Datenanalyse mit R
Workshop

Christopher Harms

Data Scientist

mail@christopherharms.de

Kurze Vorstellungsrunde

1. Wer bist du?
2. Hast du eigene Daten dabei?
3. Hast du schon Vorerfahrungen mit R, Python oder einer anderen Programmiersprache?
4. Was möchtest du im Workshop lernen?
5. Was sollte auf keinen Fall passieren?
6. Was wäre für dich ein super gelungener Workshop?

Überblick
Themen

- Was ist R?
- R-Grundlagen
- Daten laden und aufbereiten
- Projektorganisation
- Daten analysieren
- Datenaufbereitung und –transformation mit tidyverse
- Daten visualisieren
- Fortgeschrittenes

Was ist R ?

Was ist R?
SPSS vs. R

- Einfache Point & Click-Analysen
- Rudimentäre Syntax, um grafische Oberfläche zu steuern

- Code-basierte Analysen, d.h. Fokus auf geschriebene Anweisungen

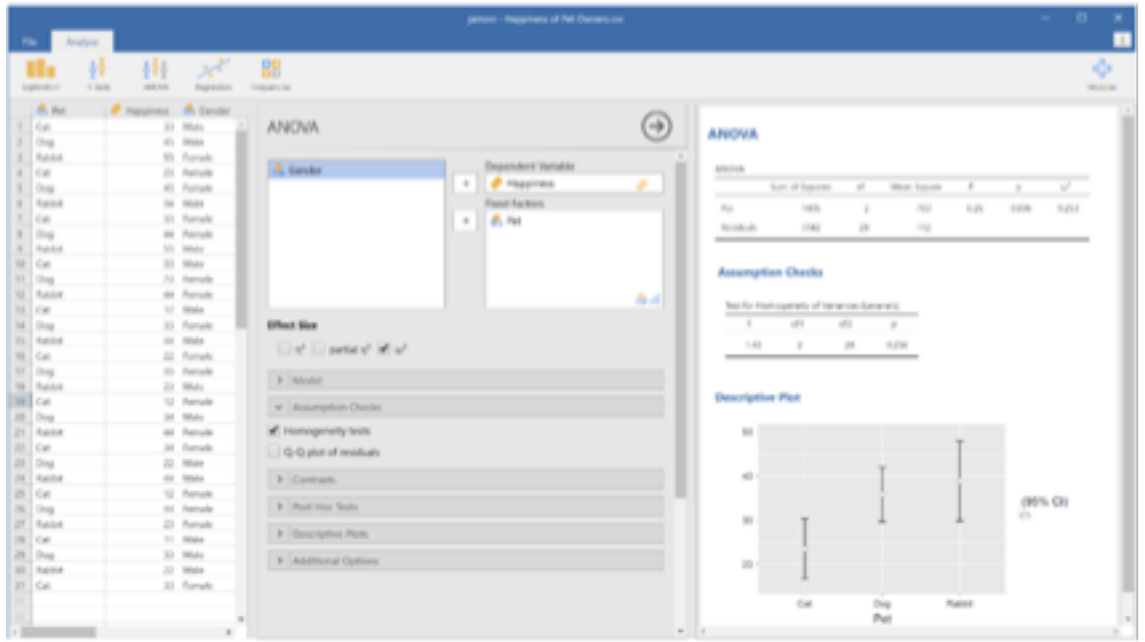
Was ist R?
R vs. RStudio

- Eig
  An
- Bie
- htt

- En
- Ma
- Vie
  Pro
- htt

Andere Alternativen zu R
jamovi

- jamovi ist ein freies und kostenloses Tool für statistische Analysen
- Nutzt intern auch R – und kann für alle Analysen entsprechenden R-Code generieren
- Einfacher in der Handhabung als R selber, aber entsprechend auch eingeschränkter
- Für die Praxis allerdings oft nützlich
- https://www.jamovi.org/



R is like a relationship

Quelle: https://bookdown.org/ndphillips/YaRrr/rrelationship.html

Was ist R?
Beispiel für einfaches R-Skript

```
# Rohdaten

group.1 <- c(10, 11, 12, 13)

group.2 <- c(2, 3, 4, 5)

# t-Test berechnen

t.test(group.1, group.2)

# Histogramm für alle Daten

hist(c(group.1, group.2))
```

