


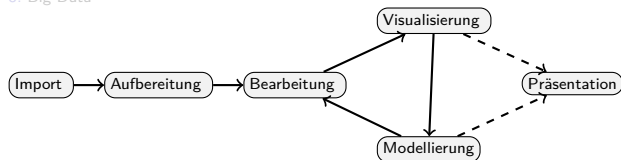
# BW09: Datenanalyse

Dr. Daniel Brunner

## Kapitel 1-8: Softwaregestützte Datenanalyse

# Inhalte

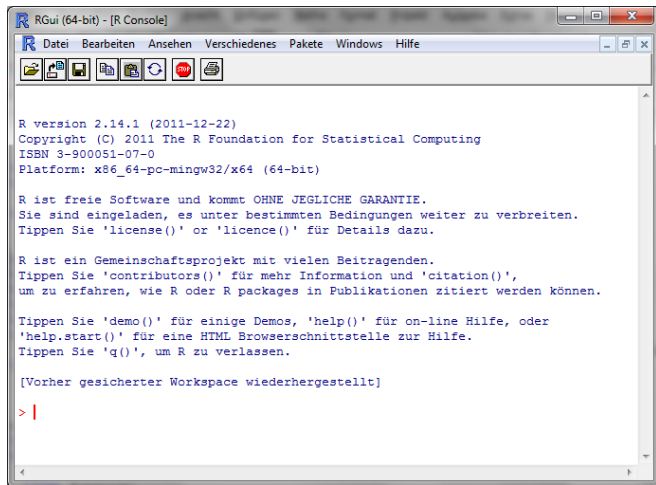
1. Grundlagen der Software 
  - Einführung in R
  - Skripte
  - Working Directory
  - Pakete
  - Coding Style
  - R-Markdown
  - Tips & Tricks
2. Datenimport
3. Grundlagen der Datenaufbereitung und -bearbeitung
4. Fortgeschrittene Datenaufbereitung und -bearbeitung
5. Grafische Auswertung
6. Unstrukturierte Daten
7. Performance
8. Big Data



# R und RStudio

- ▶ Warum R?
  - ▶ umfangreiche Programmiersprache und Statistik-Software
  - ▶ kostenlose Nutzung
  - ▶ beliebteste Software zur Datenanalyse
- ▶ R ist eine freie Software (GNU Lizenz)
  - ▶ Download unter [www.r-project.org](http://www.r-project.org)
- ▶ dort können Quellcode und einfach zu installierende Programme für unterschiedliche Betriebssysteme heruntergeladen werden
- ▶ RStudio ist eine integrierte Entwicklungsumgebung, die den Umgang mit R erheblich vereinfacht
  - ▶ Download unter [www.rstudio.com](http://www.rstudio.com)
- ▶ eine Übersicht über wichtige R Befehle ist als Handout bereitgestellt

# Screenshot R



The screenshot shows the RGui (64-bit) - [R Console] window. The title bar includes the R logo and the text "RGui (64-bit) - [R Console]". The menu bar contains "Datei", "Bearbeiten", "Ansehen", "Verschiedenes", "Pakete", "Windows", and "Hilfe". Below the menu bar is a toolbar with icons for file operations (New, Open, Save, Print, Copy, Paste, Undo, Redo, Stop, Run). The main console area displays the following text:

```
R version 2.14.1 (2011-12-22)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-pc-mingw32/x64 (64-bit)

R ist freie Software und kommt OHNE JEGLICHE GARANTIE.
Sie sind eingeladen, es unter bestimmten Bedingungen weiter zu verbreiten.
Tippen Sie 'license()' or 'licence()' für Details dazu.

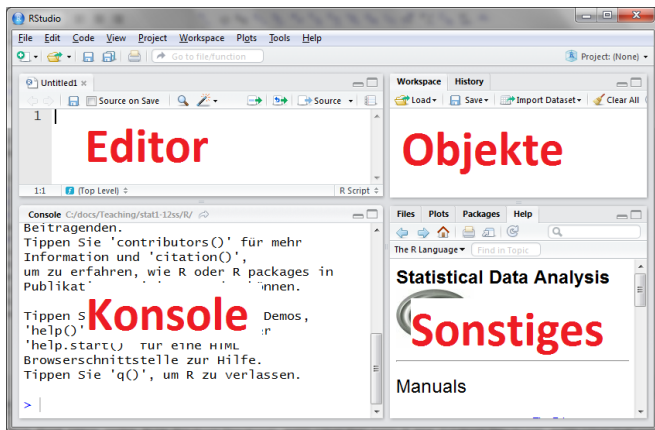
R ist ein Gemeinschaftsprojekt mit vielen Beitragenden.
Tippen Sie 'contributors()' für mehr Information und 'citation()',
um zu erfahren, wie R oder R packages in Publikationen zitiert werden können.

Tippen Sie 'demo()' für einige Demos, 'help()' für on-line Hilfe, oder
'help.start()' für eine HTML Browserschnittstelle zur Hilfe.
Tippen Sie 'q()', um R zu verlassen.

[Vorher gesicherter Workspace wiederhergestellt]

> |
```

# Screenshot R-Studio



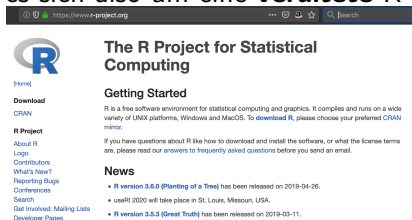
# R Version

- ▶ **Wichtig:** für diesen Kurs sind aktuelle Versionen von R und R Studio zwingend erforderlich
  - ▶ aktuelle Version von R unter [www.r-project.org](http://www.r-project.org)
  - ▶ aktuelle Version von R Studio unter [www.rstudio.com](http://www.rstudio.com)
  - ▶ ältere Versionen (z.B. aus der BS01 oder BS02) sind zu alt!
- ▶ der Befehl `R.Version()$version.string` zeigt die installierte R Version

## R-Output

```
> R.Version()$version.string  
[1] "R version 3.5.1 (2018-07-02)"
```


- ▶ dabei handelt es sich also um eine **veraltete** R Version!



The screenshot shows the R Project website. The browser address bar displays <https://www.r-project.org>. The page features the R logo and the title "The R Project for Statistical Computing". Under the "Getting Started" section, it states: "R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To [download R](#), please choose your preferred CRAN mirror." Below this, it says: "If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email." The "News" section lists recent releases: "R version 3.6.0 (Planting of a Tree) has been released on 2019-04-26.", "useR! 2020 will take place in St. Louis, Missouri, USA.", and "R version 3.5.3 (Great Truth) has been released on 2019-03-11."

# Einfache Beispiele

- ▶ einfache Rechnungen sind in R ohne großes Vorwissen möglich
- ▶ das Eintippen von folgenden Befehlen liefert z.B.

-Output

```
> 1 + 2
[1] 3
> sqrt(4) +3 # built-in Funktion sqrt
[1] 5
> testvektor <- exp(4) # Erzeugt ein Objekt
> testvektor
[1] 54.59815
```

# Skripte

- ▶ typischerweise werden R-Befehle nicht einzeln getippt, sondern in Skript-Dateien gesammelt
  - ▶ Textdateien mit der Dateiendung `.r`
- ▶ Vorteile:
  - ▶ Übersichtlichkeit
  - ▶ Fehlerkorrektur
  - ▶ Dokumentation
  - ▶ Wiederholung der Analysen auf Knopfdruck
- ▶ RStudio hat einen eigenen Editor, der u.a. solche Dateien öffnen und speichern kann



# Working Directory

Der Dateizugriff kann mit absoluten oder relativen Pfaden erfolgen:

- ▶ **absoluter Pfad:** der Dateipfad wird vollständig angegeben (betriebssystem- und nutzerspezifisch)
- ▶ **relativer Pfad:** der Zugriff erfolgt relativ zum **Working Directory**, welches alternativ festgelegt werden kann durch
  - ▶ Session > Set Working Directory > Choose Directory
  - ▶ Übergabe eines absoluten Pfades mit `setwd()`
  - ▶ Starten von R-Studio über eine Skript Datei, die sich im Working Directory befindet
- ▶ das aktuell eingestellte Working Directory kann mit `getwd()` ausgelesen werden
- ▶ beachte: die Pfadangabe ist betriebssystemabhängig
  - ▶ Unix/Mac: forward slashes, z.B. `"/Users/brunned/bw09"`
  - ▶ Windows: back slashes, z.B. `"C:\\Users\\brunned\\bw09"`

# Working Directory

- ▶ **Beispiel:** falls der Ordner BW09 das Working Directory ist, lässt sich die Datei `beispiel.RData` im Unterordner `data`, wie folgt einlesen:

 R-Code

```
rm(list = ls()) # loescht alle Objekte in der Arbeitsumgebung  
  
# relativer Pfad:  
load(file = "data/beispiel.RData")  
  
# Auslesen des Working Directory  
getwd() # Pfadangabe unter MAC liefert "/Users/brunned/bw09_k4/BW09"
```

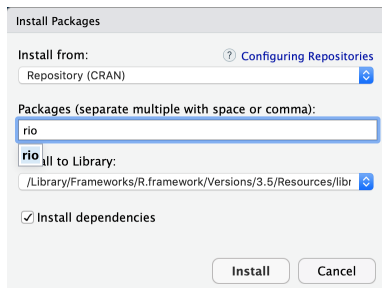
- ▶ alternativ lässt sich auch der absolute Pfad nutzen:

 R-Code

```
# absoluter Pfad (eingestelltes Working Directory irrelevant)  
load(file = "/Users/brunned/BW09/data/beispiel.RData")
```

# Pakete

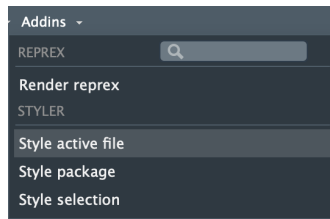
- ▶ der Funktionsumfang von R lässt sich komfortabel durch die Installation zusätzlicher Pakete erweitern
- ▶ **Vorgehen:** Tools > Install Packages



- ▶ **in Kurs 3 benötigte Pakete:** rio, tidyverse, tidytext, quantmod, hrbrthemes, lubridate, microbenchmark, Rcpp, foreachf, doParallel, styler

# Coding Style

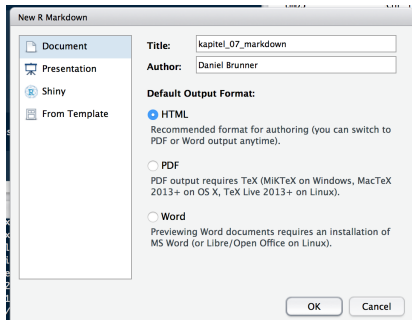
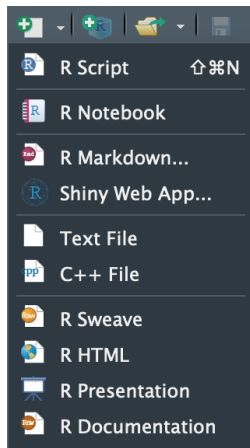
- ▶ es gibt eine Reihe von Konventionen, die bei Beachtung die Leserlichkeit von Code erhöhen und das Fehlerpotential reduzieren:
  - ▶ [Google's R Style Guide](#)
  - ▶ [The tidyverse style guide](#)
- ▶ wir wählen hier eine sehr einfache Vorgehensweise und lassen das Paket `styler` die Arbeit erledigen
- ▶ nach der Paketinstallation lässt sich die Skriptdatei auf Knopfdruck formatieren:



# R-Markdown

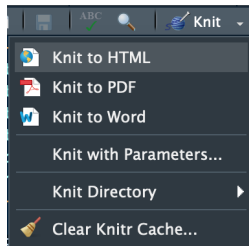
- ▶ komfortable Möglichkeit Reports unserer Analysen zu generieren: **R Markdown**
- ▶ die Funktionalität von R Markdown wird von RStudio unterstützt
- ▶ ein umfangreiches Beispiel befindet sich in der Datei `introMarkdown.Rmd`
  - ▶ wir werden hier nur `html` Dokumente erstellen...
  - ▶ PDF's sind ebenfalls problemlos generierbar, allerdings muss vorher eine komplette [Latex Distribution](#) installiert sein
  - ▶ Präsentationsfolien (im PDF Format) sind dann ebenfalls möglich
- ▶ für die ersten grundlegenden Schritte dienen die folgenden Folien

# R-Markdown



# R-Markdown

```
1 + ---
2 title: "kapitel_07_markdown"
3 author: "Daniel Brunner"
4 date: "2/19/2018"
5 output: html_document
6 ---
7
8 ```{r setup, include=FALSE}
9 knitr::opts_chunk$set(echo = TRUE)
10 ```
11
12 ## R Markdown
13
14 This is an R Markdown document. Markdown is a simple formatting syntax for
15 authoring HTML, PDF, and MS Word documents. For more details on using R
16 Markdown see <http://rmarkdown.rstudio.com>.
```



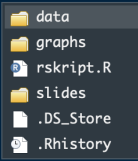
# Tips & Tricks

- ▶ **Ausführung der aktuellen Zeile** der Skriptdatei:
  - ▶ Windows: `STRG` + `Enter`
  - ▶ MAC: `command` + `Enter`
- ▶ **Navigation** im Dateisystem: Eingabe von `"` und anschließendes Drücken von `tab`


`load(file = "|` + `tab` =

`load(file = "data/|` + `tab` =

```
load(file = "|
```



```
load(file = "data/|
```





