Vorwort

2022 Edition (mit Glitzer (metaphorisch)) — Jetzt in Farbe!

Tobi & Lukas

May 8, 2022

Table of contents

1 Vorwort **5**
Über diese Einführung . . . 5
Zielgruppe . . . 6

I Vorbereitung **7**

2 Was ist R **8**
2.1 Was ist an R so toll? . . . 8
2.2 Was ist R **nicht**? . . . 9
2.3 Wieso nicht einfach SPSS? . . . 10

3 Installation **12**
3.1 R installieren . . . 12
3.1.1 Windows . . . 12
3.1.2 Mac OS X (macOS) . . . 14
3.1.3 Linux . . . 15
3.2 RStudio installieren . . . 15
3.3 RStudio benutzen . . . 16

4 Orientierung **18**
4.1 RStudio . . . 18
4.2 Konsole . . . 21
4.2.1 Gängige Probleme . . . 22
4.3 Text Editor . . . 23
4.4 Und das da rechts? . . . 26

II Grundlagen **27**

5 Erste Schritte **28**
5.1 Grundfunktionen . . . 28
5.1.1 Funktionsbeispiele . . . 30
5.2 Variablen . . . 32
5.3 Tabellen . . . 35
5.4 Umgang mit Datensätzen . . . 37

5.5 Logische Vergleiche . . . 40
5.5.1 Operatoren . . . 42
5.5.2 Spezielle Tests . . . 43
5.5.3 Indexing: Beispiele . . . 44

6 Datentypen **46**
6.1 Numeric (Zahlen und so) . . . 47
6.2 Character (Buchstabenzeugs) . . . 50
6.3 Factor (Here be dragons) . . . 52
6.4 Besondere Typen . . . 53
6.4.1 Fehlende Werte: `NA` . . . 53
6.4.2 Leere Werte: `NULL` . . . 54
6.4.3 To `Inf` and Beyo`NaN`d! . . . 54
6.5 Tabellen: `data.frame` . . . 56
6.6 Prüfen & Konvertieren . . . 57

7 Packages **58**
7.1 Installieren, Laden, Updaten . . . 58
7.1.1 Quellen . . . 60
7.1.2 Maintenance . . . 60
7.2 Das `tidyverse` . . . 62
7.3 Die `tadaatoolbox` . . . 62
7.4 Addendum: "Installier mal alles" . . . 63

8 Die Hilfe **64**
8.1 Format . . . 64
8.2 Sprache . . . 65

9 Datenimport **66**
9.1 Quellen . . . 66
9.1.1 Roher Text (`.csv`, `.txt`) . . . 68
9.1.2 SPSS (`.sav`) . . . 69
9.1.3 R (`.rds`, `.rda` & `.RData`) . . . 69
9.1.4 Excel (`.xlsx`) . . . 71
9.1.5 Google Sheets . . . 71
9.2 Daten angucken & sauber machen . . . 72

Quellen **73**

Appendices **73**

A Ressourcen **74**
A.1 Datensätze . . . 74
A.1.1 In R / Packages . . . 74
A.1.2 In der freien Wildbahn . . . 74
A.1.3 Tadaa, Datasets! . . . 74
A.2 Bücher . . . 75
A.2.1 Kollaboration und Organisation . . . 75
A.3 Kurse & Workshops . . . 75
A.3.1 Kostenlose Tutorials/Videos . . . 75
A.4 Blogs . . . 75
A.5 Community . . . 75
A.6 Dokumentation . . . 76

1 Vorwort

Das hier ist/wird/war/wurde [1] eine R-Einführung für Psychologiestudierende so mit ganz ohne Vorwissen im Bereich Software oder Programmierung. Dieses Werk ~~ist~~ war zum Einen die Grundlage für die R-Intro für Psychologiestudierende in *Quantitative Methoden*, und zum Anderen auch das Referenzwerk für das R-Seminar im Rahmen des Statistikmoduls in Public Health [2].

