* der Matrix:

```
> dim(x)    # Dimension
> nrow(x)   # Anzahl Zeilen
> ncol(x)   # Anzahl Spalten
```

Matrix als ein spezieller Vektor mit Dimensionsattribut

```
> length(x) # Laenge des Datenvektors
```

Matrizen – cbind()

Spaltenweise Vektoren und Matrizen verbinden mit `cbind()`

```
> y <- c(12, 3, 4, 1)
> cbind(x, y)
> cbind(y, y)
```

Matrizen – rbind()

Zeilenweise Vektoren und Matrizen verbinden mit `rbind()`

```
> rbind(c(100, 0), x)
```

Matrixoperationen

- Vorbelegung aller Matrixelemente mit einer bestimmten Zahl

  ```
  > matrix(nrow = 4, ncol = 2, data = 1)
  ```

- Konstruktion einer Diagonalmatrix, hier der Einheitsmatrix der Dimension 2

  ```
  > diag(1, nrow = 2, ncol = 2)
  ```

Matrixoperationen

- Transponieren einer Matrix X (X'):

  ```
  > t(x)
  ```

- Matrixmultiplikation: Operator `%*%`

  ```
  > y <- matrix(1:6, nrow = 2)
  > z <- x %*% y
  > z
  ```

Matrixoperationen

- Kreuzprodukt ($X'X$) einer Matrix X: `crossprod()` (schneller als `t(x) %*% x`)

  ```
  > xtx <- crossprod(x)
  > xtx
  ```

Matrixoperationen

- Multiplikation einer Matrix mit einem Vektor Anzahl Elemente in Vektor $\leq$ Anzahl Elemente in Matrix:

  ```
  > 4 * x
  ```

- Verrechnung von 2 Matrizen benötigt gleich Dimension!

  ```
  > t(cbind(c(1,2), y)) + x
  ```

Matrixzugriff

- Matrixzugriff als Vektorzugriff je Dimension: `matrixobjekt[⟨Zeile⟩,⟨Spalte⟩]`

  ```
  > z[3,]
  > z[,2]
  > z[3:4, -2] # einfacher als z[3:4, c(1,3)]
  ```

  Prinzipiell jede Art des Vektorzugriffs möglich separat für jede Dimension

Übersicht

`matrix()`	Erstellen einer Matrix
`t()`	Transponieren einer Matrix
`%*%`	Matrixmultiplikation
`%o%`, `outer()`	Äußeres Produkt
`crossprod()`	Kreuzprodukt
`solve()`	Invertieren
`det()`	Determinante
`backsolve()`, `forwardsolve()`	Lösen von Gleichungssystemen
`eigen()`	Eigenwerte und Eigenvektoren
`nrow()`, `ncol()`	Anzahl Zeilen und Spalte
`dim()`	Dimension

4 Listen

Listen

Die Liste (**list**) kann Objekte unterschiedlicher Klassen als Elemente enthalten (im Gegensatz zu Vektoren).

```
> numvec <- c(2.54, 4.22, 2.99, 3.14, 3.44)
> charvec <- c("Statistische", "Software")
> mat <- matrix(seq(from = 2, by = 3, length = 12),
+   nrow = 3)
```

Erzeugen einer Liste:

```
> list1 <- list(numeric = numvec, charac-
ter = charvec,
+               matrix = mat)
> list1
```

Listen

Eine Liste kann auch Listen enthalten.

```
> logicvec <- c(TRUE, FALSE, FALSE, TRUE)
> list2 <- list(logicvec, list = list1)
> list2
```

Zugriff auf Listenelemente

Der Zugriff auf die Elemente einer Liste sollte über den `[[]]` Operator erfolgen.

```
> list1[[1]][2]
```

Auf Elemente mit Namen kann auch über `$` zugegriffen werden.

```
> list2$list$numeric[2]
```

Unterschied [] und [[]] bei Listen

- `x[i]` gibt eine Liste mit dem i-ten Objekt von x zurück.
- `x[[i]]` gibt das i-te Objekt von x zurück.
- `x[i]` gibt `NA` zurück, wenn i > `length(x)`.
- `x[[i]]` gibt Fehler aus, wenn i > `length(x)`.

Abfragen von Namen

Jedes Vektoren/Listen-Element *kann* einen Namen haben.

Abfragen über Funktion `names()`.

```
> names(list1)
> names(list2)
> names(1:8)
```

DataFrames

- DataFrames sind Listen, deren Elemente wiederum Vektoren gleicher Länge sind.
- Unterschied zur Matrix: Unterschiedliche Datentypen möglich!
- Die wichtigste Struktur zur Speicherung von Daten.
- Beobachtungen in den Zeilen und Variablen in den Spalten.
- Befehl zur Erzeugung eines DataFrames: `data.frame()`

DataFrame - Beispiel

Überblick verschaffen mit Funktionen `head()` und `str()`.

```
> head(mtcars)
```

DataFrame - Beispiel

```
> str(mtcars)
```

Zeilen-/Variablennamen

Zeilennamen mit `rownames()` und Spaltennamen mit `colnames()`.

```
> rownames(mtcars)
> colnames(mtcars) # oder 'names(mtcars)'
```

Achtung: die Zeilennamen sind keine eigene Variable!

Zugriff auf Elemente

Zugriff über die Zeilen-/ und Spaltenindices wie bei einer Matrix, also über `[]`.

```
> mtcars[1:4,]
```

Einzelne Variablen auch über `$` wie bei Listen.

```
> mtcars$cyl
```

Veränderung von Elementen

Elemente werden über Zuweisungen verändert.