Was ist R?
Vergleich mit SPSS

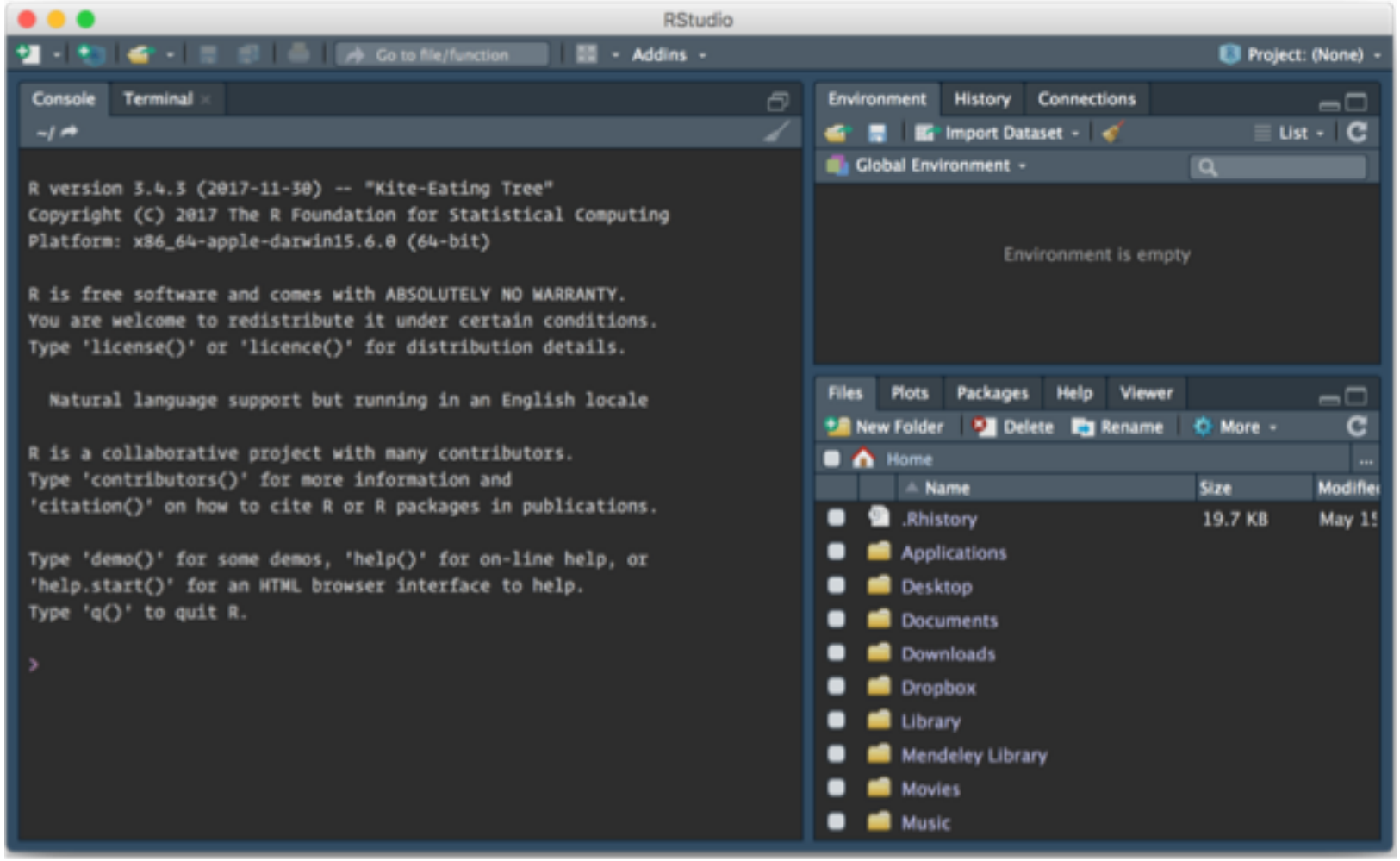
R-Skript

```
mean(df$Height)

sd(df$Height)

min(df$Height)

max(df$Height)
```

SPSS Syntax

```
DESCRIPTIVES VARIABLES= Height
   /STATISTICS=MEAN STDDEV MIN MAX.
```

Was ist R?
Arbeiten mit RStudio



Was ist R?
Zwei Zugänge zu R

R-Skript (z.B. ttest.R)

```
# Rohdaten

group.1 <- c(10, 11, 12, 13)

group.2 <- c(2, 3, 4, 5)

# t-Test berechnen

t.test(group.1, group.2)

# Histogramm für alle Daten

hist(c(group.1, group.2))
```

R-Konsole

```
> group.1 <- c(10, 11, 12, 13)

> group.2 <- c(2, 3, 4, 5)

> t.test(group.1, group.2)

> hist(c(group.1, group.2))
```

Wo findet man Hilfe?

- Interne Dokumentation für Befehle und Pakete

```
> ?ttest
> ??iqr
```

- Google
- stackoverflow https://stackoverflow.com

R-Grundlagen

R-Grundlagen
Anweisungen in der Konsole

```
> # 1+1

> 1+1

[1] 2

> 2*2

[1] 4

> 10^5

[1] 1e+05

> sqrt(9)

[1] 3
```

- Kommentare beginnen mit # und werden nicht ausgewertet
- Zeilen werden ausgewertet und das Ergebnis angezeigt

R-Grundlagen
Variablen

```
> a <- 1
> a
[1] 1
> 2*a
[1] 2
> b <- 2
> a+b
[1] 3
> pi
[1] 3.141593
```

- Wir können Daten in Variablen zwischenspeichern
- Der Operator „`<-`“ weißt der linken Seite den Wert auf der rechten Seite zu
- Variablennamen können aus Buchstaben, Zahlen, Unterstrichen und Punkten bestehen:
  - `variable.1`
  - `long_variable_name123`

R-Grundlagen
Variablen

- R ist im Allgemeinen nicht Typen-sensitiv, d.h. Variablen können ihren Typ wechseln
- Trotzdem ist es hilfreich, die unterschiedlichen Typen zu kennen und zu verstehen

Datentypen	Beispiel	Bezeichnung in R
Zeichenkette (String)	`"Hier steht Text"`	`character`
Ganzzahl (Integer)	`2`	`integer / numeric`
Fließkommazahl (Float)	`3.141`	`numeric`
Logische Werte (Boolean)	`TRUE`	`bool`
Vektor	`c(1, 2, 3)`	`vector`
Liste	`list(a = 1, b = 2)`	`list`
Data Frame		`data.frame`
Matrix		`matrix`

R-Grundlagen
Variablen

```
> var.1 <- 1
> var.2 <- 2
> var.3 <- "3"
> sum.of.vars <- var.1 + var.2
> sum.of.vars
[1] 3
> sum.of.vars <- sum.of.vars + var.3
Error: non-numeric argument to binary
operator
> sum.of.vars
[1] 3
```

- Zwar ist R nicht besonders streng hinsichtlich Datentypen, aber unterschiedliche Typen können trotzdem nicht ohne Weiteres kombiniert werden
- Enthalten Anweisungen Fehler, werden sie nicht ausgeführt, d.h. `sum.of.vars` wird nicht aktualisiert – und behält den vorherigen Wert bei
- Es gibt aber Funktionen, um Typen zu konvertieren, z.B. `as.integer` oder `as.character`

R-Grundlagen
Variablen

```
> ?as.integer

> as.integer("2")

> as.integer(2)

> as.integer("2+2")

> as.integer(3.141)

> ?as.character
```

Was passiert in den links angegebenen Beispielen?