Die grobe Idee war es, eine Einstiegspräsentation für die Übung zu haben, und zusätzlich eine long-form Intro als Nachschlagwerk und vertiefendes Material (das hier).
Beides sollte ausreichend mit Screenshots/Beispielen ausgefüttert sein um auch die verunsichertesten Neulinge bei der Stange zu halten.
Da wir aber mittlerweile auch so gar nichts mehr mit der Pszchologie an der Uni Bremen zu tun haben und mit anderem Kram beschäftigt waren und sind, ist das ganze Projekt etwas versandet, verstaubt, in die Jahre gekommen, aber aus nicht mehr nachvollziehbaren Gründen irgendwie dennoch an ausreichend vielen Stellen hängen geblieben, wo Leute das tatsächlich als ergänzendes Lehrmaterial verwenden.

Über diese Einführung

Tobi und Lukas haben ~~zwei drei~~ ein paar Jahre lang die R-Einführung in QM gehalten, anfangs basierend auf den Folien, die Lukas unter Anderem auch als Nachschlagwerk gedacht hatte.
Weil Lukas aber ein gigantischer Nerd mit viel zu viel Spaß an Zeug ist, und sich sonst niemand für technische Details interessiert, brauchte er für das dritte Jahr die Hilfe von Tobi und einer Hand voll TutorInnen um die Einführung menschenkompatibel zu halten.
Das Resultat ist das hier.

Wenn ihr mehr (oder so grob dasselbe, aber deutlich mehr und in besser erklärt) wollt, dann empfehle ich herzlichst "**R for Data Science**", ein kostenloses Buch von Hadley Wickham, einem der R-Menschen, die uns in den letzten Jahren viele nette tools beschert haben[3].

[1] time is a flat circle
[2] zumindest dieses eine Mal in 2018
[3] Allen voran ggplot2, *das* package für plots in R.

Figure 1.1: Lukas & Tobi, weil richtige Portraits zu professionell wären.

Zielgruppe

Die primäre Motivation und Zielgruppe für diese Einführung waren die Studierenden der Psychologie an der Universität Bremen, weshalb hier auch möglichst wenig Vorkenntnisse im Bereich der Statistik, Programmierung, statistischen Programmierung oder programmierenden Sta… ähm… ihr versteht schon.
Wir gehen anfangs sehr kleinschrittig vor und versuchen gängige Konzepte wie Datentypen/-klassen, Datenstrukturen, Syntax von Grund auf zu erklären.

Solltet ihr bei Begriffen wie "funktionale Programmiersprache", "generics", "functionals" und "Attribute" ein grobes Bild im Kopf haben, dann empfehle ich euch einen Blick in das weiter oben erwähnte **R for Data Science**. Also generell würde ich allen interessierten dieses Buch/Seite empfehlen, aber ich vermute, dass es für einen absoluten Sprung ins kalte Wasser R nur dann geeignet ist, wenn man entweder sehr viel Motivation oder ein entsprechendes Vorwissen mitbringt.

Part I

Vorbereitung

2 Was ist R

R ist eine Programmiersprache, und ja, das klingt abschreckend.
Für uns heißt das in erster Linie Folgendes:

- Die Lernkurve ist etwas steil, flacht dann aber schnell ab. Versprochen.
- Wir können prinzipiell *alles* damit machen, wenn wir erstmal rausgefunden haben wie's geht.

2.1 Was ist an R so toll?

Nicht alles an R ist toll — und sobald ihr das erste Mal mehrere Stunden damit verbracht habt, ein vergleichsweise einfaches Problem lösen zu wollen, werdet ihr wissen was ich meine.