Listenelement:

```
> v1 <- c(1, 2, 3)
> v2 <- c("a", "b", "c")
> l <- list("v1" = v1, "v2" = v2)
> print(l)

> l$v1[3] <- "y"
> print(l)
```

Veränderung von Elementen

Elemente werden über Zuweisungen verändert.

DataFrame-Element:

```
> v1 <- c(1, 2, 3)
> v2 <- c("a", "b", "c")
> df <- data.frame("v1" = v1, "v2" = v2)
> print(df)

> df$v1[3] <- "y"
> print(df)
```

Weitere wichtige Befehle

Subsetting:

```
> mtcars[mtcars$cyl == 4 & mtcars$hp >= 110, ]
```

Type conversion:

```
> class(VADeaths)
> df <- as.data.frame(VADeaths)
```

Weitere wichtige Befehle

Löschen von Spalten:

```
> disp <- mtcars$disp
> mtcars <- subset(mtcars, select = -disp)
> mtcars[1:3, ]
```

Hinzufügen von Spalten:

```
> mtcars$disp <- disp
> mtcars[1:3, ]
```

Weitere wichtige Befehle

Hinzufügen neuer Beobachtungen per `rbind()`:

```
> mtcars <- rbind(mtcars, mtcars[1, ])
> tail(mtcars)
```

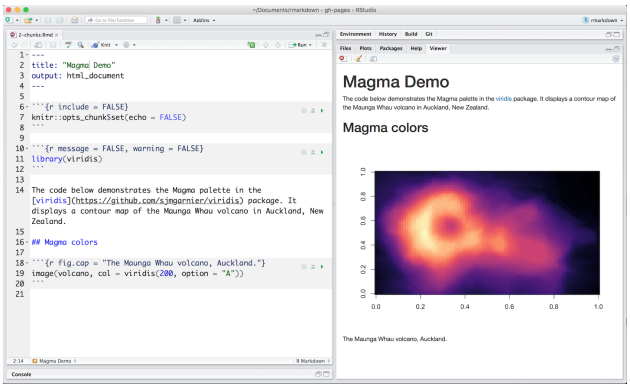
Hinzufügen von Spalten in Form eines DataFrames per `cbind()`:

```
> mtcars <- cbind(mtcars[1:3], mtcars[4:6])
> mtcars[1:6, ]
```

5 RMarkdown

RMarkdown

1. Code in Code Chunks
2. Text mit pandoc/latex



Optionen für Code Chunks

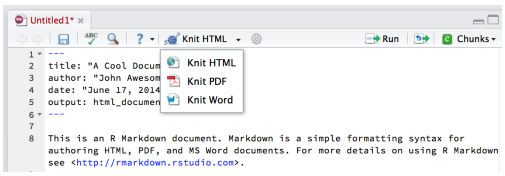
- Separaten vs. inline Code
  - Separater Code indiziert durch: “‘{r, Optionen } Code hier“‘
  - Inline Code indiziert durch: ‘r Code hier ‘
- Optionen für Code Chunks mit binären Werten (TRUE vs. FALSE)
  - `include`: Soll die Ausgabe und der Code des Chunks angezeigt werden?
  - `echo`: Soll der Code angezeigt werden?
  - `warning/message`: Sollen Warnungen/Benachrichtigungen die durch die Ausführung des Codes entstehen angezeigt werden?
  - `eval`: Soll der Code ausgeführt werden?
- Setup Chunk: Erster Code Chunk im Dokument um globale Einstellungen zu definieren (hat meist den Namen setup und `include`$=FALSE$)

Text mit Markdown

- *kursiv* = *kursiv*
- **fett** = **fett**
- `code` = `code`
- β = β (Mathematische Notation)
- # Header1 = große Überschrift
- ## Header2 = kleinere Überschrift
- * Liste, Subliste, Subsubliste (für Aufzählungen, leere Zeilen zur Abtrennung sind wichtig!)

https://github.com/rstudio/cheatsheets/raw/master/rmarkdown-2.0.pdf

Ausführen der R Markdown Datei



- Datei muss zu einer PDF Datei *gestrickt* werden

6 Dateien

Pfadangaben

Jede Datei auf dem Rechner liegt an einem Ort in der Verzeichnis-Baum-Struktur, identifiziert über den *Pfad*.

Jede Verzeichnisebene wird über Pfadtrenner verbunden, bei MS Windows `\`, bei Mac und *nix-Systemen `/`

```
> # Geht auch unter Windows
> (pfad <- "Der/Pfad/zu/meiner/Datei")
> # Nur unter Windows
> (pfad_ms <- "Der\\Pfad\\zu\\meiner\\Datei")
> # Pfadtrenner automatisch eingefuegt
> (pfad_r <- file.path("Der", "Pfad", "zur", "Datei"))
```

Relative vs. Absolute Pfadangabe

Bei der absoluten Angabe muss man immer im OS-Wurzelverzeichnis starten (Laufwerksbuchstabe bei Windows)

```
> "C:/Der/absoulte/Pfad/zu/meiner/Datei"
```

Bei der relativen Angabe nimmt R das aktuelle Arbeitsverzeichnis und geht von dort aus weiter

```
> "Der/relative/Pfad/zu/meiner/Datei"
> # entspricht
> # file.path(getwd(),"Der/relative/Pfad/zu/meiner/Datei")
```