R-Grundlagen
Funktionen

```
> sqrt(9)
[1] 3
> log(10)
[1] 2.302585
> double.it <- function(x) { 2 * x }
> double.it(5)
[1] 10
> add.it <- function(x, y) { x + y }
> add.it(1, 2)
[1] 3
```

- Funktionen sind Subroutinen, die die eigentliche Funktionalität enthalten
- Funktionen können einen oder mehrere Parameter haben
- Wir können auch eigene Funktionen definieren und ausführen

R-Grundlagen
Funktionen

```
> subtract.it <- function(x, y) { x - y }

> substract.it(10, 6)

[1] 4

> subtract.it(y = 10, x = 6)

[1] -4
```

- Die Parameter (Angaben in Klammern hinter dem Funktionsnamen) werden entweder über die richtige Reihenfolge angegeben – oder über den Namen des Parameters
- Insbesondere bei Funktionen mit sehr vielen Parametern, empfiehlt sich die Angabe über den Namen

R-Grundlagen
Matrizen, Tabellen und Data Frames

- Bisher haben wir nur einzelne Werte betrachtet (im Fall von Zahlen: „Skalare“)
- Meistens interessiert uns aber eine Menge von Zahlen
- R „denkt“ in mathematischen Objekten, d.h. Vektoren und Matrizen
- In anderen Programmiersprachen: Arrays

```
> c(1, 2, 3)
[1] 1 2 3
> # Vektoren sind immer eindimensional:
> c(c(1, 2, 3), 4, 5, 6)
[1] 1 2 3 4 5 6
```

Was passiert, wenn wir Vektoren miteinander verrechnen?

```
> c(1, 2, 3) + c(4, 5, 6)
> c(1, 2, 3) * 4
```

R-Grundlagen
Matrizen, Tabellen und Data Frames

- Vektoren sind praktisch, um mehrere Zahlen einzugeben
- Häufig wollen wir aber zwei-dimensionale Tabellen von Daten haben: Zeilen = Probanden, Spalten = Variablen
- Lösung: Matrizen und Data Frames
- Data Frames sind „beschriftete“ Matrizen

```
> matrix.1 <- matrix(c(1, 2, 3, 4, 5, 6),
  ncol = 3, byrow = TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> matrix.1[,1]
[1] 1 4
> matrix.1[1,]
[1] 1 2 3
> matrix.1[1,1]
[1] 1
```

R-Grundlagen
Matrizen, Tabellen und Data Frames

- Matrizen und Data Frames können ineinander überführt werden
- Im Gegensatz zu Matrizen können Data Frames Spalten- und Zeilennamen haben und zugewiesen bekommen

```
> as.data.frame(matrix.1)
     V1 V2 V3
1     1  2  3
2     4  5  6
> colnames(matrix.1)
NULL
> colnames(as.data.frame(matrix.1))
[1] "V1" "V2" "V3"
```

R-Grundlagen
Matrizen, Tabellen und Data Frames

```
> age <- c(18, 23, 22, 25, 26)
> group <- c(1, 0, 1, 1, 0)
> outcome <- c(4, 4, 3, 2, 1)
> df <- data.frame(age, group, outcome)
> df
  age group outcome
1  18     1       4
2  23     0       4
3  22     1       3
4  25     1       2
5  26     0       1
```

```
> colnames(df)
[1] "age" "group" "outcome"
> colnames(df) <- c("age", "exp.grp", "y")
> df
  age exp.grp y
1  18       1 4
2  23       0 4
3  22       1 3
4  25       1 2
5  26       0 1
> df$age
[1] 18 23 22 25 26
```

R-Grundlagen
Kontrollen mit if & else

```
a <- 1

if (a == 1) {

  message("a hat den Wert 1")

} else if (a == 2) {

  message("a hat den Wert 2")

} else {

  message("a hat einen anderen Wert")

}
```

- Die geschweiften Klammern (`{`) bilden einen Block
- Der erste Block wird ausgeführt, wenn der Ausdruck innerhalb von `if (<Ausdruck>)` wahr (`TRUE`) ist – der zweite, wenn der Ausdruck falsch (`{`) ist

R-Grundlagen
Crash-Kurs Aussagenlogik

- Bool'sche Werte sind immer wahr (`TRUE`) oder falsch (`FALSE`)
- Es gibt 3 Grund-Operationen, die für uns erstmal wichtig sind:

  AND (`&`), OR (`|`), NOT (`!`)
- Es lassen sich auch mehrere Operationen miteinander verknüpfen, z.B. `A | (!B & A)`

A	!A	B	A & B	A | B
TRUE	FALSE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	FALSE	FALSE

A	B	!B	!B & A	A | ...
TRUE	TRUE	FALSE	FALSE	TRUE
TRUE	FALSE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE

R-Grundlagen
Crash-Kurs Aussagenlogik

- Die Variablen `A` und `B` aus der vorherigen Folie sind Aussagen, die wahr oder falsch sind
- Um diese zu erhalten, brauchen wir weitere Operatoren, z.B.:
  - Gleichheit (`==`)
  - Ungleichheit (`!=`)
  - Größer / kleiner als (`>` und `<`)
  - Größer gleich / kleiner gleich (`>=` und `<=`)
  - Element in Menge (`%in%`)

```
> 2 > 3

[1] FALSE

> 2 == 2

[1] TRUE

> 3 != 2

[1] TRUE

> a <- 50

> a > 20 | a < 30

[1] TRUE
```

R-Grundlagen
Crash-Kurs Aussagenlogik

```
> !(A | B)

> (!A & !B)


> !(A & B)

> (!A | !B)
```

Erstelle Wahrheitstafeln für verschiedene Werte von A und B für die links stehenden Ausdrücke.