Die am häufigsten zitierten Vorteile sind in etwa die Folgenden:

- **Kostenlos**
  - Keine Lizenzgebühren
  - Keine Update-Gebühren
- **Open Source**
  - Software-Sprech für "jeder kann sich die Innereien angucken"
  - Jeder kann R beliebig erweitern und modifizieren
  - "Was genau macht R an Stelle XYZ" ist immer beantwortbar, weil der Quelltext offen verfügbar ist
- **Erweiterbarkeit**
  - Es gibt unzählige Erweiterungen (*"packages"*) für R, die neue Funktionen bereitstellen
  - Übersicht auf z.B. r-pkg.org
  - Sogar eure TutorInnen können packages schreiben. Tun sie sogar manchmal.
- **Gute Dokumentation** (meistens)
  - R bringt seine Hilfe selber mit, das ist der *"Help"*-Tab in RStudio. Jede Funktion ist dokumentiert!

  - Seiten wie rdocumentation.org oder rdrr.io erlauben es so gut wie alle verfügbaren R packages nach bestimmten Funktionen zu durchsuchen, egal wie obskur der Anwendungsfall auch sein mag
  - Communities wie stackoverflow sind beliebte "Ich brauche Hilfe bei XY"-Anlaufstellen, und eine Horde von Menschen, die meistens das gleiche Problem auch schonmal hatten, können euch helfen

- **Interaktivität**: Wo SPSS auf Buttons und Befehle reagiert und dann einen Stapel PDFs produziert, kann R mit der Konsole auch einfach schnell und interaktiv benutzt werden: "Schnell mal eben was nachgucken" ist in R deutlich angenehmer als in SPSS, zumindest für einfache Sachen.
- **Reproduzierbarkeit**: Wenn ihr ein sauberes R-Script geschrieben habt, könnt ihr das frei verfügbar machen und alle Interessierten können es bei sich selbst ausführen und *BÄM!*, Ergebnisse reproduziert. In case you didn't realize, but that's kind of a big thing in science.
- **Visualisierung**: R hat fantastische Werkzeuge zur Datenvisualisierung, primär sei hier *ggplot2* erwähnt, was euch erlaubt wunderschöne Grafiken aus euren Daten zu zaubern, und auch das vollkommen reproduzierbar (versucht das mal mit SPSS oder Excel. Worlds of pain.)
- **Flexibilität**: R ist eine Programmiersprache, das heißt per default kann R erstmal *alles*. Vielleicht nicht alles gut, aber prinzipiell lässt sich zumindest theoretisch alles damit anstellen. Dieses Dokument hier? In R (RMarkdown, *bookdown*) geschrieben. Dynamische Webseiten? Auch möglich, googlet "R shiny". Interaktive Elemente? Auch kein Ding, googlet "R htmlwidgets". Daten aus Software wie SPSS, Excel oder sogar Google Sheets importieren? Auch kein Problem. You get the idea.
- **Aktive Entwicklung & Community**: R wird *immer besser*. Immer mehr Leute benutzen R, von diesen Leuten haben einige gute Ideen, und davon wiederum gibt es einige, die diese sogar umsetzen. Das Resultat ist ein stetig wachsendes und gedeihendes Ökosystem um R und eine aktive Community auf diversen Netzwerken, die sowohl Hilfestellung liefern können als auch aktiv an besseren Tools rund um R arbeiten, von besserer Dokumentation bis zu besserer Integration mit anderer Software.

2.2 Was ist R nicht?

R ist nicht wie SPSS. R "nackt" ist ein Kommandozeilenprogramm. Terminal/Konsole, wie man es auch nennen mag, aber es ist nicht wie die Programme, die ihr im Alltag benutzt. Es gibt keine Knöpfe zum drücken und alles ist Text, ohne Formatierungskram wie **fett** oder *kursiv*, weil die Grundlagen bzw. Vorstufen von R aus einer Zeit kommen, in der Computer noch *anders* waren und die Mauer noch frisch war.