Arbeitsverzeichnis

Verzeichnis, in dem man mit R arbeitet:

- Speicherort für History und Workspace
- Wurzelverzeichnis für relative Pfadangaben

```
> getwd() # Abfragen
> setwd("Pfad/zum/arbeits/verzeichnis")
```

Einlesen in R - Textformat

DIE grundlegende Funktion in R zum Einlesen von Textformaten ist `read.table()`

Funktionen zum Einlesen von bestimmten Textformaten, rufen meist nur die Funktion `read.table()` mit anderen vorgegeben Argumenten auf, zum Beispiel `read.csv2()`

Hilfe: `?read.table`

Die Funktion liefert nach dem Einlesen ein Objekt vom Typ `data.frame` zurück

Einlesen in R - Textformat

Standardmäßig wandelt R beim Einlesen die Spalten in geeignete Formate um:

- Spalten mit Zahlen werden als `numeric` eingelesen
- Spalten mit Text werden als `factor` eingelesen

Wichtige Argumente für `read.table()`:

`header`: Erste Zeile enthält Variablennamen
`sep`: Trennzeichen zwischen den Spalten
`dec`: Dezimaltrennzeichen (1.3 vs. 1,3)
`as.is`: Keine automatische Umwandlung
`colClasses`: Typ der Spalten vorgeben

Einlesen in R - Textformat

Insbesondere zum Einlesen von sehr großen Datensätzen:

Funktionsfamilie aus Paket `readr`:

`read_delim()`	$\hat{=}$	`read.table()`
`read_csv()`	$\hat{=}$	`read.csv()`
`read_csv2()`	$\hat{=}$	`read.csv2()`
`read_fwf()`	$\hat{=}$	`read.fwf()`
	$\vdots$	

Argumente heißen etwas anders, bieten aber selbe Funktionalität.

Einlesen in R - Binärformat

Daten im Excel-Format *xls(x)* :

- Für xlsx Dateien: `read.xlsx()` aus Paket `xlsx`
- Für xls Dateien: `read.xls()` aus Paket `gdata`
- Für xls und xlsx Dateien: `read_excel()` aus Paket `readxl` (Braucht kein Java)

Daten in SPSS/Stata/SAS-Format *sav* :

- Einlesen mit Funktion `read.spss()` aus Paket `foreign`
- Siehe Dokumentation des Pakets für weitere Import-Codes

TIPP: Einlesen aus Binärformaten vermeiden!

Speichern von Dateien in R

- Alle einlesbaren Dateiformate können auch ausgegeben werden (`write.table`, `write.csv`, ...)
- Zusätzlich können R-spezifische Dateien gespeichert werden
  - `save`: `save(R_obj,file = "directory/filename.RData")` für das Speichern von einzelnen Objekten im Arbeitsbereich

- save.image: save.image(file = "directory/filename.RData") für das Speichern des gesammten Arbeitsbereichs (working directory)
- load: Befehl um R Objekte in den Arbeitsbereich zu laden

7 Ausgabe

Ausgabe – print()

Ausgabe des Objekts in der Konsole

Rückgabewert: Das Objekt, mit dem die Funktion aufgerufen wurde

Argument digits: Angabe der relevanten Stellen.

```
> print(sqrt(2))
> print(sqrt(2), digits = 2)
> print(sqrt(2), digits = 4)
> print(sqrt(2) + 100, digits = 2)
> print(sqrt(2) + 100, digits = 4)
```

Ausgabe – cat()

Umwandlung und Verknüfpung aller übergebenen Objekte in eine Zeichenkette und Ausgabe auf die Konsole oder in eine Datei

Rückgabewert: immer NULL!

Argument sep: Zeichenkette zwischen den einzelnen Elemente Argument file: Name einer Ausgabe-Datei (Falls leer: Konsole)

```
> v <- 7
> cat("Quadrat von", v, ":", v^2)
> cat("Quadrat von ", v, ": ", v^2, sep = "")
```

Operationen mit Zeichenketten – paste()

(Vektorwertiges) Zusammenfügen von Zeichenketten

Rückgabewert: Vektor von zusammengefügten Zeichenketten

Argument sep: wie bei cat(). Argument collapse: Vektor von Zeichenketten als eine Einzige

```
> pa <- paste("Aufgabe", 1:5, sep = "_")
> pa
> pa[1]

> pac <- paste(pa, collapse = "?")
> pac
```

paste0() ist eine Abkürzung für paste(, sep = "")

Operationen mit Zeichenketten – strsplit()

(Vektorwertige) Zerlegung einer Zeichenkette in Teile

Rückgabewert: **Liste** von Vektoren der getrennten Zeichenketten

Argument split: Trennzeichen

```
> x <- "Die Syntax#!von str-
split#!findet man!#wie immer!#in der Hilfe"
> strsplit(x, split = "#")
> spl <- strsplit(x, split = "!#")
> spl
```

Operationen mit Zeichenketten – strsplit()

Achtung:

Interpretation der "Split"-Zeichenkette als sog. *regulärer Ausdruck*!

```
> strsplit(pac, "?")
[[1]]
 [1] "A" "u" "f" "g" "a" "b" "e" "_" "1" "?" "A"
[12] "u" "f" "g" "a" "b" "e" "_" "2" "?" "A" "u"
[23] "f" "g" "a" "b" "e" "_" "3" "?" "A" "u" "f"
[34] "g" "a" "b" "e" "_" "4" "?" "A" "u" "f" "g"
[45] "a" "b" "e" "_" "5"
```