R-Grundlagen
Pakete

```
# Ein Paket installieren
> install.packages("ggplot2")


# Mehrere Pakete gleichzeitig installieren
> install.packages(c("psych", "tidyverse",
  "foreign"))
```

- Pakete sind das Herzstück von R: Sie enthalten Funktionen, die andere Entwickler für uns vorbereitet haben
- Beispiele:
  - `ggplot2`
  - `dplyr`
  - `psych`
- Pakete, die auf CRAN verfügbar sind, können einfach installiert werden

R-Grundlagen
Pakete

```
> # Paket laden
> library(dplyr)


> # Paket laden, dass es nicht gibt
> library(gibt.es.nicht)
Error: there is no package called
  'gibt.es.nicht'
```

- Installierte Pakete müssen, bevor ihre Funktionen genutzt werden können, erst geladen werden

Gut aufgepasst?

Worin liegt der Unterschied zwischen `"2.0"` und `2`?

Wie unterscheiden sich Vektoren, Matrizen und Data Frames?

Installiere die Pakete `ggplot2`, `tidyverse`, `foreign`, `psych`, `car`, `ez` und `MASS`!

- Die Skriptsprache in R ist sehr mächtig und mit wenigen Befehlen kann man sehr viel umsetzen. „Echte“ Programmiersprachen bestehen aus sehr ähnlichen grundlegenden Bausteinen.
- Es dauert ein wenig, bis man eine Intuition für Skriptsprachen entwickelt. Das ist völlig normal.
- Der Fokus der nächsten Abschnitte liegt auf der praktischen Anwendung.

Daten laden und aufbereiten

Typische Datenformate

CSV-Datei (Comma-separated Values, .csv)

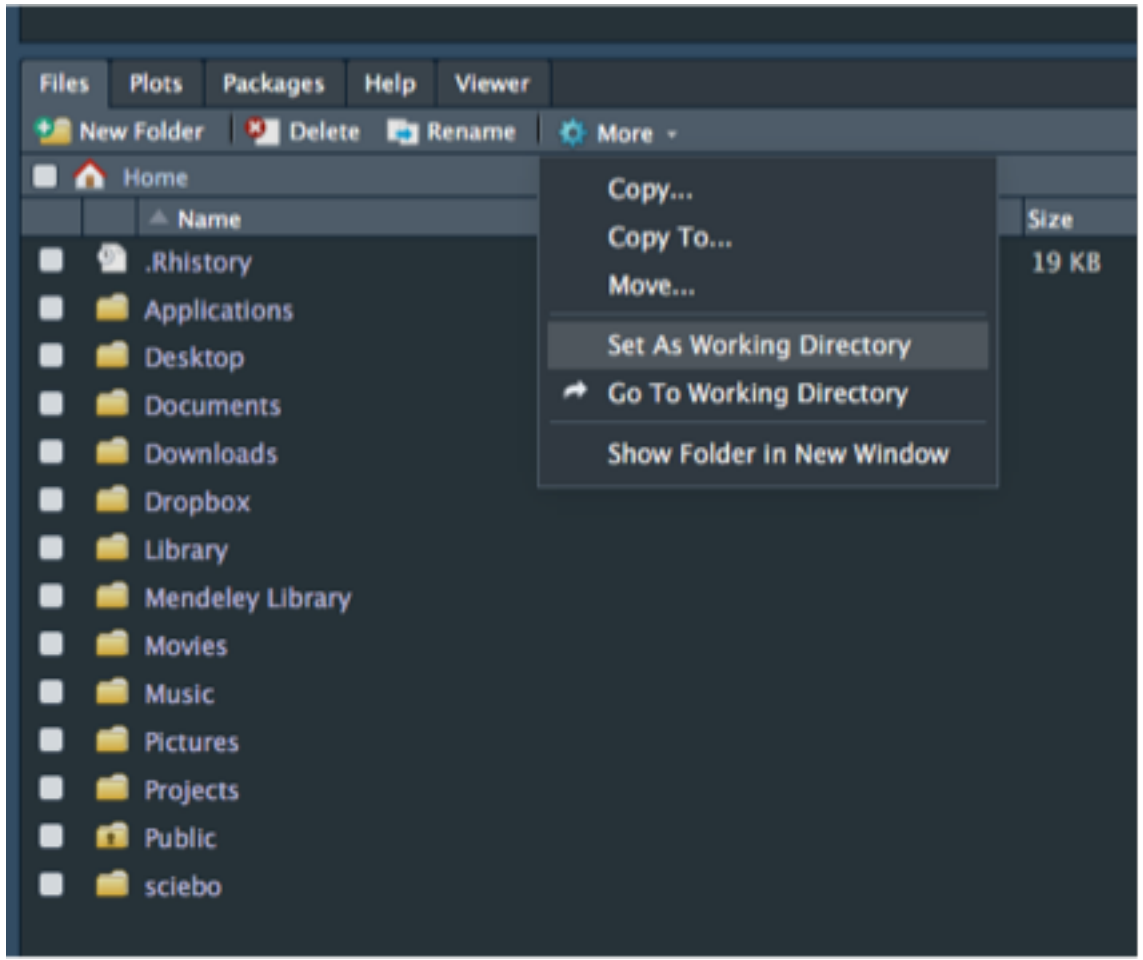
- Standard-Format zum Austausch von strukturierten Daten
- Wie eine Tabelle: Zellen sind durch Trennzeichen getrennt, meistens `,` (Komma) oder `;` (Semikolon)

```
lfdn;age;group;outcome
1;18;1;4
2;23;0;4
3;22;1;3
```

SPSS-Datei (proprietäres Format, .sav)

- Enthält neben den Daten noch Informationen über Werte- und Variablenlabels, Codierung für fehlende Werte, Skalenniveau
- Kann nur von SPSS zuverlässig gelesen werden – für R gibt es Pakete, die „ihr bestes geben“ (z.B. `foreign`)
- SPSS kann Datensätze auch nach .csv exportieren

Daten laden
Arbeitsverzeichnisse



› Relative Verzeichnisangaben (z.B. `./Daten.csv`) sind immer relativ zum Arbeitsverzeichnis. Dies kann entweder über RStudio ausgewählt werden – oder über die Konsole direkt angegeben werden:

```
> setwd("/Users/chrisha/Desktop")
> getwd()
[1] "/Users/chrisha/Desktop"
```

Daten laden
Einen CSV-Datensatz laden und inspizieren

- Legt das Arbeitsverzeichnis auf den Ordner, in dem ihr die Beispieldatensätze abgelegt habt
- Ladet die Datei `Big 5.csv` in das Objekt `df`
- Die Funktion `str()` zeigt die Struktur eines Objekts

Was sind das für Daten?