Weil das wenig benutzerfreundlich bzw. einsteigerfreundlich ist, gibt es Programme wie **RStudio**, was sozusagen auf R sitzt und es benutzbar macht. Wenn R ein Pferd ist, ist RStudio

ein Sattel.
Das heißt für euch aber auch, dass ihr eine Datenanalyse nicht durch bloßes Knöpfchendrücken zusammenstecken könnt wie das bei SPSS möglich ist — ihr müsst Befehle lernen, eure Daten(struktur) verstehen, und euch sicher sein, was ihr tun wollt.
Ich weiß, dass das alles wirklich eher krampfig klingt wenn ihr das so lest, weshalb es auch häufig vorkommt, dass Studierende in den ersten Semestern R eher ignorieren und sich lieber mit SPSS beschäftigen, weil das weniger kognitiver Aufwand ist — aber glaubt mir wenn ich euch sage, dass "R lernen" eine gute Investition ist. SPSS könnt ihr euch immer noch irgendwie beibringen oder Youtube-Tutorials gucken oder Befehlslisten ausdrucken und sowieso und überhaupt, R hingegen ist entsprechend komplexer, also nutzt die Zeit in der euch TutorInnen noch helfen können, bevor ihr dann vor eurem Expra-Datensatz sitzt, mit SPSS nicht weiter kommt und wie der Ochs vor'm Berg an R sitzt.

Ich betone das übrigens nicht, weil ich SPSS doof finde (was ich tue), oder R so toll finde (was ich tue), sondern tatsächlich aus Erfahrung und Überzeugung.
Wenn euch Statistik egal ist und ihr einfach nur bestehen wollt, dann reicht euch vielleicht SPSS. Wenn ihr langfristig gut mit Methodik und quantitativen Methoden arbeiten können wollt, dann ist R eine gute Investition.

2.3 Wieso nicht einfach SPSS?

SPSS ist viel.
SPSS kann viel.
SPSS kostet unsäglich viel.
SPSS nimmt euch das Denken ab — im guten wie im schlechten Sinn.

Wenn ihr SPSS bedienen könnt, ihr wisst was ihr tut und entweder ihr oder eure ArbeitgeberIn tief in die Tasche gegriffen haben um eine Lizenz bereitzustellen, dann ist das schön und gut, und ihr könnt den Knopf mit *"Mach mal Statistik"* drücken und dann kommen da PDFs raus wo Statistik drinsteht.
Schön.

Möglicherweise habt ihr aber nicht die finanziellen Ressourcen für SPSS.
Möglicherweise braucht ihr mehr Flexibilität.
Vielleicht funktioniert SPSS für euch nur mittels einer Remotedesktop-Verbindung (*hust hust*), das heißt sobald ihr irgendwo ohne schnelle Verbindung seid, könnt ihr nicht mehr arbeiten.
So sicher wie der Tod und Steuern müsst ihr auch irgendwann ein Expra durchführen, die erhobenen Daten auswerten und darstellen. Solltet ihr das mit SPSS machen, dürft ihr keine der Grafiken oder Tabellen benutzen (außer ihr verzichtet freiwillig auf eine gute Note). Das gleiche gilt in schwächerer Gewichtung für euer Differentielle Projekt und natürlich ggf. in stärkerer Gewichtung für eure Bachelorarbeit.

Vielleicht gefällt euch aber auch nur, dass ihr mit R interaktiv und schnell[1] einfache Dinge ausprobieren könnt.

[1] Das mit der Schnelligkeit kommt mit etwas Übung. Wirklich.

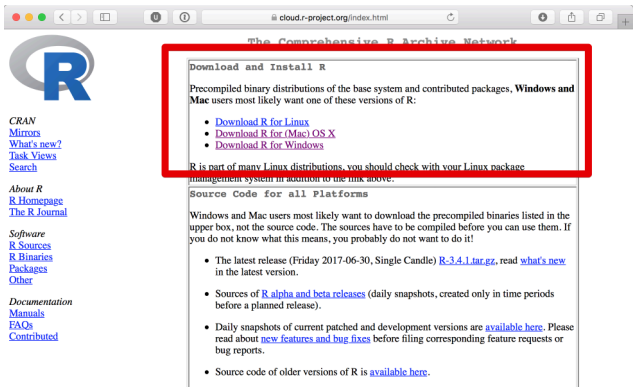
3 Installation

Hier eine kurze Anleitung um R und RStudio zu installieren.
Kurz zum Kontext: R ist eine Programmiersprache, das heißt auch "R installieren" ist anders als "Spotify installieren".