Verhindern mit fixed = TRUE

```
> strsplit(pac, "?", fixed = TRUE)
[[1]]
[1] "Aufgabe_1" "Aufgabe_2" "Aufgabe_3"
[4] "Aufgabe_4" "Aufgabe_5"
```

Zeichen mit spezieller Bedeutung bei regulären Ausdrücken: ".", "?", "^", "$", "*", "+", Klammern aller Art

Suchen und Ersetzen in Zeichenketten

Nur Muster-Suche: grep() und grepl()

```
> c(x,pac)

> grep("#!", c(x, pac) )# "#!" ist im 1-ten Ele-
ment, also in x
> grepl("#!", c(x, pac) )
```

Muster-Suche und Ersetzung: sub() und gsub()

```
> sub("#!", " ", x) # Nur das erste Vorkommen
> gsub("#!", " ", x) # Alle Vorkommen
```

Suchen und Ersetzen in Zeichenketten

Interpretation des Suchmusters als regulärer Ausdruck: Verhindern wie bei `strsplit`

```
> gsub("?", " ", pac)
> gsub("?", " ", pac, fixed = TRUE)
```

Beispiel für Verwendung eines regulären Ausdrucks:

```
> z <- "           Viele Leerzeichen am Anfang"
> z
> gsub("^\\s*", "", z) # Loeschen der Leerzeichen am Anfang
```

Übersicht

`cat()`	für Ausgabe in Konsole/Datei
`format()`	Formatierung von Zahlen
`formatC()`	Formatierung im C Stil
`grep()`, `grepl()`	Suchen von Zeichenfolgen
`nchar()`	Anzahl Zeichen in String
`paste()`, `paste0()`	Zusammensetzen von Strings
`strsplit()`	Zerlegen von Strings
`sub()`, `gsub()`	Ersetzen von (Sub-)Strings
`toupper()`, `tolower()`	Groß-/Kleinbuchstaben
`\t`, `\n`	Tab-Einrückung, Zeilenumbruch

8 Datenmanipulation

Datenmanipulation

Beispiele:

- Kombination oder Splitting von Spalten.
- Anwendung von Funktionen auf alle Reihen oder Spalten.
- Löschen oder Änderung von fehlerhaften Beobachtungen.
- Transformieren in tidy-Format.
- Intelligentes Subsetting.

Basispaket oftmals ausreichend, aber die Pakete des `tidyverse` wie `tidyr` und `dplyr` stellen zahlreiche Methoden mit einfacher Syntax zur Verfügung.

- Für eigene Software: So wenig Dependenzen auf andere Pakete wie möglich! → verwende Basispaket.
- Für explorative Analysen: So schnell und einfach wie möglich, da Code i.d.R. nicht wiederverwendet wird. → `tidyverse`

apply (Basispaket)

`apply(data, MARGIN, FUN)`

- Wende Funktion auf alle Reihen bzw. Spalten des Datensatzes an.
- MARGIN = 1 für Reihen, 2 für Spalten.

```
> apply(mtcars, MARGIN = 2, FUN = function(col) mean(col))
```

tibble

- Erbt von DataFrame.
- Beseitigt viele Nachteile von DataFrames: Keine Konvertierung von Strings zu Faktorvariablen, keine Namensänderung von Variablen, etc.
- Informativere Print-Funktion.

```
> library(dplyr)
> tbl <- tibble(mtcars)
> head(tbl)
> class(tbl)
```

Der Pipe-Operator %>%

- Verkette den Datensatz über %>% mit Funktionsaufrufen. In den Funktionsaufrufen wird der Datensatz nicht mehr als Argument übergeben.
- Beispiel ohne Pipe-Operator:

```
> apply(subset(head(mtcars, 30), select = c(mpg, cyl, hp)),
+       MARGIN = 2, mean)
```

- Beispiel mit Pipe-Operator:

```
> library(magrittr)
> mtcars %>%
+   head(30) %>%
+   subset(select = c(mpg, cyl, hp)) %>%
+   apply(MARGIN = 2, mean)
```

Einführung in `dplyr`

- Subsetting: `filter`

```
> library(dplyr)
> mtcars %>%
+   filter(hp >= 250)
```

- Selektieren von Spalten: `select`

```
> mtcars %>%
+   select(mpg, cyl, disp) %>%
+   head(5)
```

Einführung in dplyr

- Verändern und Hinzufügen von Spalten: mutate

```
> mtcars %>%
+   select(mpg, cyl, disp) %>%
+   mutate(mpg = (mpg -
 mean(mpg)) / sd(mpg)) %>%
+   head(2)
```

- Funktionsanwendung auf Spalten: summarise bzw. summarise_all

```
> mtcars %>%
+   summarise(mean(mpg))

> mtcars %>%
+   summarise_all(.funs = mean)
```

Einführung in dplyr

- Gruppieren von Beobachtungen: group_by

```
> iris %>%
+   group_by(Species) %>%
+   summarize_all(mean)
```

- Anordnung von Reihen: arrange

```
> mtcars %>%
+   arrange(desc(hp)) %>%
+   head(5)
```

Wide-Format

- Beobachtungen in Reihen
- Variablen in Spalten
- Messwiederholungen → eigene Spalte

Long-Format

- Variablen-Wert-Paare
- 1 Spalte für alle Variablen
- 1 Spalte für korrespondierende Werte
- Messwiederholungen → eigene Reihe