Wie kommen wir an die Werte der Variable `Extraversion`, d.h. den Spaltenvektor?

```
> setwd("~/sciebo/Lehre/R
  Workshop/Datasets")

> df <- read.csv("Big 5.csv")

> str(df)

# ...

> head(df)

# ...

> ncol(df)

# ...

> nrow(df)
```

Daten laden
Excel-Daten laden

- Excel trennt Spalten in einer CSV-Datei nicht mit Komma, sondern mit Semikolon
- `read.csv()` kann dann die Daten nicht ohne Weiteres lesen
- Mittels der Parameter für `read.csv()` können solche Einstellungen vorgenommen werden
- Oder man verwendet direkt `read.csv2()`

```
> ?read.csv

> read.csv("Primingstudie.csv")

# ...

> df <- read.csv("Primingstudie.csv", sep
  = ";")

> str(df)

# ...

> df <- read.csv("Primingstudie.csv", sep
  = ";", dec = ",“)

> df <- read.csv2("Primingstudie.csv")
```

Daten laden
SPSS-Daten laden

- Das proprietäre SPSS-Format kann nicht von R direkt geladen werden – dafür benötigen wir das Paket `foreign`
- Leider kann es hier auch immer wieder zu Problemen kommen (insb. bei Freitexten in den Daten) – deswegen ist CSV das präferierte Format
- Alternative: `spss.portable.file()` und `spss.system.file()` im Paket `memisc`

```
> library(foreign)

> ?read.spss

> df <- read.spss("Feeling Right – Final
  Dataset.sav", to.data.frame = TRUE)
```

Daten aufbereiten

- Daten, die wir aus Studien sammeln (z.B. in SoSciSurvey, Excel, SPSS) sind selten direkt für die Analyse bereit
- Wir haben fehlende Daten, brauchen neue Variablen, ggf. haben unterschiedliche Mitarbeiter unterschiedlich codiert, usw.
- Datenaufbereitung ist ein wichtiger und notwendiger Schritt in der Datenanalyse – auch wenn er sehr technisch wird
- **Vorteil von R:** Jeder Schritt ist transparent dokumentiert

Daten aufbereiten
Rechnen mit Variablen

- Wie ganz am Anfang schon gemacht, können wir mit Variablen rechnen
- So können wir neue Variablen berechnen und vorhandene Variablen recodieren
- Allgemein ist das Überschreiben von Spalten eher schlechter Stil: Die alten Daten gehen damit verloren

```
df <- read.csv2("Primingstudie.csv")

# Alter berechnen

df$Alter <- 2018 - df$Geburtsjahr

# Skalen-Antwort umkodieren (Achtung: Alte
  Daten gehen verloren!)

df$SE_IM_1.rec <- (5 + 1) - df$SE_IM_1

# Summenscore berechnen

df$SozErw.IM <- df$SE_IM_1.rec +
  df$SE_IM_2 + df$SE_IM_3
```

Daten aufbereiten
Bedingte Variablen-Zuordnung

- Mittels `ifelse()` können Variablenwerte unter bestimmten Bedingungen vergeben werden
- Auf diese Weise können auch Gruppen gebildet werden
- Im Paket `dplyr` gibt es die Funktion `case_when()`, die hier noch etwas flexibler eingesetzt werden kann (und lesbarer ist)

```
df <- read.csv2("Primingstudie.csv")


df$Geschlecht2 <- ifelse(df$Geschlecht ==
  "männlich", 1, 2)


df$Alter <- 2018 - df$Geburtsjahr

df$Altersgruppe <- ifelse(df$Alter < 18,
  "U18", ifelse(df$Alter > 60, "Ü60", "18
  - 60 Jahre"))
```

Daten aufbereiten
Bedingte Variablen-Zuordnung

- Mittels `ifelse()` können Variablenwerte unter bestimmten Bedingungen vergeben werden
- Auf diese Weise können auch Gruppen gebildet werden
- Im Paket `dplyr` gibt es die Funktion `case_when()`, die hier noch etwas flexibler eingesetzt werden kann (und lesbarer ist)

```
# Daten laden und Alter berechnen wie auf
  der vorherigen Folie

library(dplyr)

df$Altersgruppe <- case_when(
  df$Alter < 18 ~ "U18",
  df$Alter > 60 ~ "Ü60",
  TRUE ~ "18 – 60 Jahre"
)
```

Daten aufbereiten
Variablen umbenennen

- Viele unterschiedliche Wege:
  1. Neue Variable berechnen mit vorhandenem Wert (und anschließend Spalte entfernen)
  2. Alle Spalten auf einmal umbenennen
  3. Eine bestimmte Spalte umbenennen

```
df <- read.csv2("Primingstudie.csv")

# (1) Neue Variable:

df$Gruppe <- df$Priming

# (2) Spalten umbenennen über colnames:

colnames(df) <- c("id", "Geburtsjahr",
  "Geschlecht", "Gruppe", "Reaktionszeit",
  "Perceived.Close", "Perceived.Dist",
  "SE.IM.1", "SE.IM.2", "SE.IM.3",
  "SE.SD.1", "SE.SD.2", "SE.SD.3")

# (3) Eine einzelne Spalte umbennen:

colnames(df)[4] <- "Gruppe"
```