# Tips & Tricks

- ▶ Einfügen von Trennstrichen (z.B. ----) erlaubt eine **Strukturierung** der Skriptdatei:


- ▶ ohne Strukturierung:

```
10 ## Working Directory
11 # Aktuelles Working Directory ist korrekt eingestellt:
12 getwd() # Pfadangabe unter MAC OS
13
14 # absoluter Pfad
15 load(file = "/Users/brunned/BW09/_chap01_k3/data/beispiel.RData")
16
17 # relativer Pfad
18 load(file = "data/beispiel.RData")
19
20 ## Tips und Tricks
21 load(file = "")
22
23 load(file="data/")
```

- ▶ mit Strukturierung:

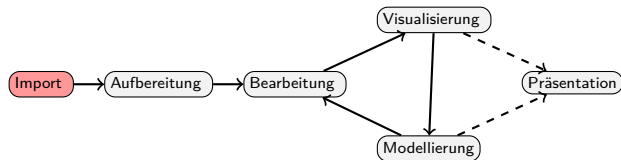
```
10 ## Working Directory ----
11 # Aktuelles Working Directory ist korrekt eingestellt:
12 getwd() # Pfadangabe unter MAC OS
13
14 # absoluter Pfad
15 load(file = "/Users/brunned/BW09/_chap01_k3/data/beispiel.RData")
16
17 # relativer Pfad
18 load(file = "data/beispiel.RData")
19
20 ## Tips und Tricks ----
21 load(file = "")
22
23 load(file="data/")
```

- ▶ mit Strukturierung und Komprimierung:

```
10 ## Working Directory 
20 ## Tips und Tricks ----
21 load(file = "")
22
23 load(file="data/")
```

# Inhalte

1. Grundlagen der Software 
2. Datenimport
  - Datentypen
  - Datenimport und -export
  - Einfache Datenanalysen
  - Grafiken
3. Grundlagen der Datenaufbereitung und -bearbeitung
4. Fortgeschrittene Datenaufbereitung und -bearbeitung
5. Grafische Auswertung
6. Unstrukturierte Daten
7. Performance
8. Big Data



# Allgemeines zu Daten in diesem Kurs

- ▶ in diesem Kurs werden wir uns häufig mit **strukturierten Daten** beschäftigen, d.h.
  - ▶ es liegen Daten für verschiedene Beobachtungen...
  - ▶ ...mit identischen Variablen vor
  - ▶ die Daten weisen also eine gleichartige Struktur auf
- ▶ um Daten in R einzulesen werden wir überwiegend auf **externe Datenquellen** zugreifen
  - ▶ dies kann u.a. eine lokal abgespeicherte Excel Datei sein
  - ▶ oder ein online bereit gestellter Datensatz

# Datentypen und Variablen

- ▶ **Datentyp:** legt fest wie R Daten intern speichert und welche Funktionen zulässig sind
- ▶ es gibt vier wichtige **Elementartypen:**
  - ▶ logical: TRUE oder FALSE
  - ▶ integer: -2L, 3L, 4L,...
  - ▶ double (auch als numeric bezeichnet): 1.12, 5.3,...
  - ▶ character: "abc",... oder 'abc'...
- ▶ **strukturierte Typen** bestehen aus Daten gleichartiger oder unterschiedlicher Elementartypen (atomic type) und/ oder weiteren strukturierten Typen
- ▶ wichtige strukturierte Typen in diesem Kurs:
  - ▶ **Vektoren:** eine Dimension für Werte vom gleichen Elementartyp
  - ▶ **Matrizen:** zwei Dimensionen für Werte vom gleichen Elementartyp
  - ▶ **Datensätze:** Beobachtungseinheiten in Zeilen, Variablen mit Werten unterschiedlicher Elementartypen in Spalten
- ▶ Daten eines bestimmten Typs können mit <- dauerhaft unter einem Objektnamen gespeichert werden

# Datentypen und Variablen

**Exkurs:** wie speichert der Computer Werte des Typs `double`?

- ▶ aus der R Dokumentation: *“All R platforms are required to work with values conforming to the IEC 60559 (also known as IEEE 754) standard. This basically works with a precision of 53 bits, and represents to that precision a range of absolute values from about  $2e-308$  to  $2e+308$ .”*
- ▶ eine Gleitkommazahl wird im Format

$$x = V \cdot M \cdot B^E$$


gespeichert, wobei  $V$  das Vorzeichen,  $M$  die Mantisse,  $B$  die Basis und  $E$  den Exponent beschreibt

- ▶ für einen Wert des Typs `double` werden insgesamt 64 Bits im Speicher reserviert (siehe IEEE 754)
  - ▶ 1 Bit für  $V$  und 52 Bits für  $M$  (`.Machine$double.digits`)
  - ▶ 11 Bits für  $E$  (`.Machine$double.exponent`)

# Datentypen und Variablen

**Exkurs:** wie speichert der Computer Werte des Typs double?

- ▶ 64 Bits garantieren eine Genauigkeit von mindestens 15 Ziffern einer Dezimalzahl
- ▶ viele weitere Informationen sind auch mit dem R-Befehl `.Machine` auslesbar
- ▶ Beispiele:

 R-Output

```
> sprintf("%.25f", 0.1) # Beispiel 1
[1] "0.1000000000000000055511151"
>
> sprintf("%.25f", 100000.1)# Beispiel 2
[1] "100000.1000000000058207660913467"
>
> test <- seq(-10, 10, length.out = 10) # Beispiel 3
> mean(test)
[1] 3.549244e-16
```

# Datentypen und Variablen

Erläuterungen zu den Beispielen:

- ▶ **Beispiel 1:** obwohl wir den exakten Wert 0.1 speichern und ausgeben, sorgt die 64 Bit Kodierung für Rundungsfehler (hier: 18. Nachkommastelle)
- ▶ **Beispiel 2:** der Rundungsfehler tritt hier schon in der 12. Nachkommastelle auf, da 64 Bit nur für eine Genauigkeit von 15 Ziffern (inkl. Ziffern vor dem Dezimaltrennzeichen) ausreichen
- ▶ **Beispiel 3:** wir wissen, dass das arithmetische Mittel exakt 0 betragen sollte, aber ausgegeben wird  $3.549244e-16 = 0.00000000000000003549244$  (hier: Fehler ab der 16. Nachkommastelle, d.h. 16 Dezimalziffern sind korrekt)

# Datentypen und Variablen

## -Output

```
> ## Vektoren
> testvektor <- c(1, 5, 9, 15)
>
> # Zugriff mit eckigen Klammern:
> testvektor[ 2 ] # 2ter Wert
[1] 5
> testvektor[ c(2, 3, 4) ]
[1] 5 9 15
> testvektor[ 2:4 ]
[1] 5 9 15
> testvektor[ c(FALSE, TRUE, TRUE, TRUE) ]
[1] 5 9 15
```

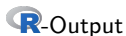


# Datentypen und Variablen

## -Output

```
> ## Matrizen
> testmat <- matrix(c(1, 5, 9, 15), nrow = 2, ncol = 2)
> testmat
      [,1] [,2]
[1,]    1    9
[2,]    5   15
> # Zugriff mit eckigen Klammern und Komma:
> testmat[ 2, 1 ] # 2te Zeile, 1te Spalte
[1] 5
> # 1te Spalte:
> testmat[ c(1, 2), 1]
[1] 1 5
> testmat[, 1]
[1] 1 5
```

# Datentypen und Variablen



```
> ## Datensätze
> testdf <- data.frame(
+   var1 = c(1, 4, 9),
+   var2 = c("a", "b", "c"),
+   stringsAsFactors = FALSE
+ )
> testdf
  var1 var2
1    1   a
2    4   b
3    9   c
> # Zugriff auf var2:
> testdf$var2
[1] "a" "b" "c"
> testdf[, 2 ]
[1] "a" "b" "c"
> testdf[[ 2 ]]
[1] "a" "b" "c"
> # Zugriff auf Teile von var2:
> testdf$var2[ 1:2 ]
[1] "a" "b"
```

# Datenimport und -export

- ▶ in den meisten Fällen werden Daten nicht selbst in der Konsole erfasst, sondern aus externen Datenquellen **importiert**
- ▶ hier wird u.a. das Paket `rio` für den Import/ Export genutzt:
  - ▶ leistungsfähiger und bequemer Datenimport
  - ▶ viele Formate werden automatisch erkannt und importiert mit der Funktion `import()`
  - ▶ genauso bequem ist der Export mit der Funktion `export()`
- ▶ das R-eigene Format hat die Endung `.RData`
  - ▶ eignet sich für komplexere Objekte, die z.B. nicht einer Rechteckform entsprechen
  - ▶ Import/ Export mit den Funktionen `load()` bzw. `save()`

# Datenimport

## -Code

```
# Quelle: de.finance.yahoo.com:
library(rio)
aktien <- import(file = "../data/aktien.xlsx")

# Direkter Import aus Online Ressourcen:
library(quantmod)
google <- getSymbols("GOOG",
  from = "2016-12-31", to = "2017-12-31",
  auto.assign = FALSE
)
```

## -Output

```
> head(aktien, 2)
  date google volkswagen
1 2017-01-02  NA  135.8457
2 2017-01-03 886.14  138.5935
>
> head(google, 2)
  GOOG.Open GOOG.High GOOG.Low GOOG.Close GOOG.Volume GOOG.Adjusted
2017-01-03  778.81   789.63  775.80   786.14   1657300     786.14
2017-01-04  788.36   791.34  783.16   786.90   1073000     786.90
```

# Datencheck

- ▶ **Überprüfung der importierten Daten unerlässlich:**  
importierte Daten können fehlende oder ungültige Daten enthalten
- ▶ dazu hilfreich: **data dictionary** mit allen Variablenbeschreibungen und eventuellen Kordierungen für fehlende Werte
  - ▶ im yahoo finance Beispiel nicht optimal (einzige Ressource ist die [Hilfeseite](#))
  - ▶ aber Variablen größtenteils selbsterklärend (Eröffnungskurs an Tag x, höchster Kurs an Tag x,...)
- ▶ im vorherigen Beispiel fehlen z.B. immer dann Werte, wenn an einem Tag kein Handel stattfand (z.B. Feiertage)
- ▶ weitere häufige Fehlerquelle: .csv Dateien wurden zuvor in Excel geöffnet

# Arbeiten mit Datensätzen

- ▶ **Datentypen:** `import()` liest einen Datensatz als `data.frame` ein
- ▶ erste Beispiele zur Datenbearbeitung:

## R-Output

```
> ## Import:
> aktien <- import(file = "../data/aktien.xlsx")
> class(aktien)
[1] "data.frame"
> ## Zugriff auf Google:
> google <- aktien$google
> mean(google, na.rm = TRUE)
[1] 921.7808
> ## Zugriff auf Subsample:
> google_mod <- google[ google > mean(google, na.rm = TRUE) ]
> mean(google_mod, na.rm = TRUE)
[1] 975.0103
> ## Zuweisung eines Elements:
> aktien$volkswagen[5] <- 200
```

# Arbeiten mit Datensätzen

- ▶ viele Funktionen in R sind **vektoriert**, d.h. für jedes Element eines Arguments (oder mehrerer Argumente) wird die Funktion angewandt
- ▶ das folgende Beispiel demonstriert das Verhalten vektorisierter Funktionen:

## R-Output

```
> ## einfache vektorisierte Operationen:  
> c(1, 2, 3, 4) * c(2, 3, 4, 5) # Vorsicht bei unterschiedl. Laenge!  
[1] 2 6 12 20  
> 3:7 + 4:8  
[1] 7 9 11 13 15  
>  
> ## eine Anwendung:  
> spread <- aktien$google - aktien$volkswagen  
> spread[1:4]  
[1] NA 647.5465 649.9456 655.9086
```

# Arbeiten mit Datensätzen

- ▶ weitere Beispiele für mögliche Berechnungen mit der importierten Zeitreihe:
  - ▶ diskrete Rendite:  $r_t = \frac{p_t - p_{t-1}}{p_{t-1}}$
  - ▶ stetige Rendite:  $\tilde{r}_t = \ln(p_t) - \ln(p_{t-1})$
  - ▶ gleitende Durchschnitte
- ▶ die Formeln lassen sich einfach mit vektorisiertem Code berechnen, z.B.:

## R-Code

```
## Bsp. diskrete Renditeberechnung:  
kurse <- aktien$google  
T <- length(kurse)  
differenzen <- kurse[2:T] - kurse[1:(T - 1)]  
rendite <- differenzen / kurse[1:(T - 1)]
```

- ▶ großer **Vorteil** der Vektorisierung: schnelle Berechnung (mehr in Kapitel 7)



# Grafiken

- ▶ die grafischen Auswertungsmöglichkeiten in R sind unendlich und beschäftigen uns später ausführlich
- ▶ zwei nützliche Funktionen werden wir permanent benötigen:
  - ▶ der `hist` Befehl generiert ein Histogramm
  - ▶ der `plot` Befehl generiert eine 2D Grafik
- ▶ für beide Befehle folgen Beispiele...