Long to Wide - spread

```
spread(data, key, value, ...)
```

```
> data_wide <- data_long %>%
+   spread(key = variable, value = value)
> data_wide
```

Wide to Long - gather

```
gather(data, key, value, ...)
```

```
> data_long <- gather(data_wide, key = vari-
able, value = value, -id)
> data_long
```

Trennen von Spalten - separate

```
separate(data, col, into, sep, ...)
```

```
> data_long <- data_long %>%
+   separate(variable, in-
to = c("variable", "time"))
> data_long
```

Zusammenfügen von Spalten - unite

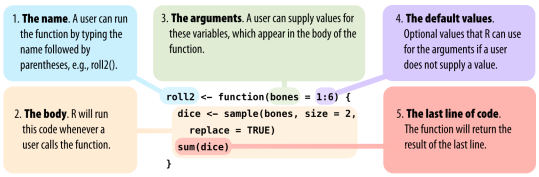
```
unite(data, col, ..., sep, remove)
```

```
> data_long %>%
+   unite(col = "variable", "vari-
able", "time", sep = ".", remove = TRUE)
```

9 Funktionen

Bestandteile einer Funktion

1. Name der Funktion.
2. Parameter der Funktion: Was wird der Funktion übergeben mit welchen Standardwerten?
3. Rumpf der Funktion: Inhalt der Funktion und Rückgabe.



Rückgabewerte einer Funktion

- Was soll die Funktion zurückgeben? ⇒ return
- Falls nichts zurückgegeben wird ⇒return(NULL)!
- Bei mehrdimensionalen Rückgabewerten bietet sich eine Liste an.

```
> getMeanSD <- function(vec) {
+   vec_mean <- mean(vec)
+   vec_sd <- sd(vec)
+   res <- list("mean" = vec_mean, "sd" = vec_sd)
+   return(res)
+ }
>
> v <- c(10, -50, 100)
> getMeanSD(v)
```

`if-else`-Anweisung

- Häufig möchten wir eine Anweisung nur dann ausführen, wenn eine bestimmte Bedingung erfüllt ist, d.h. Wenn ⇒ Dann | Sonst
- `if (Bedingung) { Dann-Anweisung } else { Sonst-Anweisung }`

```
> rand <- rnorm(1000, 0, 1)
> head(rand)
> if (mean(rand) <= 0) {
+   print(-1)
+ } else {
+   print(1)
+ }
```

Schleifen

- Viele Funktionen in R sind vektorisiert und beruhen auf Schleifen (z.B. die der vorherigen Folie).
- Schleifen wiederholen einen Anweisungsblock, solange eine Schleifenbedingung erfüllt ist.
- Beispiel Mittelwertberechnung: Wir müssen über alle Elemente eines Vektors iterieren, um die kumulative Summe zu berechnen.
- Es gibt 2 Varianten, um Schleifen zu konstruieren: `for` und `while`. Beide Varianten können immer ineinander überführt werden.

`for`-Schleifen

```
  for (i in Vektor) {
Anweisungsblock }
```

```
> cumsum <- 0
> for (i in 1:length(rand)) {
+   cumsum <- cumsum + rand[i]
+ }
> print(cumsum)
> 1/length(rand) * cumsum
```

`while`-Schleifen

```
  while (Bedingung) {
Anweisungsblock }
```

```
> cumsum <- 0
> i <- 1
> while (i <= length(rand)) {
+   cumsum <- cumsum + rand[i]
+   i <- i + 1
+ }
> print(cumsum)
> 1/length(rand) * cumsum
```

Kombination aus Schleife und `if-else`

- Viele Operationen, die wir alltäglich durchführen beruhen auf einer Kombination von Schleife und `if-else`-Abfrage.
- Beispiel: Subsetting

```
> subs1 <- iris$Species == "setosa"
```

```
> subs2 <- vector(length = nrow(iris))
> for (i in 1:nrow(iris)) {
+   if (iris[i, ]$Species == "setosa") {
+     subs2[i] <- TRUE
+   } else {
+     subs2[i] <- FALSE
+   }
+ }
>
> all(subs1 == subs2)
```

Schleifen in R

- Die `apply`-Funktionen bieten eine performantere R-Implementierung von Schleifen als die herkömmlichen Varianten mit `for` und `while`.
- `apply(dataframe, MARGIN, FUN)`
- `lapply(vector, FUN)`
- `vapply(vector, FUN, FUN.VALUE)`
- `sapply(vector, FUN)`

```
> apply(mtcars, MARGIN = 2, FUN = mean)
> lapply(mtcars$mpg, FUN = function(i) i^2)
> vapply(mtcars$mpg, FUN = func-
tion(i) i^2, FUN.VALUE = numeric(1))
> sapply(mtcars$mpg, FUN = function(i) i^2)
```

10 Plots: R Base

Grafikausgabe

- Die Grafikausgabe erfolgt in ein sogenanntes Gerät (Device).
- Die öffnende Funktion bestimmt das Gerät.
- Standardmäßig zur Verfügung stehen u.a.: `bmp()`, `jpeg()`, `pdf()`, `png()`, `postscript()`, `x11()` für Bildschirmfenster.
- Tatsächliche Ausgabe bei Datei-Geräten erst nach Schließen.

Die `plot()` Funktion

`plot()` ist die wichtigste *High-level* Funktion für Grafiken mit dem Basispaket.

Oft einfachste Variante um Basis-Grafiken zu erstellen.

Je nach Datentyp der übergebenen Objekte liefert die Funktion eine andere Grafik:

Die `plot()` Funktion ist generisch.