Daten aufbereiten
Spalten auswählen

- Data Frames sind wie Matrizen zwei-dimensionale Objekte
- Wir können jede individuelle Position in diesem zwei-dimensionalen Objekt ansprechen
- Über ein Vektoren können wir mehrere Zeilen/Spalten ansprechen
- Diese Vektoren können auch die Namen der Spalten enthalten

```
df <- read.csv2("Primingstudie.csv")

# Nur die erste Zeile:

df[1,]

# Nur die erste Spalte

df[,1]

# Erste Zeile, erste Spalte

df[1,1]

# Nur bestimmte Spalten über Nummer

df[,c(2:4, 6:ncol(df))]

# Bestimmte Spalten über Namen auswählen

df[,c("Geburtsjahr", "Geschlecht", “Priming")]
```

Daten aufbereiten
Zeilen auswählen / Filtern

- Zur Erinnerung: Wir haben mit Aussagenlogik „wahre“ und „falsche“ Aussagen geübt
- Mit „Wahr“ und „Falsch“ können wir R auch mitteilen, welche Zeilen (oder Spalten) wir gerne haben möchten

```
df <- read.csv2("Primingstudie.csv")

# Nur Pbn, die 1989 geboren sind

df[df$Geburtsjahr == 1989,]

# Nur männliche Teilnehmer

df[df$Geschlecht == "männlich",]
```

Wie funktioniert das genau? Tipp: Schaue dir an, was `df$Geschlecht == "männlich“` ausgibt!

Daten aufbereiten
Faktoren: nominale und ordinale Daten

- Beim Import legt R automatisch „Faktoren“ an, wenn in den Daten Zeichenketten vorkommen
- Oft ist das hilfreich, manchmal aber auch nicht (bspw. wenn eine Variable Freitexte enthält)
- Faktoren haben zwei Vorteile:
  - Teilen R mit, dass es sich um nominale Daten sind
  - Faktoren haben `levels` (1, 2, 3 …) und `labels` („Cognition“, „Feeling“, “Control“, …)

```
> df <- read.csv2("Primingstudie.csv")

> str(df)

'data.frame':  14 obs. of  6 variables:

 $ Probandennr.            : int  1 2 ...

 $ Geburtsjahr             : int  1988 1986
...

 $ Geschlecht              : Factor w/ 2
levels "männlich","weiblich": 1 2 ...

 $ Priming                 : 1 2 ...
```

Daten aufbereiten
Faktoren: nominale und ordinale Daten

- Die Spalte Priming soll eigentlich die Gruppenzugehörigkeit angeben – also auch ein Faktor sein
- Allerdings wurde sie nur numerisch mit 1 und 2 befüllt
- Wir können Zahlen in Faktoren umwandeln

```
> df <- read.csv2("Primingstudie.csv")

> df$Priming <- factor(df$Priming, labels
  = c(1, 2), levels = c("Feeling",
  "Cognition")

> as.numeric(df$Priming)

# ...

> as.character(df$Priming)

# ...
```

Daten aufbereiten
Fehler in Daten finden

- Codierfehler sind nervig und manchmal sehr schwer zu finden
- Eine erste explorative Analyse kann helfen, solche Ausreißer zu finden
- Wenn nicht eindeutig der korrekte Wert bekannt ist (z.B. Papierfragebogen), lieber auf `NA` setzen

Finde mittels der angegebenen Funktionen die fehlerhaften Codierungen in `Geburtsjahr`!

```
df <- read.csv2("Primingstudie.csv")

mean(df$Alter)

min(df$Geburtsjahr)

max(df$Geburtsjahr)

# Werte ersetzen

df[df$Geburtsjahr == 93,]$Geburtsjahr <-
  NA
```

Daten laden & aufbereiten

Lade deine eigenen Daten aus einer SPSS- oder CSV-Datei in ein `data.frame`

Filtere alle Zeilen heraus, in denen es fehlende Werte gibt! Tipp: `?complete.cases`

Survival Tipps!

1. Arbeitsverzeichnis richtig setzen!
2. Daten mit `read.csv` (Comma) oder `read.csv2` (Semikolon) einlesen?
3. SPSS-Daten mit Paket „`foreign`“ (Paket laden!) und Funktion „`read.spss`“ lesen!
4.
```
df <- read.spss(„dateiname.sav“,
to.data.frame = TRUE)
```

› RStudio hat „Import Dataset“ und kann damit auch grafisch die Daten laden – aber das kann ja jeder… ;-)

Projektorganisation

Exkurs - Projektorganisation

- Bei größeren Projekten empfiehlt es sich, Daten, Skripte usw. ordentlich aufzubewahren
- Drei Dinge sind dafür wichtig:
  - **Sauberer Code:** Hier helfen Styleguides (z.B. http://style.tidyverse.org)
  - **Kommentare:** Beschreiben was wir in unseren Skripten machen
  - **Hilfreiche Verzeichnisstruktur:** Hier gibt es kein richtig oder falsch

Beispiel für Verzeichnisstruktur:

- …/Masterarbeit/
  - Daten/
    - Rohdaten.sav
    - Arbeitsdaten_2018-06-06.csv
  - Analysen/
    - 0_Datenaufbereitung.R
    - 1_Datenanalyse.R
    - 2_Visualisierungen.R
  - Plots/
  - Manuskript/

Daten analysieren

Daten analysieren
Allgemeines

- Für die gängigen Testverfahren gibt es Funktionen in Basis R – ansonsten gibt es auch eine Vielzahl von Paketen für verschiedene, weitere Verfahren (Multilevel-Modelle, Faktorenanalysen, etc.)
- Nicht immer sind die Ergebnisse zu SPSS identisch – meistens ist das eine Frage von Standardeinstellungen, so dass mit ein bisschen Recherche die gleichen Ergebnisse erzielt werden können (siehe ANOVA)