## R-Code

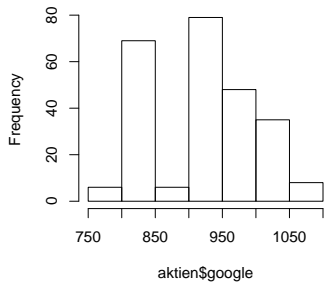
```
# Histogramm
hist(aktien$google, main = "Histogramm der Google Aktie")

# 2D Plot
time <- as.Date(aktien$date) # setze Datentyp auf "Date"
plot(time, aktien$google, main = "Kursverlauf der Google Aktie")
```

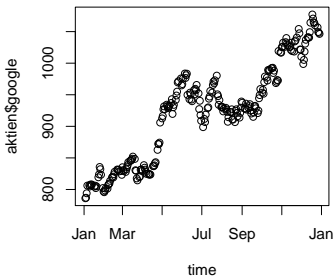
- ▶ auch Grafiken können exportiert werden: die folgende Grafik wurde z.B. mit dem Befehl `pdf()` erstellt (siehe Übung)

# Befehle hist und plot


**Histogramm der Google Aktie**

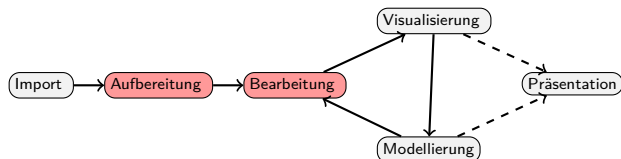


**Kursverlauf der Google Aktie**



# Inhalte

1. Grundlagen der Software 
2. Datenimport
3. Grundlagen der Datenaufbereitung und -bearbeitung  
Einführung in dplyr und tidyr  
Relationale Datensätze
4. Fortgeschrittene Datenaufbereitung und -bearbeitung
5. Grafische Auswertung
6. Unstrukturierte Daten
7. Performance
8. Big Data



# Einführung in dplyr und tidyr

- ▶ in der “echten” Welt: Daten liegen nie in der benötigten Form vor
- ▶ ggf. benötigte Schritte bevor Daten weiter analysiert werden können:
  - ▶ Rohdaten besorgen
  - ▶ Daten importieren
  - ▶ Datenfehler finden und bereinigen
  - ▶ Relevanten Unterdatensatz extrahieren
  - ▶ Verschiedene Datensätze zusammenfügen
  - ▶ Neue Variablen generieren
  - ▶ Daten zusammenfassen
- ▶ **mehrere Möglichkeiten** diese Schritte durchzuführen:
  - ▶ moderne, leistungsfähige und flexible Funktionen aus den Paketen dplyr, tidyr
  - ▶ Nutzung der Funktionen aus “base” R (also ohne spezielle Pakete)

# Einführung in dplyr

- ▶ Ziel dieser einzelnen Schritte ist ein **“aufgeräumter”** Datensatz, daher der Name für diesen Ansatz: “Tidyverse”
- ▶ Definition eines “aufgeräumten” Datensatzes aus Wickham und Grolemund (2017):
  - ▶ jede Variable steht in einer eigenen Spalte
  - ▶ jede Beobachtung steht in einer eigenen Zeile
  - ▶ jeder Wert hat seine eigene Zelle
- ▶ Ausgangspunkt in der “echten” Welt sind also alle möglichen Formen von **“unaufgeräumten”** Datensätzen
- ▶ für das Aufräumen der Daten nutzen wir im folgenden die Pakete `dplyr` und `tidyr` (neben vielen anderen in `tidyverse` enthalten)

## Beispiel für einen aufgeräumten Datensatz

- ▶ jede **Variable** hat ihre eigene Spalte:

date	google	volkswagen
2017-01-02	NA	135.85
2017-01-03	786.14	138.59
2017-01-04	786.90	136.95

- ▶ jede **Beobachtung** hat ihre eigene Zeile:

date	google	volkswagen
2017-01-02	NA	135.85
2017-01-03	786.14	138.59
2017-01-04	786.90	136.95

- ▶ jeder **Wert** hat seine eigene Zelle:

date	google	volkswagen
2017-01-02	NA	135.85
2017-01-03	786.14	138.59
2017-01-04	786.90	136.95

# Grundlegende Funktionen

- ▶ wichtige Gemeinsamkeiten **aller** Funktionen aus dplyr und tidyr:
  - ▶ das erste Argument ist immer ein Datensatz
  - ▶ der Output ist immer ein (veränderter) Datensatz
- ▶ wichtige grundlegende Funktionen:
  - ▶ rename: Umbenennung von Variablen (siehe Bsp.)
  - ▶ filter: Wähle Beobachtungen (Zeilen) nach Kriterien
  - ▶ mutate: Erzeuge neue Variablen (siehe Bsp.)
  - ▶ select: Wähle Variablen (Spalten) aus (siehe Bsp.)
  - ▶ arrange: Sortiere den Datensatz
- ▶ mit folgendem Datensatz werden einige dieser Funktionen demonstriert:

-Code

```
library(rio)
library(tidyverse) # laedt dplyr und tidyr

aktien <- import(file = "../data/aktien.xlsx")
```

# Grundlegende Funktionen: Beispiele

## R-Code

```
aktien2 <- rename(aktien, Datum = date)
aktien3 <- mutate(aktien2, total = google + volkswagen)
aktien4 <- select(aktien3, Datum, total)
```

Die Anwendung der Funktionen hat folgende Auswirkung:

## R-Output

```
> head(aktien, n = 2)
  date google volkswagen
1 2017-01-02   NA    135.8457
2 2017-01-03 786.14    138.5935
>
> head(aktien2, n = 2)
  Datum google volkswagen
1 2017-01-02   NA    135.8457
2 2017-01-03 786.14    138.5935
>
> head(aktien3, n = 2)
  Datum google volkswagen total
1 2017-01-02   NA    135.8457   NA
2 2017-01-03 786.14    138.5935 924.7335
>
> head(aktien4, n = 2)
  Datum total
1 2017-01-02   NA
2 2017-01-03 924.7335
```



# Pipes

- ▶ umständliches Vorgehen in den vorangegangenen Beispielen: schrittweise Erzeugung unterschiedlicher Datensätze
- ▶ **Pipes** erlauben, das Gleiche besser zu machen:
  - ▶ effizienter
  - ▶ leserlicher
  - ▶ weniger fehleranfällig
- ▶ eine Pipe ist eine besondere Art von Funktion, die folgende Syntax verlangt: `argument1 %>% argument2`
- ▶ die Pipe nimmt den Ausdruck auf der linken Seite und verwendet ihn als (erstes) Argument für die Funktion auf der rechten Seite
- ▶ so können mehrere Funktionen miteinander verkettet werden:

## R-Output

```
> # Einfaches Beispiel:  
> sqrt(25)  
[1] 5  
> # Das Gleiche mit Pipe:  
> 25 %>% sqrt()  
[1] 5
```

## Pipes: Beispiel

- ▶ Funktionen aus `dplyr` und `tidyr` können per Pipe einfach verkettet werden
- ▶ um den gleichen Datensatz wie in `aktien4` (obiges Beispiel) zu erzeugen, reicht folgender Code:

-Code

```
aktien_final <- aktien %>%  
  rename(Datum = date) %>%  
  mutate(total = google + volkswagen) %>%  
  select(Datum, total)
```

- ▶ das klappt nur problemlos, weil für alle Funktionen aus `dplyr` und `tidyr` gilt:
  - ▶ das erste Argument ist immer ein Datensatz
  - ▶ der Output ist immer ein (veränderter) Datensatz


# Herstellung eines “tidy” Datensatzes mit `tidyr`

- ▶ je nachdem in welcher (unaufgeräumten) Form die Daten vorliegen, sind die folgenden Funktionen nützlich:
  - ▶ `gather()`: sammelt Werte einer Variablen, die auf mehrere Spalten verteilt sind (siehe Beispiel)
  - ▶ `spread()`: teilt Werte unterschiedlicher Variablen in einer Spalte auf mehrere Spalten auf
  - ▶ `unite()`: Zusammenfügen mehrerer Spalten zu einer Spalte (siehe Beispiel)
  - ▶ `separate()`: Aufteilen einer Spalte in mehrere Spalten

# Herstellung eines "tidy" Datensatzes mit tidyr


Beispiel:

- ▶ **Ziel:** aufgeräumte Daten in folgendem Format...

 R-Output

	Wertpapier	Jahr	Rendite
1	Aktie1	2015	0.10
2	Aktie1	2016	0.05
3	Aktie2	2015	0.07
4	Aktie2	2016	0.03

- ▶ **Ausgangssituation:**

 R-Output

	Wertpapier	Rendite_2015	Rendite_2016
1	Aktie1	0.10	0.05
2	Aktie2	0.07	0.03

# Herstellung eines "tidy" Datensatzes mit tidyr

- ▶ Lösung: gather()


## -Code

```
# Datenerstellung:
datensatz_untidy <- data.frame(
  Wertpapier = c("Aktie1", "Aktie2"),
  Rendite_2015 = c(0.1, 0.07),
  Rendite_2016 = c(0.05, 0.03)
)

# Anwendung von gather:
datensatz_tidier <- datensatz_untidy %>%
  gather(Rendite_2015, Rendite_2016,
         key = "Zeitpunkt", value = "Rendite"
  )
```

# Herstellung eines "tidy" Datensatzes mit tidyr

## ► Ergebnis:

-Output

```
> datensatz_tidier
  Wertpapier   Zeitpunkt Rendite
1   Aktie1 Rendite_2015  0.10
2   Aktie2 Rendite_2015  0.07
3   Aktie1 Rendite_2016  0.05
4   Aktie2 Rendite_2016  0.03
```


- noch problematisch: Variable Zeitpunkt ist nicht numerisch
- Lösung: separate()

-Code

```
datensatz_tidy <- datensatz_tidier %>%
  separate(Zeitpunkt,
    into = c("uninteressant", "Jahr"),
    sep = "_"
  )
```

# Herstellung eines “tidy” Datensatzes mit tidyr

- ▶ der Datensatz ist nun aufgeräumt:

-Output

```
> datensatz_tidy
  Wertpapier uninteressant Jahr Rendite
1   Aktie1      Rendite 2015   0.10
2   Aktie2      Rendite 2015   0.07
3   Aktie1      Rendite 2016   0.05
4   Aktie2      Rendite 2016   0.03
```

# Keys

- ▶ **Keys** sind Variablen, die eine Beobachtung eindeutig identifizieren
  - ▶ **primary Key**: kennzeichnet eine Beobachtung eindeutig im “eigenen” Datensatz
  - ▶ **foreign Key**: kennzeichnet eine Beobachtung eindeutig in einem fremden Datensatz (auch Sekundärschlüssel genannt)
  - ▶ beachte: bei dem Kriterium der “Eindeutigkeit” geht es nicht darum, ob eine Variable für gegebene Daten nur unterschiedliche Werte annimmt, sondern darum, ob dies auch theoretisch, d.h. für beliebig viele weitere Beobachtungen, sichergestellt ist (siehe Variable Schlusskurs)

- ▶ **Beispiel:** Datensatz 1

Datum	Schlusskurs	Wertpapier
2017-02-15	135.51	Apple
2017-02-14	135.02	Apple
2017-02-13	133.29	Apple
2017-02-10	132.12	Apple
2017-02-09	132.42	Apple

- ▶ das Datum ist in Datensatz 1 ein primary Key!



▶ **Beispiel (Fortsetzung):** Datensatz 2

Datum	Handelsplätze
2017-02-15	Xetra
2017-02-15	NASDAQ
2017-02-15	LSE
2017-02-14	Xetra
2017-02-14	NASDAQ

- ▶ in Datensatz 2 ist die Variable Datum ein foreign Key (identifiziert einen Kurs in Datensatz 1 eindeutig)
- ▶ im Folgenden fügen wir Datensätze zusammen, die über primary und foreign Keys zueinander in Beziehung stehen
  - ▶ durch eine **1 zu 1 Beziehung** (Bsp.: ein beliebiges Datum aus Datensatz 1 kommt maximal einmal in Datensatz 2 vor)
  - ▶ durch eine **1 zu n Beziehung** (Bsp.: ein beliebiges Datum aus Datensatz 1 kann mehrmals in Datensatz 2 vorkommen)

## Datensätze vereinen mit Keys


- ▶ **Ziel:** Matching von Beobachtungen unterschiedlicher Datensätze anhand von Keys
- ▶ Funktionen, die uns diese Arbeit abnehmen, folgen dem gleichen Muster:

*funktionsname(datensatz<sub>links</sub>, datensatz<sub>rechts</sub>, key)*

- ▶ die join Funktionen unterscheiden sich im Umgang mit **nicht gematchten** Beobachtungen:
  - ▶ `inner_join`: finaler Datensatz enthält nur gematchte Daten
  - ▶ `left_join`: finaler Datensatz enthält Daten aus linkem Datensatz und gematchte
  - ▶ `right_join`: finaler Datensatz enthält die Daten aus rechtem Datensatz und gematchte
  - ▶ `full_join`: finaler Datensatz enthält Daten aus linkem und rechtem Datensatz (gematcht und nicht gematcht)
- ▶ gut nachvollziehbar, wenn primary Keys aus zwei Datensätzen in Beziehung gesetzt werden

## Beispiel: Matching mit 1 zu 1 Beziehungen

- ▶ als Ausgangspunkt dienen uns zwei Datensätze mit Keys, die zugleich primary als auch foreign sind (1 zu 1 Beziehung)

 R-Output

```
> daten_apple
  Datum Kurs_Apple
1 2017-02-15     135.51
2 2017-02-14     135.02
3 2017-02-13     133.29
>
> daten_amazon
  Datum Kurs_Amazon
1 2017-02-16     844.14
2 2017-02-15     842.70
3 2017-02-14     836.39
```

- ▶ in beiden Datensätzen existieren Beobachtungen, die sich nicht mit dem jeweils anderen Datensatz matchen lassen