Wichtige High-Level Grafikfunktionen

Funktion	Datentyp(en)	Beschreibung
`plot()`	numeric	Scatterplot
`plot()`, `pairs()`	data.frame	Sctt Matrix
`sunflowerplot()`	numeric x 2	Sctt (diskret)
`plot()`	factor/1-dim. tbl	Barplot
`barplot()`	numeric	Barplot
`barplot()`	matrix	Barplot
`hist()`	numeric	Histogramm
`boxplot()`	(list of) numeric	(bedingte) Bxplt
`plot()`	factor, numeric	bedingte Bxplt
`plot()`	factor, factor	Spineplot
`plot()`	2-dim. table	Mosaic plot
`mosaicplot()`	n-dim. table	Mosaic plot

Argumente für Grafikfunktionen

Anpassung von Aussehen über Argumente der Grafikfunktion, z.B. Titel, Achsenbeschriftung, Farbe, ...

Argument	Beschreibung
`main`	Haupttitel der Grafik.
`xlab`, `ylab`	Titel der X-Achse bzw Y-Achse.
`xlim`, `ylim`	Vektor mit Minimum/Maximum für Werte. in der Plot-Region in X bzw. Y Richtung.
`cex`	Größe des Elements (beispielsweise von Punkten).
`cex.main`, `cex.axis`, `cex.lab`	Größe von Titel, Achsenbeschriftung, und /-titeln relativ zu `cex`.
`col`	Farbe der Objekte in der Plot-Region.
`bg`, `fill`	Hintergrund-/ bzw. Füllfarbe von Objekten.
`las`	Rotation der horizontalen Achsenbeschriftungen.
`lty`, `lwd`	Linientyp, Linienbreite.
`pch`	Zeichenart für Punkte.

Das `type` Argument in `plot()`

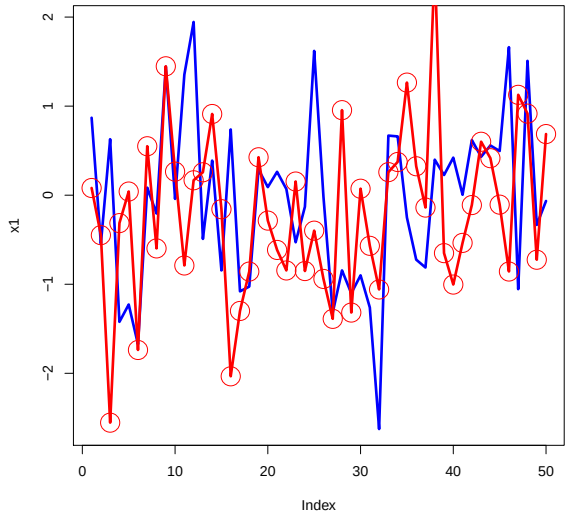
```
> y <- rnorm(20)
> plot(y, type = "p")
> plot(y, type = "l")
> plot(y, type = "b")
> plot(y, type = "h")
```

- `type = "p"`: Streudiagramm
- `type = "l"`: Liniendiagramm
- `type = "b"`: Streu-/und Liniendiagramm
- `type = "h"`: Stabdiagramm

Weitere Elemente hinzufügen: *low-level*

- Zeichnen des Plots inklusive Generierung des Koordinatensystems mit `plot()` (high-level).
- Hinzufügen einer Linie mit `lines()` (low-level).
- Hinzufügen von Punkten mit `points()` (low-level).

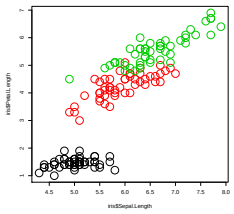
```
> x1 = rnorm(50, 0, 1)
> x2 = rnorm(50, 0, 1)
> plot(x1, type = "l", col = "blue",
+      lwd = 3)
> lines(x2, col = "red", lwd = 3)
> points(x2, col = "red", cex = 3)
```



Gruppieren von Beobachtungen

- Gruppierungen von Beobachtungen (beispielsweise je Ausprägung einer Faktorvariable) können durch unterschiedliche Farben visualisiert werden.
- Wir können R einen Farbcode generieren lassen, indem wir dem Farbparameter eine Faktorvariable übergeben.

```
> plot(iris$Sepal.Length, iris$Petal.Length,
+    cex = 3, col = iris$Species)
```



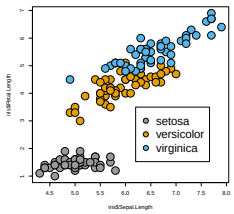
Gruppieren von Beobachtungen - Eigener Farbcode

- Wir können außerdem einen eigenen Farbcode als benannten Vektor spezifizieren.
- Anschließend gleichen wir die Ausprägungen unserer Beobachtungen per `match` mit dem Farbcode ab.
- Der Vektor mit Farbangabe je Beobachtung wird anschließend an die `plot`-Funktion übergeben.

Legenden

- Hinzufügen von Legenden über `legend(x, y, legend, pch, col, pt.bg, ...)`.
- In R Markdown müssen zusammenhängende Befehle für einen Plot, z.b. `plot()` und `legend()` innerhalb geschweifter Klammern gesetzt werden, d.h. `{plot(...) legend(...)}`.