Daten analysieren
Deskriptive Statistiken

- Funktionen für klassische deskriptive Statistiken sind recht einfach zu berechnen
- Die meisten Statistik-Funktionen in R funktionieren auf Basis von Vektoren (d.h. wir brauchen kein `data.frame`)
- Für den Modus gibt es keine Funktion – er kann aber aus der Häufigkeitstabelle ausgelesen werden

Berechne die deskriptiven Statistiken in einem Beispieldatensatz, z.B. `Primingstudie.csv`!

```r
data1 <- c(5, 3, 5, 6, 7, 9)

mean(df$Alter) # Mittelwert

median(df$Alter) # Median

quantile(data1) # Quantile

sd(data1) # Standardabweichung

var(data1) # Varianz

sd(data1)^2 == var(data1)

table(data1) # Häufigkeitstabelle
```

Daten analysieren
Korrelation: Pearson & Spearman

- Auch für bivariate Statistiken gibt es Funktionen
- Über die Parameter `method` können unterschiedliche Korrelationen berechnet werden – standardmäßig wird Pearson's Korrelation verwendet

```
var1 <- c(1, 2, 3, 4, 5)

var2 <- c(5, 4, 3, 2, 1)

cov(var1, var2) # Kovarianz

cor(var1, var2) # Korrelation

# Mit einem beliebigen data.frame

df <- data.frame(var1, var2)

cor(df$var1, df$var2)

cor(x = df$var1, y = df$var2, method =
  "spearman")
```

Daten analysieren
Korrelation: Pearson & Spearman

- `cor()` liefert die deskriptive Größe – häufig wollen wir aber auch wissen, ob die Korrelation 0 signifikant verschieden ist
- Dies liefert uns `cor.test()`
- Auch hier kann über `method` eine Rangkorrelation gewählt werden
- Über `alternative` kann die Alternativhypothese, also die Testrichtung angegeben werden

```
var1 <- c(1, 2, 3, 4, 5)

var2 <- c(5, 2, 3, 2, 1)

cor(var1, var2)

cor.test(var1, var2)

cor.test(var1, var2, alternative = "less",
  method = "spearman")

?cor.test
```

Daten analysieren
t-Test und Chi-Quadrat

- **Chi-Quadrat-Test:** Prüft, ob zwei Faktoren statistisch unabhängig voneinander sind

- **t-Test**: Testet, ob zwei (abhängige oder unabhängige) Stichproben gleiche Populations-Mittelwerte haben

```
df <- read.csv("Primingstudie.csv")

chisq.test(df$Priming, df$Geschlecht)



t.test(df[df$Priming == 1,]$SE_IM_1,
  df[df$Priming == 2,]$SE_IM_1, paired =
  FALSE, alternative = "two.sided")
```

Daten analysieren
Lineare Regression

- Lineare Regression ist eine einfache Funktion, ähnlich wie die Tests zuvor
- Für `lm()` wird jedoch die Formel-Notation verwendet, die in R in verschiedenen Kontexten genutzt wird

```
df <- read.csv("Primingstudie.csv")

lin.reg <- lm(Wahrgenommene.Distanz ~
  Priming + Geburtsjahr, data = df)

lin.reg

summary(lin.reg)

plot(lin.reg)
```

Daten analysieren
Varianzanalyse: One-way ANOVA

- Spezialfall der linearen Regression (bzw. des linearen Modells)
- Verschiedene Pakete bieten verschiedene Funktionen:
  - `car::Anova()`
  - `aov()`
  - `ez::ezANOVA()` ist am einfachsten zu verwenden

```
library(ez)

ezANOVA(data = data.frame(iris, subnum =
  1:nrow(iris)),

  between = Species,

  dv = Petal.Length,

  wid = subnum)
```

Daten analysieren
Varianzanalyse: Two-way ANOVA

- R und SPSS liefern unterschiedliche Ergebnisse für die gleichen Daten bei mehr als zwei Blockfaktoren
- Grund: es werden unterschiedliche Typen von Quadratsummen berechnet
- Wir können R aber dazu bringen, das gleiche Ergebnisse wie SPSS auszugeben
- Welches „Ergebnis“ richtig ist, hängt eigentlich von den Annahmen und den Daten ab – dies wird aber selten (nie!) berücksichtigt

```
# Alle Blockvariablen als Faktoren:
> df$priming <- as.factor(df$priming)
> df$gender <- as.factor(df$gender)
# Kontraste umstellen
> options(contrasts = c("contr.helmert",
  "contr.poly"))
# Typ-3-Quadratsummen berechnen
> library(car)
> lin.model <- lm(outcome ~ priming *
  gender, data = df)
> Anova(lin.model, type = 3)
```