# Prinzip des inner\_join am Beispiel (1 zu 1 Beziehung)

Datum	Kurs_Apple
2017-02-15	135.51
2017-02-14	135.02
2017-02-13	133.29

Datum	Kurs_Amazon
2017-02-16	844.14
2017-02-15	842.70
2017-02-14	836.39

inner\_join()

Datum	Kurs_Apple	Kurs_Amazon
2017-02-15	135.51	842.70
2017-02-14	135.02	836.39

-Output

```
> library(tidyverse)
> inner_join(daten_apple, daten_amazon, by = "Datum")
  Datum Kurs_Apple Kurs_Amazon
1 2017-02-15     135.51     842.70
2 2017-02-14     135.02     836.39
```

# Prinzip anderer join Befehle (1 zu 1 Beziehung)

Datum	Kurs_Apple
2017-02-15	135.51
2017-02-14	135.02
2017-02-13	133.29

left\_join()

Datum	Kurs_Amazon
2017-02-16	844.14
2017-02-15	842.70
2017-02-14	836.39

Datum	Kurs_Apple	Kurs_Amazon
2017-02-15	135.51	842.70
2017-02-14	135.02	836.39
2017-02-13	133.29	-

Datum	Kurs_Apple
2017-02-15	135.51
2017-02-14	135.02
2017-02-13	133.29

right\_join()

Datum	Kurs_Amazon
2017-02-16	844.14
2017-02-15	842.70
2017-02-14	836.39

Datum	Kurs_Apple	Kurs_Amazon
2017-02-16	-	844.14
2017-02-15	135.51	842.70
2017-02-14	135.02	836.39

# Prinzip anderer join Befehle am Beispiel (1 zu 1 Beziehung)

## R-Output

```
> left_join(daten_apple, daten_amazon, by = "Datum")
  Datum Kurs_Apple Kurs_Amazon
1 2017-02-15     135.51     842.70
2 2017-02-14     135.02     836.39
3 2017-02-13     133.29         NA
>
> right_join(daten_apple, daten_amazon, by = "Datum")
  Datum Kurs_Apple Kurs_Amazon
1 2017-02-16         NA     844.14
2 2017-02-15     135.51     842.70
3 2017-02-14     135.02     836.39
```


# Prinzip anderer join Befehle am Beispiel (1 zu 1 Beziehung)

Datum	Kurs_Apple
2017-02-15	135.51
2017-02-14	135.02
2017-02-13	133.29

Datum	Kurs_Amazon
2017-02-16	844.14
2017-02-15	842.70
2017-02-14	836.39

full\_join()

Datum	Kurs_Apple	Kurs_Amazon
2017-02-16	-	844.14
2017-02-15	135.51	842.70
2017-02-14	135.02	836.39
2017-02-13	133.29	-

-Output

```
> full_join(daten_apple, daten_amazon, by = "Datum")
  Datum Kurs_Apple Kurs_Amazon
1 2017-02-15    135.51    842.70
2 2017-02-14    135.02    836.39
3 2017-02-13    133.29         NA
4 2017-02-16         NA    844.14
```

# Prinzip anderer join Befehle am Beispiel (1 zu n Beziehung)

- ▶ die vorgestellten Befehle sind auch auf **1 zu n** Beziehungen anwendbar
- ▶ folgendes Beispiel demonstriert das Prinzip:

Datum	Schlusskurs
2017-02-15	135.51
2017-02-14	135.02
2017-02-13	133.29
2017-02-10	132.12
2017-02-09	132.42

Datum	Handelsplätze
2017-02-15	Xetra
2017-02-15	NASDAQ
2017-02-15	LSE
2017-02-14	Xetra
2017-02-14	NASDAQ


inner\_join()

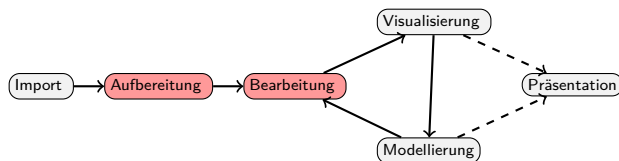
Datum	Schlusskurs	Handelsplätze
2017-02-15	135.51	Xetra
2017-02-15	135.51	NASDAQ
2017-02-15	135.51	LSE
2017-02-14	135.02	Xetra
2017-02-14	135.02	NASDAQ

- ▶ was passiert bei **n zu n** Beziehungen?



# Inhalte

1. Grundlagen der Software 
2. Datenimport
3. Grundlagen der Datenaufbereitung und -bearbeitung
4. Fortgeschrittene Datenaufbereitung und -bearbeitung
  - Einführung
  - Kontrollstrukturen
  - Funktionen
5. Grafische Auswertung
6. Unstrukturierte Daten
7. Performance
8. Big Data



# R als Programmiersprache

- ▶ Ziel dieses Kapitels ist weiterhin die Herstellung eines “aufgeräumten” Datensatzes
- ▶ um komplexere Aufräumarbeiten durchzuführen, werden wir R als Programmiersprache verwenden
- ▶ R ist eine vollwertige objektorientierte Programmiersprache, die sehr flexible Datenmanipulationen erlaubt
  - ▶ sinnvoll, wenn Standardfunktionen (z.B. aus tidyverse ) nicht mehr weiterhelfen

# Kontrollstrukturen

- ▶ häufig geht es darum, einen bestimmten Codeabschnitt...
  - ▶ nur dann ausführen zu lassen wenn eine oder mehrere Bedingungen erfüllt sind
  - ▶ wiederholt ausführen zu lassen, wobei die Anzahl der Wiederholungen bekannt oder unbekannt sein kann
- ▶ eine einfache Bedingung ist vom Typ `logical` und entweder `TRUE` oder `FALSE` und kann von einem `if` Statement ausgewertet werden



```
# ausfuehrbarer Code ohne Fehlermeldung:  
einfache_Bedingung <- FALSE  
if (einfache_Bedingung) {  
  log(-5)  
}
```

# Operatoren

eine einfache Bedingung kann durch Vergleichsoperatoren realisiert werden

- ▶ gleich: ==
- ▶ ungleich: !=
- ▶ kleiner/ kleiner gleich: < <=
- ▶ größer/ größer gleich: > >=



```
# ausfuehrbarer Code ohne Fehlermeldung:  
einfache_Bedingung <- 1 != 1 # Gegenteil von 1 == 1  
if (einfache_Bedingung) {  
  log(-5)  
}
```

# Operatoren

**logische Operatoren** kombinieren mehrere einfache Bedingungen zu einer zusammengesetzten Bedingung

- ▶ und: & (true, wenn alle Bedingungen true)
- ▶ oder: | (true, wenn mindestens eine Bedingung true)

-Code

```
bedingung_1 <- 1 == 2
bedingung_2 <- 2 < 1
bedingung_3 <- 1 != 2
# Beispiel 1: wertet zu FALSE aus
bedingung_and <- bedingung_1 & bedingung_2 & bedingung_3
# Beispiel 2: wertet zu TRUE aus
bedingung_or <- bedingung_1 | bedingung_2 | bedingung_3
```

## if else Statements

- ▶ wie im Eingangsbeispiel gezeigt, können einfache oder zusammengesetzte Bedingungen dazu genutzt werden, um zu entscheiden, ob bestimmte Code Teile ausgeführt werden sollen (`if` Teil)
- ▶ falls die Bedingung nicht zutrifft, kann optional Alternativcode ausgeführt werden (`else` Teil)



```
# berechne den Absolutbetrag
wert <- -5
if (wert < 0) {
  absolut_wert <- -wert # wird ausgefuehrt
} else {
  absolut_wert <- wert
}
```

# Schleifen

- ▶ for Schleife: Anzahl der Wiederholungen für bestimmte Codeabschnitte **bekannt**

 R-Code

```
## for
test_vektor <- c(1, 4, 2, 6, 3, 2)
erste_differenz <- numeric(5)
for (i in 2:6) {
  erste_differenz[i - 1] <- test_vektor[i] -
    test_vektor[i - 1]
}
erste_differenz # enthaelt:  3 -2  4 -3 -1
```

- ▶ der Index  $i$  wird in jedem Schleifendurchlauf automatisch um 1 erhöht bis die vorgegebene Anzahl an Durchläufen erreicht ist

# Schleifen

- ▶ sollte die Anzahl der Wiederholungen **unbekannt** sein, kann eine `while` Schleife genutzt werden
- ▶ die prüft vor jedem Schleifendurchlauf, ob die Schleifenbedingung erfüllt ist oder nicht

## R-Code

```
## while
test_vektor <- c(1, 4, 2, 6, 3, 2)
erste_differenz <- numeric(5)
i <- 2
while (i <= length(test_vektor)) {
  erste_differenz[i - 1] <- test_vektor[ i ] -
    test_vektor[ i - 1 ]
  i <- i + 1
}
erste_differenz# enthaelt:  3 -2  4 -3 -1
```



# Funktionen

- ▶ Funktionen greifen u.a. auf die eben vorgestellten Konzepte zurück
- ▶ Funktionen haben viele Vorteile:
  - ▶ mehrfach benötigter Code wird übersichtlich in einer Funktion zusammengefasst
  - ▶ Veränderungen im Code müssen an nur einer Stelle vorgenommen werden
  - ▶ geringere Fehleranfälligkeit
- ▶ Aufbau einer Funktion:
  - ▶ Funktionskopf mit Namen und Argumenten: Wie heißt die Funktion und welche Eingangsinformationen werden benötigt?
  - ▶ Funktionskörper: Was macht die Funktion?

# Aufbau einer Funktion: Beispiel



```
# Definition der Funktion
unsere_erste_funktion <- function(arg1, arg2) {
  out <- arg1 + arg2 + 5
  return(out)
}
# Aufruf der Funktion
unsere_erste_funktion(arg1 = 5, arg2 = 15) # liefert: 25
```

# Scoping

- ▶ in R legen sogenannte Umgebungen (environment) fest, wie Funktionen die Werte einer bestimmten Variable finden
- ▶ jede Funktion erstellt beim Aufruf seine eigene Umgebung
  - ▶ sollte ein aufgerufene Variable nicht gefunden werden, wird in der hierachisch höheren Umgebung (bei uns eigentlich immer die globale Umgebung) gesucht (lexical scoping)
  - ▶ die Funktion hat aber nur lesenden Zugriff auf Werte anderer Umgebungen, d.h. Veränderungen innerhalb der Funktion finden nur lokal statt
- ▶ Argumente der Funktion werden ebenfalls nur lokal verändert
- ▶ **guter Programmierstil:** das Verhalten der Funktion hängt nur von übergebenen Argumenten ab und nicht von global definierten Variablen (bzw. Variablen aus anderen Umgebungen)

# Scoping

## -Output

```
> a <- 2
> b <- 3
> test_function <- function(arg1) {
+   a <- arg1^b # Zugriff auf b (globale Umgebung)
+   # Veraenderung von a nur lokal
+   return(a)
+ }
> test_function(4)
[1] 64
>
> a
[1] 2
```

# Anwendungsbeispiel: stetige Rendite



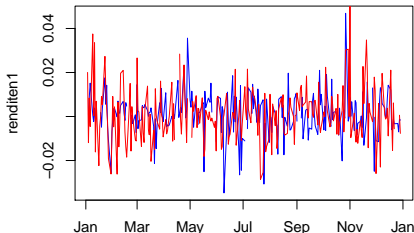
```
stetige_rendite <- function(zeitreihe) {  
  T <- length(zeitreihe)  
  
  ergebnis <- numeric(T - 1)  
  for (i in 2:T) {  
    ergebnis[i - 1] <- log(zeitreihe[i]) - log(zeitreihe[i - 1])  
  }  
  
  return(ergebnis)  
}
```

# Anwendungsbeispiel: stetige Rendite




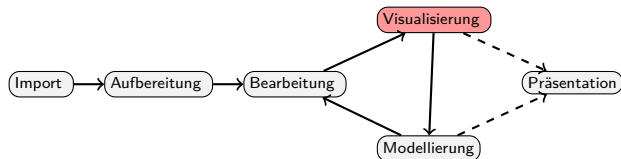
```
library(rio)
test_daten <- import(file = "../data/aktien.xlsx")
renditen1 <- stetige_rendite(test_daten$google)
renditen2 <- stetige_rendite(test_daten$volkswagen)
datum <- as.Date(test_daten$date[-1])

plot(datum, renditen1, type = "l", col = "blue")
lines(datum, renditen2, type = "l", col = "red")
```



# Inhalte

1. Grundlagen der Software 
2. Datenimport
3. Grundlagen der Datenaufbereitung und -bearbeitung
4. Fortgeschrittene Datenaufbereitung und -bearbeitung
5. Grafische Auswertung
  - Einführung in ggplot2
  - Grafiken mit einer Variablen
  - Grafiken mit zwei Variablen
  - Grafiken mit mehreren Variablen
  - Statistische Transformationen
6. Unstrukturierte Daten
7. Performance
8. Big Data



# Einführung in ggplot2

- ▶ **Grafiken:** eines der wichtigsten Werkzeuge für die Datenanalyse!
- ▶ in R gibt es viele Wege, diese zu erzeugen
  - ▶ “base” R, z.B. mit den Funktionen `plot()` oder `hist()`
  - ▶ **hier:** Einführung in die leistungsfähige und beliebte Alternative `ggplot2`
- ▶ das “gg” in `ggplot` steht für “grammar of graphics” und umschreibt die Philosophie des Ansatzes
- ▶ auf die Weise werden z.B. die meisten Grafiken bei Google, Microsoft, New York Times, etc. erzeugt
- ▶ `ggplot2` ist enthalten im Paket `tidyverse`
- ▶ **Ausgangspunkt:** ein “aufgeräumter” Datensatz