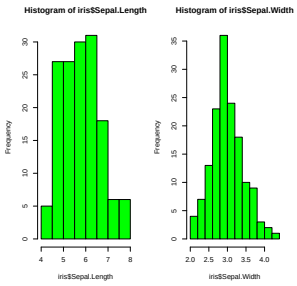
```
> legend(6.2, 3.5, names(color_code), pch = 21,
+    cex = 2, col = "black", pt.bg = color_code)
```



Kombinierte Grafiken

Über `par(mfrow(Reihen, Spalten))` können wir Grafiken auf einem Gitter kombinieren.

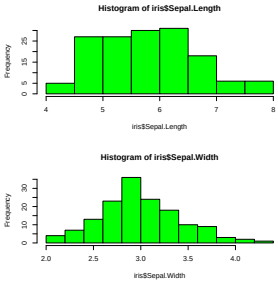
```
> par(mfrow = c(1, 2))
> hist(iris$Sepal.Length, col = "green")
> hist(iris$Sepal.Width, col = "green")
```



Kombinierte Grafiken

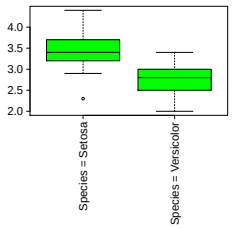
Über `par(mfrow(Reihen, Spalten))` können wir Grafiken auf einem Gitter kombinieren.

```
> par(mfrow = c(2, 1))
> hist(iris$Sepal.Length, col = "green")
> hist(iris$Sepal.Width, col = "green")
```



Ränder (Margins)

```
> par(mar = c(18, 5, 1, 1))
>
> boxplot(
+   iris$Sepal.Width[iris$Species=="setosa"],
+   iris$Sepal.Width[iris$Species=="versicolor"],
+   names = c("Species = Setosa",
+     "Species = Versicolor"),
+   cex.axis = 2,
+   las = 2,
+   col = "green")
```



11 Plots: ggplot

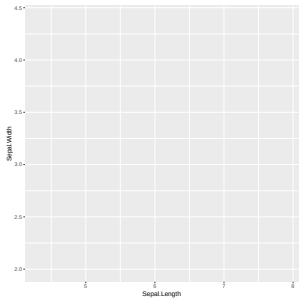
ggplot-Objekt

- Base-Plots werden bei der Erzeugung in ein Device ausgegeben. Anschließend können nur noch zusätzliche Elemente in das gleiche Device geplottet werden. Eine grundlegende Modifikation des Plots ist nicht möglich, da kein Objekt existiert.
- ggplots ermöglichen im Gegensatz zu Base-Grafiken die Erstellung von Plot-Objekten, die gespeichert und modifiziert werden können.

Plot-Objekt

Das ggplot-Objekt wird über `ggplot()` erstellt und kann im Nachhinein modifiziert werden.

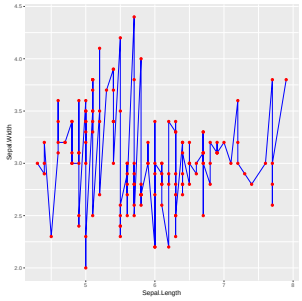
```
> library(ggplot2)
> p <- ggplot(data = iris,
+   aes(x = Sepal.Length, y = Sepal.Width))
> p
```



Erweiterung über Schichten (Layer)

Oder beide Layer:

```
> p + geom_line(col = "blue") + ge-
om_point(col = "red")
```

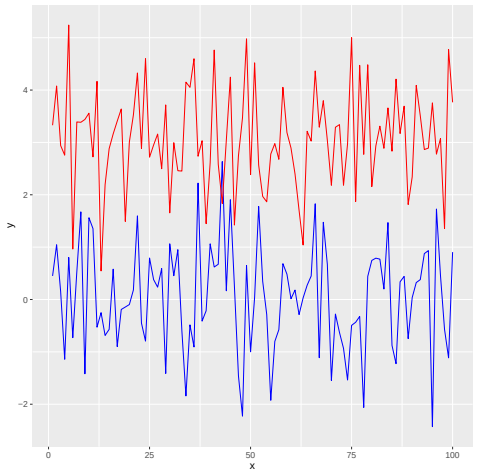


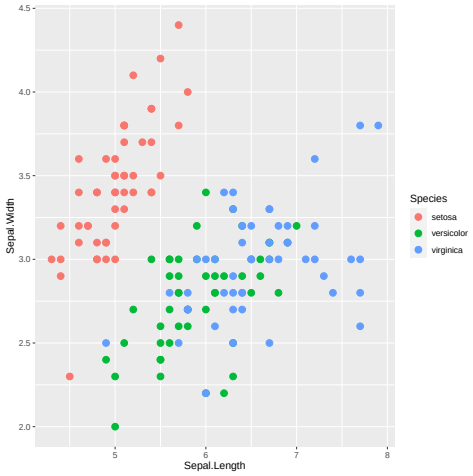
Globale und Layer-spezifische Settings

Global verwendete Settings können im Plot-Objekt gespeichert werden. Layer-spezifische Settings werden dem Layer übergeben.