Daten laden & aufbereiten

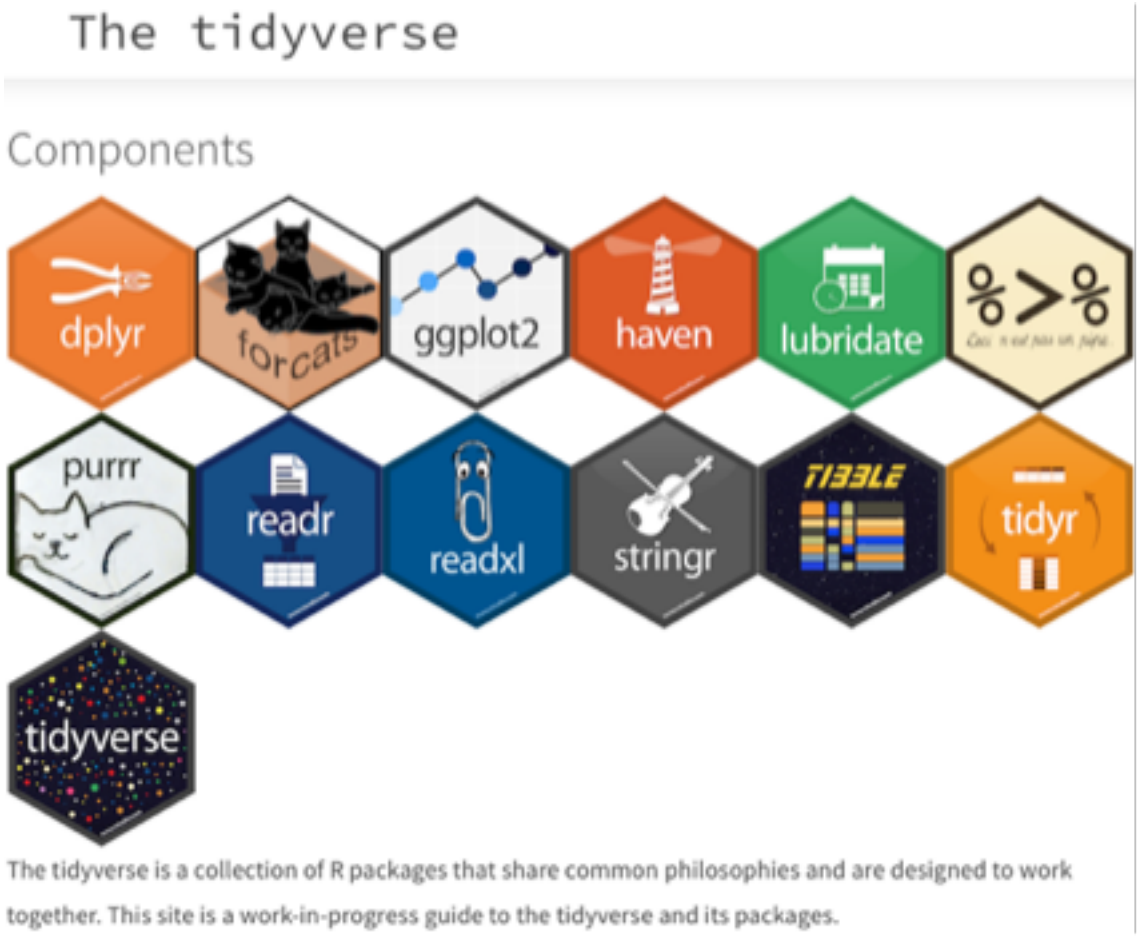
Wertet eure eigene Daten aus – sofern sie einfach genug strukturiert sind! (Alternativ: `cars`, `mtcars`, `iris`)

Welche Probleme kann es geben, wenn wir Daten analysieren wollen?

Datenaufbereitung und –transformation mit `tidyverse`

Datentransformation mit tidyverse

- Das `tidyverse` ist eine Sammlung verschiedener Pakete, um saubere Datentransformationen durchzuführen
- Ermöglicht vieles auf schnellere Weise als Basis R
- https://www.tidyverse.org/packages/



Tidyverse-Pakete installieren und laden

```
> install.packages("tidyverse")
> library(tidyverse)
-- Attaching packages ------------------------------------ tidyverse 1.2.1 --
v ggplot2 2.2.1     v purrr   0.2.5
v tibble  1.4.2     v dplyr   0.7.5
v tidyr   0.8.1     v stringr 1.3.1
v readr   1.1.1     v forcats 0.3.0
-- Conflicts --------------------------------------- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

Was ist „tidy data“?

- „Daten“ müssen nicht immer strukturiert wie in einer SPSS-Datei vorliegen
- „Daten“ sind Daten, wenn sie Informationen enthalten, unabhängig von ihrer Struktur
- Das heißt, dass wir immer unterschiedliche Möglichkeiten haben, Daten darzustellen

```
{
  "birthyears": [
     { "id": 1, "data": 1990 },
     { "id": 2, "data": 1996 }
  ],
  "additional_data": {
     "id_1": [ "male", "student" ],
     "id_2": [ "female", "taxi driver" ]
  }
}
```

Was ist „tidy data“?
Zwei Arten Daten darzustellen

Wide Format

ID	Age	Gender	Group
1	26	male	Contr
2	22	female	Exp
3	31	male	Contr

Long Format

ID	Name / Key	Value
1	Age	26
1	Gender	male
1	Group	Contr
2	Age	22
2	Gender	female
2	Group	Exp
3	Age	31
3	Gender	male
3	Group	Contr

Was ist „tidy data“?

- Kein Format ist prinzipiell „gut“ oder „schlecht“ – es kommt auf den Einsatzzweck an
- Guter Mittelweg – sowohl für Speicherung als auch für Verarbeitung – sind „saubere“ (tidy) Daten
- Daten sind „sauber“, wenn …
  - jede Variable in einer eigenen Spalte steht,
  - jede Beobachtung einer Einheit eine Zeile ist, und
  - jede Einheit eine eigene Tabelle ist.

Wickham, H. (2014). Tidy Data. *Journal of Statistical Software, 59*(10), 1 - 23. doi:http://dx.doi.org/10.18637/jss.v059.i10

ID	Time	Age	Outcome
1	0	21	5
1	1	21	7
1	2	21	6
1	3	21	8
2	0	33	3
2	1	33	4
2	2	33	5

Es geht nicht um die „perfekte“ Struktur, aber um pragmatische und nützliche Lösung

Nützliche Funktionen im tidyverse

- `readxl`: Excel-Daten einlesen
- `haven`: SPSS- und SAS-Daten einlesen (auch mit UTF-8-Unterstützung)
- `DBI`: Daten aus Datenbanken lesen (MySQL, PostgreSQL etc.)
- `forcats`: kategoriale und ordinale Daten besser handhaben

- Statt `data.frames` verwendet das tidyverse sogenannte `tibble` (kurz: `tbl`) – können wie `data.frames` verwendet werden, sind aber bisweilen etwas nützlicher
- Die meisten Transformationen können über „Pipes“ durchgeführt werden: `%>%` leitet Daten von links nach rechts weiter
- Das ist etwas ungewohnt, an vielen Stellen aber hilfreich, um Anweisungen zu Blöcken zu bündeln