# Einführung in ggplot2

Konstruktion einer ggplot-Grafik:

- ▶ ggf. Paket laden: `library(ggplot2)` (oder `tidyverse`)
- ▶ neue Grafik erzeugen: `ggplot`
- ▶ geometrische Objekte (Geoms) werden mit `+` angehängt
- ▶ weitere Formatierungen (Hintergrundfarben, Titel, Legende, Achsenskalierung...) möglich, siehe Beispiele

Geoms können kategorisiert werden nach:

- ▶ der Anzahl der zu plottenden Variablen
- ▶ der Skalierung von Variablen (stetig oder diskret)

# Einführung in ggplot2

- ▶ jede Grafik besteht aus einem oder mehreren geometrischen Objekten (“Geom”), z.B.
  - ▶ Linien: mit `geom_line`
  - ▶ Punkte: mit `geom_point`
  - ▶ Balken: mit `geom_bar`
  - ▶ Stufen: mit `geom_step`
- ▶ jedes Geom benötigt
  - ▶ einen Datensatz (`data`), in dem die Daten stehen
  - ▶ eine Zuordnung (`mapping`) der enthaltenen Variablen zu grafischen Elementen (`aesthetics`):
    - ▶ x: x-Achse
    - ▶ y: y-Achse (ggf.)
  - ▶ ggf. Farbe, Punktgröße, Form, etc.
  - ▶ ggf. zusätzliche Optionen wie die allgemeine Farbe, Transparenz, etc.
- ▶ hilfreich um den Überblick zu behalten: [ggplot cheatsheet](#)

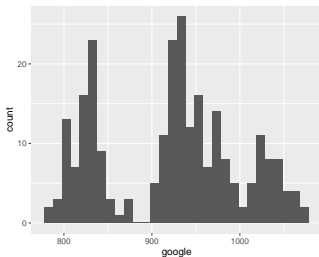
# Grafiken mit einer Variablen

- ▶ grafische Darstellungsmöglichkeit für **diskrete** Variablen:
  - ▶ Balkendiagramm mit `geom_bar`
- ▶ grafische Darstellungsmöglichkeiten für **stetige** Variablen:
  - ▶ Histogramm mit `geom_histogram`
  - ▶ Dichteschätzung mit `geom_density`

## R-Code

```
library(tidyverse) # beinhaltet ggplot2
library(rio)
aktien <- import(file = "../data/aktien.xlsx")
aktien$date <- as.Date(aktien$date)

ggplot(data = aktien, mapping = aes(google)) +
  geom_histogram()
```

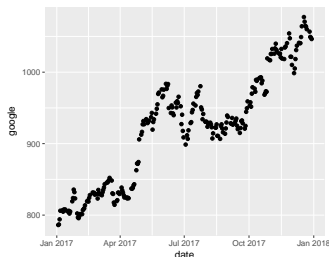


# Grafiken mit zwei Variablen

- ▶ `geom_point` und `geom_line` eignen sich zur zweidimensionalen Darstellung (ähnlich wie `plot`)
- ▶ wir betrachten ein Beispiel mit den Variablen Datum (x-Achse) und Preis (y-Achse):

## R-Code

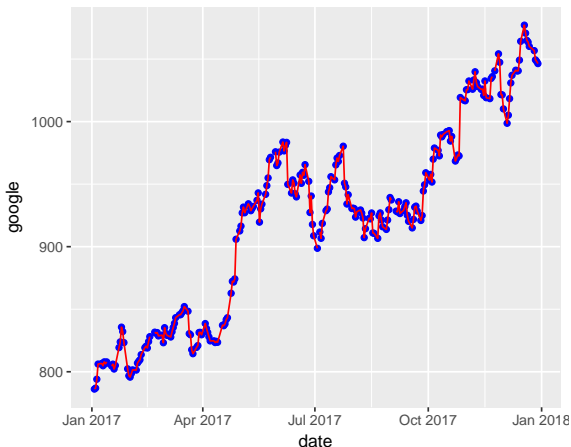
```
ggplot(data = aktien, mapping = aes(x = date, y = google)) +  
  geom_point() # geom zur Darstellung von Punkten
```



# Grafiken mit zwei Variablen (Fortsetzung)

## R-Code

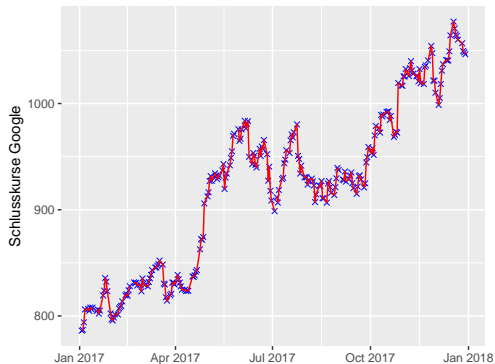
```
ggplot(data = aktien, mapping = aes(x = date, y = google)) +  
  geom_point(color = "blue") + # geom mit zus. Argument  
  geom_line(color = "red") # geom fuer Liniendiagramm
```



# Grafiken mit zwei Variablen (Fortsetzung)




```
ggplot(data = aktien, mapping = aes(x = date, y = google)) +  
  geom_point(color = "blue", shape = 4) +  
  geom_line(color = "red") +  
  xlab("") + # Beschriftung x Achse  
  ylab("Schlusskurse Google") # Beschriftung y Achse
```



## Grafiken mit mehreren Variablen

- ▶ ggplot Grafiken setzen einen “aufgeräumten” Datensatz voraus
- ▶ so eignet sich z.B. der folgende Datensatz nicht für die Darstellung mit ggplot:


-Output

```
> datensatz_untidy
  Wertpapier Rendite_2015 Rendite_2016
1   Aktie1         0.10         0.05
2   Aktie2         0.07         0.03
```

- ▶ stattdessen kann nur eine Variable (Rendite) für die y-Achse übergeben werden, die anhand einer weiteren Variable (Wertpapier) in unterschiedliche Gruppen geteilt wird

# Grafiken mit mehreren Variablen

- ▶ der Datensatz muss also in die folgende Form gebracht werden:

-Output

```
> datensatz_tidy
  Wertpapier Jahr Rendite
1   Aktie1 2015   0.10
2   Aktie1 2016   0.05
3   Aktie2 2015   0.07
4   Aktie2 2016   0.03
```

- ▶ so sieht das Prinzip der grafischen Umsetzung aus:

-Code

```
ggplot(  
  data = datensatz_tidy,  
  mapping = aes(  
    x = Jahr, y = Rendite,  
    group = Wertpapier # group betrifft alle geoms  
  )  
) + geom_line()
```



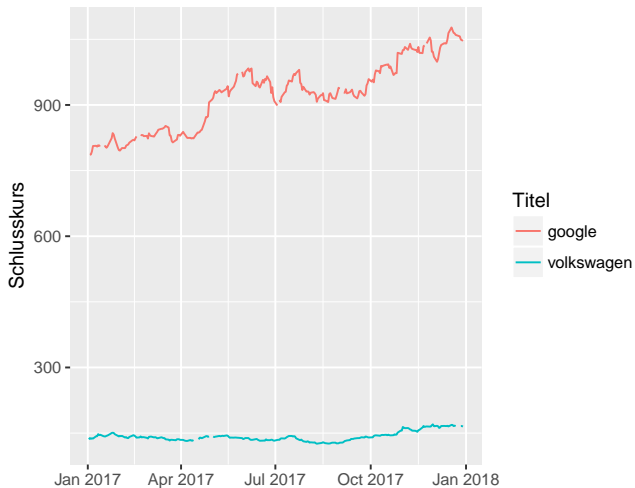
## Beispiel (Fortsetzung)



```
daten <- aktien %>%
  gather(google, volkswagen,
    key = "Titel", value = "Kurs"
  )

ggplot(
  data = daten,
  mapping = aes(
    x = date, y = Kurs,
    group = Titel,
    color = Titel
  )
) +
  # color gibt Variable fuer Farbcodierung an
  geom_line() +
  xlab("") + ylab("Schlusskurs")
```

## Beispiel (Fortsetzung)



# Statistische Transformationen

- ▶ statt der Rohdaten werden mithilfe sogenannter “Stats” Ergebnisse statistischer Methoden in die Grafik aufgenommen
- ▶ diese grafische Analyse eignet sich besonders gut zur **explorativen Datenanalyse**
- ▶ Beispiel: Glättung mit `geom_smooth`
  - ▶ ohne Angabe einer Methode wird standardmäßig eine Glättung mit lokaler Approximation durchgeführt (`span` legt fest, wie stark benachbarte Datenpunkte mit einfließen)
  - ▶ es sind verschiedene Methoden verfügbar, z.B. `lm` für eine lineare Glättung
- ▶ die statistischen Methoden, die hinter den Stats stehen, lernen wir in Kurs 4 kennen

# Beispiel (Fortsetzung)



```
ggplot(  
  data = aktien,  
  mapping = aes(x = date, y = google)  
) +  
  geom_line() +  
  geom_smooth(method = "lm", se = TRUE) + # lin. Trend  
  xlab("") +  
  ylab("Schlusskurs")
```



# Beispiel (Fortsetzung)



```
ggplot(  
  data = aktien,  
  mapping = aes(x = date, y = google)  
) +  
  geom_line() +  
  geom_smooth(method = "loess", span = 0.3) +  
  xlab("") +  
  ylab("Schlusskurs")
```



# Inhalte

1. Grundlagen der Software 
2. Datenimport
3. Grundlagen der Datenaufbereitung und -bearbeitung
4. Fortgeschrittene Datenaufbereitung und -bearbeitung
5. Grafische Auswertung
- 6. Unstrukturierte Daten**  
    Grundlegende Funktionen  
    Regular Expressions
7. Performance
8. Big Data

# Motivation

- ▶ in der **echten Welt**: Daten...
  - ▶ sind (scheinbar) unstrukturiert (z.B. strukturiert HTML die Informationen einer Website im Quellcode nach bestimmten Regeln)
  - ▶ bestehen nicht nur aus numerischen Werten (qualitative Variablen)
- ▶ diese Ausgangssituation motiviert die Diskussion grundlegender Funktionalitäten für Zeichenfolgen (engl. strings), die in R als Objekte der Klasse `character` realisiert sind
- ▶ **Ziel**: Herstellung eines aufgeräumten Datensatzes
- ▶ außerdem: viele spannende unstrukturierte Datensätze sind frei im Internet verfügbar (z.B. Tweets, Unternehmensinformationen, etc.)
- ▶ guter Einstieg: [Kapitel 14, R for Data Science](#)

# Basics in R

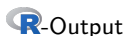
- ▶ **hier genutzt:** Paket `stringr`
  - ▶ arbeitet sehr gut mit den bisher diskutierten Funktionen zusammen und ist im Paket `tidyverse` enthalten
  - ▶ alle Funktionen sind aber auch direkt in R verfügbar ("built-in")
- ▶ weiteres Vorgehen:
  - ▶ Analyse von Zeichenfolgen mit einem **festen Muster**
  - ▶ Analyse von Zeichenfolgen mit **flexiblem Muster**: Grundlagen der sog. Regular Expressions
  - ▶ ein fortgeschritteneres Beispiel: Analyse von Tweets
- ▶ für die folgenden Beispiele dient der Datensatz `allstrings`:

## -Code

```
string1 <- "Hallo?"  
string2 <- "Good Bye"  
string3 <- "Good Day?"  
string4 <- "Good-Bye"  
string5 <- "Hello"  
string6 <- "H^llo"  
allstrings <- c(string1, string2, string3, string4, string5, string6)
```



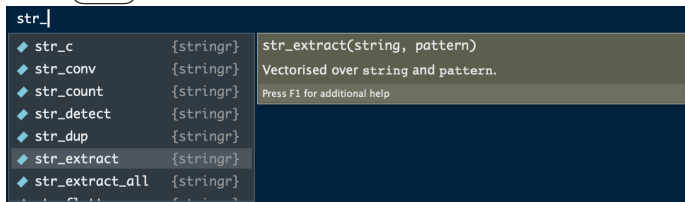
# Basics in R - Beispiele



```
> ### base R
> substr(x = allstrings, start = 1, stop = 2) # 1. bis 2. Buchstabe
[1] "Ha" "Go" "Go" "Go" "He" "H^"
> substr(allstrings, 1, 2) <- "XX" # Ueberschreibung(!) von allstrings
> allstrings
[1] "XXllo?" "XXod Bye" "XXod Day?" "XXod-Bye" "XXllo" "XXllo"
> nchar(allstrings) # Anzahl an Zeichen
[1] 6 8 9 8 5 5
>
> ### stringr (in tidyverse enthalten)
library(tidyverse)
>
> str_sub(string = allstrings, start = 3, end = 4) # 3. bis 4. Buchstabe
[1] "ll" "od" "od" "od" "ll" "ll"
> str_sub(allstrings, 1, -2) # 1. bis vorletzter Buchstabe
[1] "XXllo" "XXod Bye" "XXod Day?" "XXod-Bye" "XXll" "XXll"
> str_sub(allstrings, 3, 4) <- "XX" # Ueberschreibung(!) von x
> allstrings
[1] "XXXXo?" "XXXX Bye" "XXXX Day?" "XXXX-Bye" "XXXXo" "XXXXo"
"
"
> str_pad(allstrings, 9) # setze Laenge aller strings auf 9 Zeichen
[1] " XXXXo?" " XXXX Bye" "XXXX Day?" " XXXX-Bye" " XXXXo" "
XXXXo"
```