Wir können auch mehrere Layer mit unterschiedlichen Daten und Mappings verwenden.

```
> ggplot() +
+   geom_line(data = df1, aes(x = x, y = y),
+             col = "blue") +
+   geom_line(data = df2, aes(x = x, y = z),
+             col = "red")
```

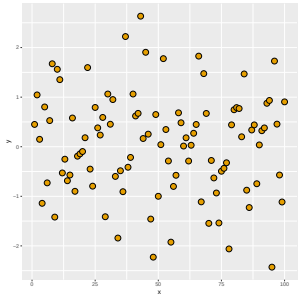




Layer-Settings

Typische Layer-Settings sind die Größe, Form und Farbe der Layer-Elemente, z.B. `shape` (Form), `fill` (Füllfarbe), `color` (Randfarbe) und `stroke` (Randbreite) von `geom_point()`.

```
> ggplot() + geom_point(data = df1, aes(x = x, y = y),
+     shape = 21, fill = "#E69F00",
+     color = "black", size = 4, stroke = 1)
```



Aesthetic Mappings (aes)

Aesthetic Mappings weisen einem Plot-Parameter eine Datenvariable zu. Die Koordinaten (x, y) sind wichtige Mappings.

Zusätzlich können wir beispielsweise dem Farb-Parameter eine Datenvariable übergeben.

```
> ggplot(data = iris,
+        aes(x = Sepal.Length,
+            y = Sepal.Width,
+            col = Species)) +
+   geom_point(size = 3)
```

Group-/ und Color-Mapping

Gruppierte Beobachtungen, z.B. Messwiederholungen können im Long-Format über das Group-Mapping gruppiert und über das Color-Mapping farblich unterschieden werden.

```
> stocks <- as.data.frame(EuStockMarkets)
> stocks$time <- as.numeric(time(EuStockMarkets))
> str(stocks)

> library(tidyr)
> stocks <- gather(stocks, key = "index", val-
ue = "value", -time)
> str(stocks)
```

Manuell konfiguriertes Color-Mapping

Über `scale_color_manual()` bzw. `scale_fill_manual()` können wir manuell konfigurierte Color-Mappings je Ausprägung einer Variable übergeben. Color bzw. Fill wird je nach Shape der Plot-Elemente benutzt. Die Funktion erstellt automatisch eine Legende, der wir einen Namen übergeben können.

```
> color_code <- c("DAX" = "#E69F00",
+                 "CAC" = "#56B4E9",
+                 "FTSE" = "#009E73",
+                 "SMI" = "#0072B2")
>
> p <- ggplot(data = stocks,
+        aes(x = time, y = value,
+            group = index, color = index)) +
+   geom_line(size = 1) +
+   theme(legend.position = "top") +
```

```
+   scale_color_manual("Index",
+     values = color_code)
>
> p
```

Achsentitel und Überschriften

Achsentitel werden über `labs()` und Überschriften über `ggtitle()` spezifiziert.

```
> p + labs(x = "Year", y = "Index Value") +
+   ggtitle("Zeitliche Entwicklungen von wichtigen Aktienindices")
```

Achsenbeschriftungen

Die Achsen selbst können über `scale_x_continuous()` und `scale_y_continuous()` für stetige Variablen bzw. `scale_x_discrete()` und `scale_y_discrete()` für diskrete Variablen konfiguriert werden.

```
> p +
+   scale_x_continuous(
+     breaks = c(1992, 1995, 1998),
+     labels = c("Year = 1992", "Year = 1995",
+       "Year = 1998")) +
+   scale_y_continuous(
+     breaks = c(3000, 5000, 7000),
+     labels = c("Index = 3000",
+       "Index = 5000", "Index = 7000"))
```

Themes

ggplots können über eine Vielzahl an vorgefertigten Themes optisch verändert werden. Zusätzliche Themes sind unter anderem über das `ggthemes`-Paket verfügbar.

Auszug an verfügbaren Themes:

- theme_bw()
- theme_base()
- theme_classic()
- theme_minimal()
- theme_economist()
- theme_economist_white()
- theme_gdocs()
- theme_wsj()

Themes

Über das `theme()`-Layer können wir das Theme manuell konfigurieren.

```
> p + theme(axis.text.x = element_text(angle = 90, size = 14),
+         axis.text.y = element_text(size = 14),
+         axis.title.x = element_text(size = 16),
+         axis.title.y = element_text(size = 16),
+         legend.text = element_text(size = 20),
+         legend.title = element_text(size = 16),
+         legend.key.size = unit(2, "cm"),
+         legend.position = "top")
```

Margins

Margins werden in ggplot2 über das `plot.margin`-Argument des `theme`-Layers spezifiziert. Die Reihenfolge ist unterschiedlich zur Konfiguration für Base-Plots: top, right, bottom left (Eselsbrücke: TRouBLe).

```
> p + theme(plot.margin = unit(c(1, 2, 1, 1), "cm"))
```

Kombinationen von Plots

Über die `grid.arrange()`-Funktion des `gridExtra`-Pakets können wir ggplots auf einem Gitter (Grid) kombinieren.

```
> library(gridExtra)
>
> p1 <- ggplot(data = iris,
+              aes(x = Sepal.Length)) +
+   geom_density(fill = "#E69F00", lwd = 2)
>
> p2 <- ggplot(data = iris,
+              aes(x = Sepal.Width)) +
+   geom_density(fill = "#56B4E9", lwd = 2)
>
> p3 <- ggplot(data = iris,
+              aes(x = Petal.Length)) +
+   geom_density(fill = "#009E73", lwd = 2)
>
> p4 <- ggplot(data = iris,
+              aes(x = Petal.Width)) +
+   geom_density(fill = "#0072B2", lwd = 2)
>
> p_combined <- grid.arrange(p1, p2, p3, p4,
+   nrow = 2, ncol = 2)
```

Speichern von Plot-Objekten

ggplot-Objekte können über `ggsave()` abgespeichert werden.

```
> ggsave(p_combined, file = "p_combined", device = "jpg")
```