# Basics in R

- ▶ `stringr` enthält viele weitere Funktionen
- ▶ falls man den Überblick verliert, hilft das Tippen von `str_ + Tab`:



The screenshot shows a dark-themed R console with a tab completion menu. The prompt is `str_`. The menu lists several functions with their return types in curly braces: `str_c` {stringr}, `str_conv` {stringr}, `str_count` {stringr}, `str_detect` {stringr}, `str_dup` {stringr}, `str_extract` {stringr}, and `str_extract_all` {stringr}. The `str_extract` option is highlighted. To the right of the menu, a tooltip for `str_extract(string, pattern)` is visible, stating it is vectorised over `string` and `pattern`, and suggesting to press F1 for additional help.

```
str_  
◆ str_c           {stringr}  str_extract(string, pattern)  
◆ str_conv        {stringr}  Vectorised over string and pattern.  
◆ str_count       {stringr}  Press F1 for additional help  
◆ str_detect      {stringr}  
◆ str_dup         {stringr}  
◆ str_extract     {stringr}  
◆ str_extract_all {stringr}
```

# Regular Expression

- ▶ die bisherigen Funktionen reichen in der Praxis **nicht** aus um Zeichenfolgen effizient analysieren zu können
- ▶ ein **flexibleres Werkzeug** zum Auffinden oder Bearbeiten von Zeichenfolgen ist die “Regular Expression” (kurz RegEx)
- ▶ Regular Expression: komprimierte Sprache, die es erlaubt bestimmte Muster in Zeichenfolgen zu beschreiben
- ▶ es existieren syntaktische Unterschiede je nach RegEx-Implementierung (POSIX, Perl) und verwendeter Software

# Regular Expression und R

- ▶ Anwendung in der Programmierung:
  - ▶ **Global regular expression print** zum Auffinden von Zeichenfolgen (wichtige Argumente: Suchmuster)
  - ▶ **Global substitution** zum Ersetzen innerhalb von Zeichenfolgen (wichtige Argumente: Suchmuster, Ersetzung)
- ▶ Implementierung in R über verschiedene Pakete: `base`, `stringr`, `stringi`, `rebus`, ...
- ▶ Einige hilfreiche Ressourcen:
  - ▶ [RegExr.com](http://RegExr.com), [RegEx101.com](http://RegEx101.com) (Live Previews)
  - ▶ [awesome-RegEx](http://awesome-RegEx), [regular-expressions.info](http://regular-expressions.info) (Online Ressourcen)
  - ▶ [stringr.tidyverse.org](http://stringr.tidyverse.org) (hilfreiche Beispiele in R)
- ▶ das folgende Beispiel zeigt zwei sehr einfache Suchmuster...

## R-Output

```
> ### (Wieder-) Herstellung des Datensatzes
> allstrings <- c(string1, string2, string3, string4, string5, string6)
>
> ### base R
> grep(pattern = "ll", x = allstrings, value = TRUE) # Strings von allstrings mit "ll"
[1] "Hallo?" "Hello" "H^llo"
> gsub(pattern = "ll", replacement = "X", x = allstrings) # ersetzt Strings von allstrings mit "ll" mit "X"
[1] "HaXo?" "Good Bye" "Good Day?" "Good-Bye" "HeXo" "H^Xo"
>
> ### stringr (in tidyverse enthalten)
> str_subset(string = allstrings, pattern = "ll") # entspricht grep()
[1] "Hallo?" "Hello" "H^llo"
> str_replace(allstrings, pattern = "ll", replacement = "X") # entspricht gsub()
[1] "HaXo?" "Good Bye" "Good Day?" "Good-Bye" "HeXo" "H^Xo"
```

# Metazeichen

- ▶ komplexere RegEx Ausdrücke werden mit Hilfe von **Metazeichen** [ ] ( ) { } | ? + - \* ^ \$ \ . in Kombination mit dem Alphabet gebildet
- ▶ Metazeichen können auch kontextabhängig unterschiedlich funktionieren, je nachdem in welcher Kombination sie verwendet werden (u.a. ^ - +)
- ▶ wird ein Metazeichen innerhalb einer Zeichenfolge als Zeichen benötigt, kann die Metaeigenschaft mit einem Backslash (escape character) \ aufgehoben werden
- ▶ da RegEx Ausdrücke in R als String definiert werden müssen, und \ bereits innerhalb eines Strings "escape" Eigenschaften besitzt, muss auch dieses innerhalb des Strings "escaped" werden:
  - ▶ **Beispiel:** ? wird als Zeichen gesucht → Suchmuster in R: \\?

# Eine Auswahl an Metazeichen

- ▶ `.`: Wildcard-Zeichen, d.h. Platzhalter für beliebige Zeichen
- ▶ `[ ]`: die Zeichenmenge in der eckigen Klammer beschreibt Zeichenfolgen mit genau einem Zeichen aus der Zeichenmenge
  - ▶ diese Zeichenmenge beschreibt eine **Zeichenklassendefinition**
- ▶ `^`: kontextabhängig...
  - ▶ Bedeutung am Anfang einer Zeichenklassendefinition: Negation
  - ▶ sonstige Bedeutung in einer Zeichenklassendefinition: Interpretation als Zeichen
  - ▶ Bedeutung außerhalb Zeichenklassendefinition: Verweis auf den Anfang einer Zeichenfolge
- ▶ `-`: kontextabhängig...
  - ▶ Bedeutung am Anfang und Ende einer Zeichenklassendefinition: Interpretation als Zeichen
  - ▶ sonstige Bedeutung in einer Zeichenklassendefinition: Interpretation als Zeichenintervall, z.B. `[A-Z]`, `[a-zA-Z]`, `[0-9]`, ...

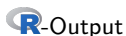
# RegEx in R

## -Output

```
> ## Escape Characters
> str_subset(allstrings, pattern = "?") # Interpretation als RegEx Metazeichen
Error [...]
> str_subset(allstrings, pattern = "\\?") # Interpretation als R Befehl
Error [...]
> str_subset(allstrings, pattern = "\\?") # Interpretation als Zeichenfolge
[1] "Hallo?" "Good Day?"
>
>
> ## Wildcards
> str_subset(allstrings, pattern = "H.l")
[1] "Hallo?" "Hello" "H^llo"
> str_subset(allstrings, pattern = "Good ...")
[1] "Good Bye" "Good Day?"
```



```
> ### Zeichenklassendefinition
>
> # Suche nach "a oder e" nach "H" und vor "llo" :
> str_subset(allstrings, pattern = "H[ae]llo")
[1] "Hallo?" "Hello"
>
> ### Kontextabhaengigkeit von ^
>
> ### Nutzung ausserhalb Zeichenklassendefinition
> str_subset(allstrings, pattern = "^H") # Verweis auf Anfang
[1] "Hallo?" "Hello" "H^llo"
> str_subset(allstrings, pattern = "^B") # B steht nirgendwo am Anfang
character(0)
>
> ### Nutzung am Anfang einer Zeichenklassendefinition
>
> # Suche nach "nicht e" nach "H" und vor "llo" :
> str_subset(allstrings, pattern = "H[^e]llo")
[1] "Hallo?" "H^llo"
>
> ### sonstige Nutzung in einer Zeichenklassendefinition
>
> # Suche nach "e oder ^" nach "H" und vor "llo" :
> str_subset(allstrings, pattern = "H[e^]llo")
[1] "Hello" "H^llo"
```



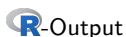
```
> ## Kontextabhaengigkeit von -
>
> ### Nutzung am Anfang/Ende einer Zeichenklassendefinition
>
> # Suche nach - oder Leerzeichen nach "Good" und vor "Bye":
> str_subset(allstrings, pattern = "Good[- ]Bye")
[1] "Good Bye" "Good-Bye"
>
> ### Nutzung in der Mitte einer Zeichenklassendefinition
>
> # Suche nach Zeichen zwischen a bis z nach "H" und vor "llo":
> str_subset(allstrings, pattern = "H[a-z]llo")
[1] "Hallo?" "Hello"
>
> # Suche nach Zeichen zwischen a bis d nach "H" und vor "llo":
> str_subset(allstrings, pattern = "H[a-d]llo")
[1] "Hallo?"
```

## Vordefinierte Zeichenklassen

Neben der manuellen Definition von Zeichenklassen existieren bereits **vordefinierte Klassen**, die sehr hilfreich sein können:

Code	Bedeutung	Beispiel	Äquivalent
<code>\d</code>	<b>digit</b>	eine Ziffer	<code>[0-9]</code>
<code>\D</code>	no <b>digit</b>	ein Zeichen, das keine Ziffer ist	<code>[^0-9]</code>
<code>\w</code>	word character	ein Buchstabe, eine Ziffer, Unterstrich	<code>[a-zA-Z_0-9]</code>
<code>\W</code>	no word character	Zeichen, das weder Buchstabe, Ziffer, Unterstrich ist	<code>[^\w]</code>
<code>\s</code>	whitespace	(meist) Leerzeichen und Klasse der Steuerzeichen <code>\f</code> , <code>\n</code> , <code>\r</code> , <code>\t</code> , <code>\v</code>	
<code>\S</code>	no whitespace	Zeichen, das weder Buchstabe, Ziffer, Unterstrich ist	<code>[^\s]</code>

# Vordefinierte Zeichenklassen in R



```
> ## Vordefinierte Zeichenklassen
>
> ### Beispiel 1
> str_subset(allstrings, pattern = "H\\wll")
[1] "Hallo?" "Hello"
> #### entspricht:
> str_subset(allstrings, pattern = "H[a-zA-Z_0-9]ll")
[1] "Hallo?" "Hello"
>
> ### Beispiel 2
> str_subset(allstrings, pattern = "Good\\sBye")
[1] "Good Bye"
```

# Wiederholungen

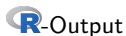
- ▶ weitere Verallgemeinerung durch **Quantoren**: definierte Ausdrücke können in einer bestimmten Häufigkeit in einer Zeichenfolge auftreten
- ▶ Kombination mit Zeichenklassen möglich

Code	Bedeutung
?	der vorausgehende Ausdruck ist optional: er kommt 0 oder 1 mal vor
+	der vorausgehende Ausdruck muss mindestens 1 mal vorkommen
*	der vorausgehende Ausdruck darf beliebig oft vorkommen (auch keinmal)
{n}	der vorausgehende Ausdruck muss exakt n mal vorkommen
{min,}	der vorausgehende Ausdruck muss mindestens min mal vorkommen
{min,max}	der vorausgehende Ausdruck muss mindestens min mal und darf höchstens max mal vorkommen
{0,max}	der vorausgehende Ausdruck darf höchstens max mal vorkommen

# Gruppierungen

- ▶ mit ( ) lassen sich Ausdrücke zusammenfassen und gruppieren
  - ▶ (abc){2} sucht z.B. nach Strings, die abc exakt zweimal hintereinander enthalten
  - ▶ (abc|xyz) sucht z.B. nach Strings, die abc oder xyz enthalten
- ▶ werden mehrere solcher Gruppen angegeben lassen sich diese mit \1 \2,... (je nachdem an welcher Stelle sie in dem RegEx Ausdruck definiert worden sind) referenzieren

# Wiederholung und Gruppierung in R



```
> ## Wiederholungen
>
> str_subset(allstrings, pattern = "o?") # tritt "o" an irgendeiner Stelle auf oder
    nicht
[1] "Hallo?" "Good Bye" "Good Day?" "Good-Bye" "Hello" "H^llo"
> str_subset(allstrings, pattern = "o{2}") # tritt "o" an irgendeiner Stelle zweimal
    auf
[1] "Good Bye" "Good Day?" "Good-Bye"
> str_subset(allstrings, pattern = "Go{2}d ")
[1] "Good Bye" "Good Day?"
>
> ## Gruppierungen
>
> allstrings2 <- c("abcabcabc", "abczabcabc", "abc abc abc2")
> str_subset(allstrings2, pattern = "(abc){3}")
[1] "abcabcabc"
> str_subset(allstrings2, pattern = "(abc.){3}")
[1] "abc abc abc2"
```

# RegEx in R

- ▶ es gibt weitere Metazeichen, die Suchmuster noch flexibler gestalten können
- ▶ ein weiteres nicht besprochenes Thema ist die Hierarchie in der Metazeichen zueinander stehen (precedence order) und muss im Zweifel beachtet werden
- ▶ ein sicherer Umgang mit RegEx erfordert viel Erfahrung und/oder Ausdauer
- ▶ **aber:** ist das Prinzip verstanden, lassen sich viele interessante Datensätze aufbereiten

## Beispiel:

- ▶ eine Datenaufbereitung mit RegEx ermöglicht z.B. die Erstellung einer Wordcloud
- ▶ um die Tweets welches Politikers handelt es sich wohl?







# #therealdonaldtrump

## ► Schritt 2: Anwendung zahlreicher RegEx Ausdrücke



```
# html Quelltext laden und Zeilenumbruch-Zeichen entfernen:
html <- read_file('../data/realdonaldtrump.txt')
html <- gsub('(?:\\r|\\n)', ' ', html)

# Text des Tweets extrahieren:
tweet_element_text <- unlist(
  str_extract_all(
    html, pattern='(?<=<p class=\\\"TweetTextSize TweetTextSize--normal js-tweet-text tweet-text\\\" lang
    =\\\"(?:en|et|und)\\\" data-aria-label-part=\\\"\\d\\\">).*?(?=</p>)'
  )
)
# Oberflächliche Säuberung: Entfernen aller html tags innerhalb des Tweets.
tweet_element_text <- gsub('<.*?>', ' ', tweet_element_text)
# html Block der Aktionen comment, retweet, like extrahieren:
tweet_element_stats <- unlist(
  str_extract_all(
    html, pattern='<div class=\\\"ProfileTweet-actionList js-actions\\\" role=\\\"group\\\" aria-label=\\\"Tweet
    actions\\\">.*?</li>'
  )
)
# Extrahieren der Interaktionszahlen:
tweet_element_stat_actions <- str_extract_all(
  tweet_element_stats, pattern='(?<=<span class=\\\"ProfileTweet-actionCount\\\" data-tweet-stat-count=\\\"
  )\\d+'
)
...

```


## ► Schritt 3: Analyse mit dem aufgeräumten Datensatz

### R-Output

```
> as_data_frame(tweets)[,c(1,3,4,5)]
# A tibble: 333 x 4
  time                retweets likes text
<dtm>                <dbl> <dbl> <chr>
1 2019-01-24 08:16:03  28041 119423 Nancy just said she ?just?
2 2019-01-24 05:37:59  23078 101707 Without a Wall there cann?
3 2019-01-24 05:34:26  15880 74922 ...back home where they b?
4 2019-01-24 05:21:59  17488 76183 The Fake News Media loves?
5 2019-01-24 04:48:32  19167 69012 So interesting that bad I?
6 2019-01-24 03:56:31  17405 78036 The economy is doing grea?
7 2019-01-24 03:51:52  13590 54911 ?This is everything FDR d?
8 2019-01-24 03:35:48  11850 58911 A great new book just out?
9 2019-01-23 20:18:30  17740 81409 ....alternative venue for?
10 2019-01-23 20:12:07  18644 84162 As the Shutdown was going?
# ... with 323 more rows
>
## Wordcloud
> tweets %>%
+ unnest_tokens(word, text) %>%
+ inner_join(
+   get_sentiments('bing')
+ ) %>%
+ count(word, sort=T) %>%
+ with(wordcloud::wordcloud(word, n, max.words=500))
```



# Inhalte

1. Grundlagen der Software 
2. Datenimport
3. Grundlagen der Datenaufbereitung und -bearbeitung
4. Fortgeschrittene Datenaufbereitung und -bearbeitung
5. Grafische Auswertung
6. Unstrukturierte Daten
- 7. Performance**
  - Grundlagen
  - Performanceprobleme
  - Lösungsansätze
8. Big Data

# Grundlagen

- ▶ Problem: R ist eine **keine schnelle Sprache**
- ▶ die Gründe sind vielfältig:
  1. R's dynamischer Aufbau
  2. Scoping Regeln
  3. Lazy Evaluation
- ▶ in dieser Kurseinheit
  - ▶ lernen wir Werkzeuge kennen um die Performance zu messen
  - ▶ identifizieren wir die größten Effizienzprobleme in R
  - ▶ wenden wir gängige Techniken an um die Auswirkungen der Effizienzprobleme zu reduzieren

## Grundlagen: Performancemessung

- ▶ die Funktion `microbenchmark()` im gleichnamigen Paket funktioniert wie ein Stoppuhr und misst die Ausführungszeit einer übergebenen Funktion
- ▶ diese Funktion wird mehrere Male ausgeführt und im Anschluss erhält man u.a. die mittlere Ausführungsdauer (systemabhängig!)
- ▶ R passt je nach Dauer die Zeiteinheiten automatisch an, wobei 1 Sekunde
  - ▶  $10^3$  Millisekunden (*ms*) ,
  - ▶  $10^6$  Mikrosekunden (*μs*) ,
  - ▶  $10^9$  Nanosekunden (*ns*) entspricht
- ▶ es gibt anspruchsvollere Techniken, die den Ressourcenverbrauch ermitteln (sog. Profiling)
- ▶ im folgenden Beispiel wird nochmals die Funktion `stetige_rendite()` betrachtet

# Grundlagen: Performancemessung

## R-Code

```
library(rio)
library(microbenchmark)
test_daten <- import(file = "../data/aktien.xlsx")

stetige_rendite <- function(zeitreihe) {
  T <- length(zeitreihe)

  ergebnis <- numeric(T - 1)
  for (i in 2:T) {
    ergebnis[i - 1] <- log(zeitreihe[i]) - log(zeitreihe[i - 1])
  }

  return(ergebnis)
}

microbenchmark(
  stetige_rendite(test_daten$google)
)
```

## R-Output

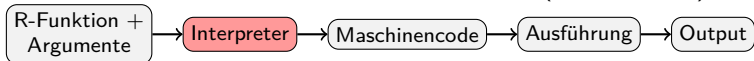
```
> microbenchmark(
+   stetige_rendite(test_daten$google)
+ )
Unit: microseconds
              expr   min      lq   mean  median    uq
stetige_rendite(test_daten$google) 60.841 61.2425 118.9791 61.49 61.905
```



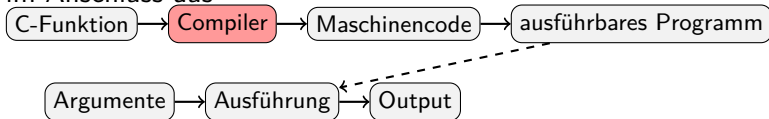
# Grundlagen: Übersetzung

der Compiler/ Interpreter übersetzt menschenlesbaren Code in maschinenlesbare Form (binärer Maschinencode aus 0 und 1)

- ▶ eine interpretierte Sprache (z.B. **R**) übersetzt **jede Zeile** in Maschinencode und führt diese unmittelbar (zur Laufzeit) aus



- ▶ eine kompilierte Sprache (z.B. C++) übersetzt erst das **gesamte Programm** in Maschinencode und führt den Code im Anschluss aus



- ▶ da der Übersetzungsvorgang in kompilierten Sprachen nur einmal stattfindet, sind diese in der Regel **deutlich schneller**

## Performanceprobleme: R's dynamischer Aufbau

- ▶ R ist eine dynamische Sprache: so gut wie alle Objekte sind nach ihrer Erstellung veränderbar → das geht auf Kosten der Performance jedes Interpreters, da die Unsicherheit über das Verhalten einer Funktion berücksichtigt werden muss
- ▶ **Beispiel:** R muss in `loop1` bei jeder Addition erneut den Typ feststellen um entscheiden zu können, was "+" bedeutet (z.B. ob ganze Zahlen oder Gleitkommazahlen addiert werden sollen, oder ob bei der Addition von Buchstaben eine Fehlermeldung ausgegeben werden soll)

### -Code

```
loop1 <- function(){  
  x<- 0L  
  for(i in 1:1e6){  
    x <- x + 1 # Typ von x unbekannt  
  }  
}
```

# Performanceprobleme: R's dynamischer Aufbau

- ▶ selbst wenn der Typ einer Variablen vorher festgelegt wird, bringt das keine Vorteile, da Änderungen jederzeit erlaubt sind
- ▶ so ist (im Gegensatz zu anderen Programmiersprachen) die folgende Zuweisung gültig:

-Code

```
x <- numeric(1) # Initialisierung von x als numeric
class(x) # liefert "numeric"
x[1] <- "test" # character Zuweisung, obwohl vorher als
               numeric deklariert
class(x) # liefert "character"
```

- ▶ trotz vorheriger Typfestlegung muss der R Interpreter vor jeder Operation den Typ erneut feststellen → **zeitintensiv!**

# Performanceprobleme: Scoping Regeln

```
f1 <- function(){  
  g1 <- function(){  
  
    print( a )  
  }  
  
  g1()  
}
```


```
f2 <- function(){  
  g2 <- function(){  
  
    print( a )  
  }  
  a <- 2  
  g2()  
}
```

```
f3 <- function(){  
  g3 <- function(){  
    a <- 3  
    print( a )  
  }  
  a <- 2  
  g3()  
}
```

- ▶ R arbeitet mit sog. **Umgebungen**: diese kann man sich als Container vorstellen, der die Namen von einer Menge von Objekten enthält
- ▶ Umgebungen stehen in einer hierarchischen Beziehung zueinander
- ▶ **Scoping Regeln**: legen fest in welcher Reihenfolge auf Objekte verschiedener Umgebungen zugegriffen wird

## Performanceprobleme: Scoping Regeln

- ▶ das obige Beispiel demonstriert die hierarchische Beziehung von Umgebungen:
  - ▶ die Funktion g erstellt bei Aufruf eine Umgebung
  - ▶ Elternumgebung von g ist f
  - ▶ Elternumgebung von f ist die globale Umgebung


-Output

```
> a <- 1
> f1()
[1] 1
> f2()
[1] 2
> f3()
[1] 3
```

- ▶ auch ohne die explizite Definition eigener Umgebungen arbeitet R mit Umgebungen um z.B. den Überblick zu behalten welche Funktion aus welchem Paket stammt

## Performanceprobleme: Scoping Regeln

- ▶ die Reihenfolge der Umgebungen in der Funktionen (oder andere Objekte) beim Aufruf gesucht werden nennt sich "searchpath"

 R-Output

```
> search()
[1] ".GlobalEnv"           "package:microbenchmark"
[3] "tools:rstudio"       "package:stats"
[5] "package:graphics"    "package:grDevices"
[7] "package:utils"       "package:datasets"
[9] "package:methods"     "Autoloads"
[11] "package:base"
```

- ▶ standardmäßig arbeitet man in der globalen Umgebung ".GlobalEnv", weshalb die Funktionen `f` aus dem Beispiel auch dort definiert sind
- ▶ viele häufig genutzte Funktionen wie `+`, `-` oder `[]` befinden sind in der Umgebung `base`

# Performanceprobleme: Scoping Regeln

## R-Output

```
> sum <- function(a,b) print("Nicht nachmachen")
> sum(1,2) # Aufruf aus .GlobalEnv
[1] "Nicht nachmachen"
>
> # die Standardfunktion existiert weiterhin:
> base::sum(1,2)
[1] 3
```

- ▶ das Beispiel verdeutlicht, wie R mit Umgebungen arbeitet:
  - ▶ die Definition von `sum` erfolgt in der globalen Umgebung
  - ▶ wenn die Funktion aufgerufen wird, sucht R als erstes in der globalen Umgebung und wird fündig
  - ▶ die Standardfunktion `sum` existiert weiterhin in der "base" Umgebung, die auf dem `searchpath` als letztes liegt
- ▶ Was hat das mit der Performance von R zu tun? Das Absuchen von Umgebungen nach Objekten kostet Zeit!


# Performanceprobleme: Scoping Regeln



```
f1 <- function(){  
  f2 <- function(){  
    f3 <- function(){  
      f4 <- function(){  
        rep(1+1,10000)  
      }  
  
      f4()  
    }  
  
    f3()  
  }  
  
  f2()  
}  
  
g1 <- function(){  
  rep(1+1,10000)  
}
```



# Performanceprobleme: Scoping Regeln

-Output

```
> microbenchmark(f1(),  
+                 g1(),  
+                 times=10000)  
Unit: microseconds  
expr   min    lq   mean  
f1()  11.358 14.066 42.41115 16.9135 18.660 ...  
g1()  10.476 13.014 30.75982 15.9560 17.573 ...
```

## Was passiert?

- ▶ + wird jedes mal in entlang des searchpath's in base gefunden

## Warum braucht f1 länger?

- ▶ in f1 liegen 3 Umgebungen mehr zwischen der aufrufenden und der base Umgebung

# Performanceprobleme: Lazy Evaluation

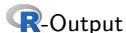
- ▶ Funktionsargumente werden in R nur ausgewertet, wenn sie tatsächlich benötigt werden ("lazy evaluation")

 R-Code

```
f <- function( a ) "Argument wird nicht ausgewertet"  
f( a= log(-2) ) # gibt keinen Fehler aus
```

- ▶ Damit das funktioniert wird bei jedem Funktionsaufruf ein Objekt erstellt, das alle übergebenen Argumente speichert und bei Bedarf auswertet → **zeitintensiv!**
- ▶ viele andere Programmiersprachen akzeptieren keine Argumente, die nicht benötigt werden

# Performanceprobleme: Lazy Evaluation



```
> f0 <- function( ) "kein Argument benötigt"
> f1 <- function( a ) "kein Argument benötigt"
> f2 <- function( a , b ) "kein Argument benötigt"
> f3 <- function( a , b , c ) "kein Argument benötigt"
>
> microbenchmark(
+   f0(),
+   f1(log(-2)),
+   f2(log(-2),log(-2)),
+   f3(log(-2),log(-2),log(-2)),
+   times= 10000
+ )
Unit: nanoseconds
```

	expr	min	lq	mean	
	f0()	115	127	205.9221	...
	f1(log(-2))	152	168	273.0264	...
	f2(log(-2), log(-2))	186	212	373.8053	...
	f3(log(-2), log(-2), log(-2))	228	265	425.1848	...

# Lösungsansätze

- ▶ Vermeidung der beschriebenen Performanceprobleme
  - ▶ **vektorierte Funktionen:** führen mehrfach Berechnungen für alle Elemente der übergebenen Vektoren durch (meistens deutlich effizienter)



```
mult_1 <- function(x, y) {  
  T <- length(x)  
  out <- numeric(T)  
  for (i in 1:T) {  
    out[i] <- x[i] * y[i]  
  }  
  return(out)  
}  
  
mult_2 <- function(x, y) {  
  out <- x * y # "*" ist vektorisiert  
  return(out)  
}  
  
microbenchmark(  
  mult_1(1:100, 1:100),  
  mult_2(1:100, 1:100)  
)
```

# Lösungsansätze

## R-Output

```
> microbenchmark(  
+   mult_1(1:100, 1:100),  
+   mult_2(1:100, 1:100)  
+ )  
Unit: microseconds  
      expr   min      lq   mean  median     uq   max neval  
mult_1(1:100, 1:100) 12.226 12.3940 65.72708 12.5360 12.8260 5281.075 100  
mult_2(1:100, 1:100)  1.420  1.4985 31.13495  1.6775  1.8015 2893.788 100
```

# Lösungsansätze

- ▶ **Verwendung einer kompilierten Sprache:** obwohl es sich mit R um eine interpretierte Sprache handelt, lässt sich mit dem Paket Rcpp die Performance von C++ nutzen
- ▶ **High Performance Computing:** z.B. an der [HHU](#) "*Unser aktuelles HPC-System "HILBERT" ist vor kurzem in Betrieb genommen worden und besteht aus einem Shared-Memory-Teil (SGI UV2000 mit 512 Cores und 16 TByte RAM) und einem MPI-Teil (Bull INCA mit 2688 Cores und 128 GByte Ram je Rechenknoten).*"

## Lösungsansätze: Rcpp

- ▶ für R existiert eine komfortable Anbindung an C++ : das Paket Rcpp
- ▶ Vorgehensweise: Schreibe eine C++ Funktion und Rcpp kümmert sich um die Kompilierung und Einbindung in R
- ▶ Vorteil: teilweise sehr **große Performancesteigerungen**
- ▶ im folgenden Beispiel werden wir die R Funktion `stetige_rendite` in C++ umsetzen

## Lösungsansätze: Rcpp

Definiere eine C++ Datei `stetige_rendite_3.cpp`

```
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector stetige_rendite_3(NumericVector x) {
    int T = x.size();

    NumericVector out(T-1) ; // gefuelllt mit 0

    for(int i=0;i<(T-1);i++){
        out[i] = log( x[i+1] ) - log( x[i] );
    }
    return out;
}
```



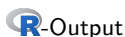
# Lösungsansätze: Rcpp

Vergleich mit anderen Implementierungen:



```
stetige_rendite_1 <- function(zeitreihe) {  
  T <- length(zeitreihe)  
  ergebnis <- numeric(T - 1)  
  # Performanceproblem Schleife:  
  for (i in 2:T) {  
    ergebnis[i - 1] <- log(zeitreihe[i]) - log(zeitreihe[i - 1])  
  }  
  return(ergebnis)  
}  
  
stetige_rendite_2 <- function(zeitreihe) {  
  T <- length(zeitreihe)  
  # Besser als vektorisierte Operation:  
  ergebnis <- log(zeitreihe[2:T]) - log(zeitreihe[1:(T - 1)])  
  return(ergebnis)  
}
```

# Lösungsansätze: Rcpp



```
> library(Rcpp)
> sourceCpp(file = "../misc/stetige_rendite_3.cpp")
>
> # Funktionen machen alle das gleiche
> stetige_rendite_1(test_daten$google)[1:4]
[1] NA 0.0009662933 0.0090074690 0.0151611841
> stetige_rendite_2(test_daten$google)[1:4]
[1] NA 0.0009662933 0.0090074690 0.0151611841
> stetige_rendite_3(test_daten$google)[1:4] # C++
[1] NA 0.0009662933 0.0090074690 0.0151611841
>
>
>
> microbenchmark(
+   stetige_rendite_1(test_daten$google),
+   stetige_rendite_2(test_daten$google),
+   stetige_rendite_3(test_daten$google)
+ )
Unit: microseconds
      expr   min    lq  mean median    uq
stetige_rendite_1(test_daten$google) 61.010 62.1155 70.60541 63.4120 66.5580
stetige_rendite_2(test_daten$google) 13.966 14.9175 20.07131 17.1905 20.8215
stetige_rendite_3(test_daten$google) 12.026 12.5460 25.60303 13.4695 15.9665
```

In diesem Fall liefert die C-Funktion kaum Performancegewinn!

## Warum?


- ▶ die vektorisierte Funktion \* ist selbst schon in C ausgelagert

# Inhalte

1. Grundlagen der Software 
2. Datenimport
3. Grundlagen der Datenaufbereitung und -bearbeitung
4. Fortgeschrittene Datenaufbereitung und -bearbeitung
5. Grafische Auswertung
6. Unstrukturierte Daten
7. Performance
- 8. Big Data**

# Herausforderung für große Datensätze

- ▶ **Ausgangspunkt:** Was ist bei folgender Fehlermeldung zu tun?

-Output

```
> bigData <- matrix(1.00, nrow= 10000, ncol = 1000000)
Error: vector memory exhausted (limit reached?)
```

- ▶ wir unterteilen Datensätze in folgende Kategorien:
  - ▶ Daten passen in den lokalen Arbeitsspeicher eines Systems
  - ▶ Daten passen auf die Festplatte eines Systems
  - ▶ Daten sind verteilt auf mehreren Systeme
- ▶ **“echte”** Big Data fallen überwiegend in die letzte Kategorie und werden häufig mit dem **3V Modell** definiert:
  - ▶ velocity: Daten müssen in Echtzeit analysiert werden
  - ▶ volume: große Datenmengen (Petabyte Bereich)
  - ▶ variety: Datenformen variieren (Videos, Fotos, Text,...)
- ▶ im folgenden gehen wir einfachheitshalber davon aus, dass alle Probleme für die der Speicher nicht ausreicht, **“big”** sind

# Architekturen

- ▶ je nach Kategorie sind unterschiedliche Architekturen nötig um mit Datensätzen arbeiten zu können
- ▶ **unser Fokus** in diesem Kapitel: die Datensätze passen auf die Festplatte, und es können relativ einfache Techniken auf dem lokalen System genutzt werden
- ▶ **fortgeschrittener** (und nicht weiter behandelte) Ansatz: Verarbeitung und Abfrage von großen Datenmengen, die verteilt in einem Netzwerk aus Systemen liegen, mit dem **Hadoop** Framework
- ▶ die Funktionen von Hadoop beruhen auf zwei wichtigen Bestandteilen:
  - ▶ Datenorganisation über mehrere Systeme (Hadoop Distributed File System)
  - ▶ Umsetzung des MapReduce Ansatzes, wobei Anfragen aufgeteilt und auf vielen Systemen parallel bearbeitet werden

# Lösung von Big Data Problemen

- ▶ die gute Nachricht: alle Techniken der vorherigen Kapitel lassen sich nach Vorarbeit genauso, oder in abgewandelter Form anwenden
- ▶ **möglicher Lösungsansatz von Big Data Problemen:** Aufteilung des “Big Data” Problems in viele “Small Data” Probleme
  - ▶ Hinweis: auch wenn dieses Vorgehen häufig möglich ist, gibt es Probleme bei denen keine Aufteilung möglich ist
- ▶ **Beispiel:** diese Technik wird nun an dem Datensatz `flightsText.csv` demonstriert
  - ▶ hierbei handelt es sich natürlich um “Small Data” (passt in den Speicher!)
  - ▶ das Vorgehen kann aber ohne weiteres auf echte “Big Data” Probleme übertragen werden

-Code


```
library(rio)
airlines <- import("../data/flightsText.csv")
```

# Arbeiten mit großen Datensätzen

- ▶ ressourcenschonende Herangehensweise:



- ▶ statt alle Daten auf einmal in den Speicher zu laden, werden kleinere Teile des Datensatzes nacheinander bearbeitet
- ▶ demonstriert wird das Vorgehen mit dem Mittelwert von `distance`:

-Output

```
> ## ein grosses Problem:  
> mean(airlines[,16]) # distance  
[1] 1039.913
```

# Arbeiten mit großen Datensätzen

## -Code

```
## viele kleine Probleme:
chunks <- 8 # Anzahl Teilprobleme
nrow(airlines)
chunksize <- nrow(airlines) / chunks # Beob. pro Teilproblem
chunk_means <- numeric(chunks)
for( i in 1:chunks ){
  lower <- (i-1) * chunksize + 1
  airlines_i <- import("../data/flightsText.csv", skip = lower, nrow = chunksize )
  chunk_means[i] <- mean(airlines_i[,16])
}
```

## -Output

```
> chunksize
[1] 42097
> chunk_means
[1] 1018.111 1046.508 1043.155 1009.660 1040.570 1052.221 1059.985 1049.091
> mean(chunk_means)
[1] 1039.913
```



# Arbeiten mit großen Datensätzen

- ▶ der Vorteil dieses Vorgehens besteht in der Bearbeitung eines Datensatzes, der nur **ein achtel** des Speichers verbraucht
- ▶ **flexibler Ansatz**: je nach Hardware kann die Aufteilung des Datensatzes auf beliebig viele Teildatensätze erfolgen
- ▶ der Speicherverbrauch kann mit dem Paket `profvis` untersucht werden:

## R-Code

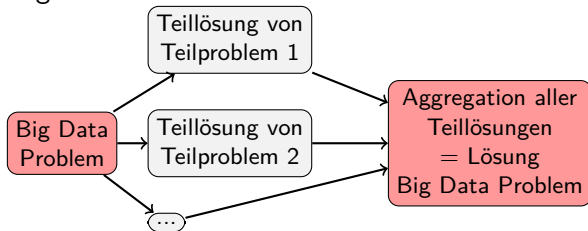
```
## Analyse der Performance mit profvis
library(profvis)
profvis({
  import("../data/flightsText.csv") #kompletter Datensatz
  import("../data/flightsText.csv", nrow = 42097) # 1/8 des Datensatzes
})
```

- ▶ der Speicherverbrauch ist deutlich geringer:

<expr>	Memory	Time
1 profvis({		
2   import("../data/flightsText.csv") #kompletter Datensatz	35.2	370
3   import("../data/flightsText.csv", nrow = 42097) # 1/8 des Datensatzes	3.9	100
4   })		
5 }		

# Arbeiten mit großen Datensätzen

- ▶ wenn mehrere Recheneinheiten zur Verfügung stehen, lässt sich die Rechenzeit verkürzen indem Teilprobleme **parallel** abgearbeitet werden:



- ▶ die Recheneinheiten können
  - ▶ lokal auf einem System vorhanden sein
  - ▶ verteilt über ein Netzwerk an Systemen sein

# Arbeiten mit großen Datensätzen

- ▶ Grundlagen des parallelen Rechnens in R:
  - ▶ Definition von Teilproblemen
  - ▶ Zuweisung von Teilproblemen zu Recheneinheiten
  - ▶ Prozessoren können mehrere Recheneinheiten (**Kerne**) beinhalten und Systeme können mehrere Prozessoren haben

## R-Output

```
> detectCores()  
[1] 4
```

- ▶ technische Details überlassen wir den Paketen `foreach` und `doParallel`

## R-Code

```
## paralleles Abarbeiten der Probleme  
library(foreach)  
library(doParallel)  
numCores <- detectCores() - 1  
  
# Initialisierung von Recheneinheiten zur  
# Bearbeitung von Teilproblemen:  
registerDoParallel(numCores)
```

# Arbeiten mit großen Datensätzen

## R-Code

```
chunk_means <-  
foreach(i = 1:chunks, .combine = rbind, .packages = "rio") %dopar% {  
  # Definition von Teilproblem:  
  chunksize <- 336776 / chunks # Beobachtungen pro Chunk  
  lower <- (i - 1) * chunksize + 1  
  airlines_i <- import("../data/flightsText.csv",  
    skip = lower, nrow = chunksize  
  )  
  mean(airlines_i[, 16])  
}  
  
# Freigabe der blockierten Recheneinheiten  
stopImplicitCluster()
```

## R-Output

```
> # Ergebnis:  
> chunks  
[1] 8  
> numCores  
[1] 3  
> c(chunk_means)  
[1] 1018.111 1046.508 1043.155 1009.660 1040.570 1052.221 1059.985 1049.091  
> mean(chunk_means)  
[1] 1039.913
```

# Arbeiten mit großen Datensätzen



```
## Performancecheck
library(microbenchmark)


bigdata <- function() {
  airlines <- import("../data/flightsText.csv")
  mean(airlines[, 16]) # distance
}

smalldata <- function(chunks) {
  numCores <- detectCores() - 1
  registerDoParallel(numCores)
  chunk_means <-
    foreach(i = 1:chunks, .combine = rbind, .packages = "rio") %dopar% {
      chunksize <- 336776 / chunks # Beobachtungen pro Chunk
      lower <- (i - 1) * chunksize + 1
      airlines_i <- import("../data/flightsText.csv",
        skip = lower, nrow = chunksize
      )
      mean(airlines_i[, 16])
    }
  stopImplicitCluster()
  return(mean(chunk_means))
}

microbenchmark(
  bigdata(),
  smalldata(4)
)
```

# Arbeiten mit großen Datensätzen

- ▶ Ergebnis:

 R-Output

```
> microbenchmark(  
+   bigdata(),  
+   smalldata(4)  
+ )  
Unit: milliseconds  
      expr      min       lq      mean  ...  
bigdata() 170.9806 181.2989 210.2912 ...  
smalldata(4) 198.4574 214.0963 246.9835 ...
```

- ▶ Warum ist die Funktion `smalldata()` nicht schneller?
  - ▶ zusätzlicher Rechenaufwand für die Aufteilung der Teilaufträge
  - ▶ zusätzlicher Rechenaufwand für das Zusammentragen der Teilergebnisse
  - ▶ die Funktion `import()` braucht auch für sehr kleine Datensätze relativ lange
- ▶ trotzdem schon das Vorgehen den Speicherverbrauch