Ihr installiert zunächst R, und bekommt dann auf dem Desktop Verknüpfungen angeboten für "R GUI" (o.Ä.) — Dabei handelt es sich um "nackte" Konsolenprogramme.
Damit *könntet* ihr R benutzen, aber es macht beim besten Willen keinen Spaß.
Deshalb installiert ihr **RStudio** — Ein Programm (wie Spotify, nur ganz anders!), mit dem ihr R komfortabler benutzen könnt.

3.1 R installieren

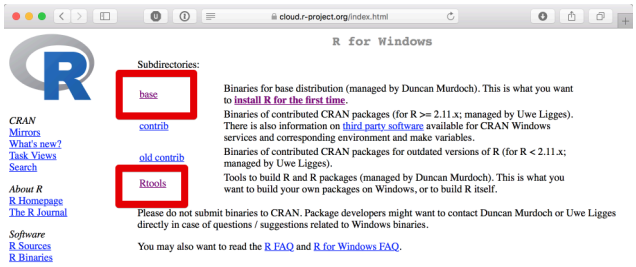
R bekommt ihr von der offiziellen seite, hier: https://cran.r-project.org/



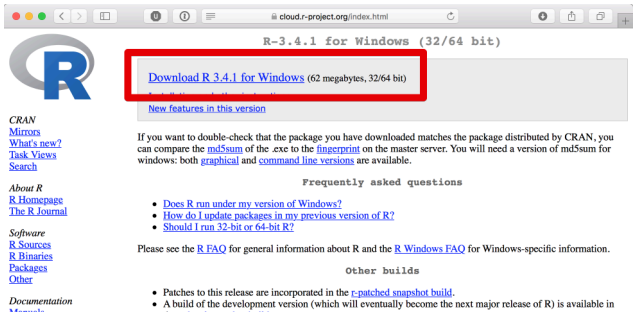
Ja, die Seite sieht nach heutigen Standards furchtbar alt aus, aber die gibt's nunmal auch schon ewig und sie hat den Anspruch möglichst spartanisch zu sein um auch auf jedem noch so ranzigen Computer ordentlich dargstellt und ggf. von Scripten ausgelesen zu werden.

3.1.1 Windows

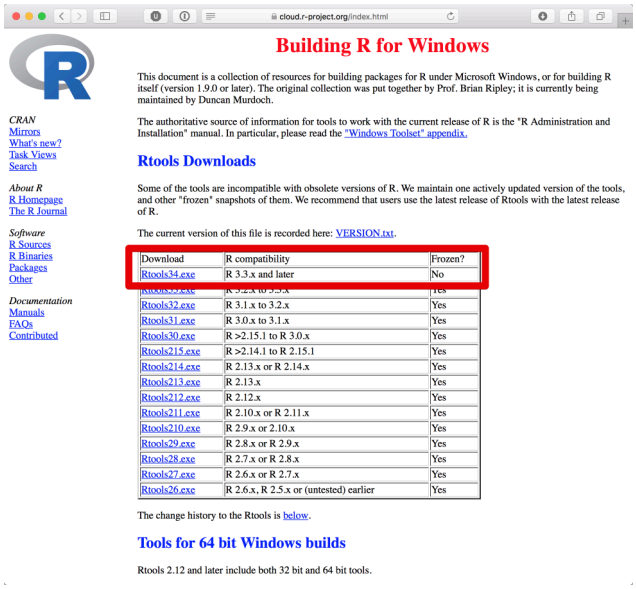
Klickt auf der oben genannte Seite auf `Download R for Windows`. Vollkommen unerwartet.



Hier benötigt ihr `base` (der R installer):



…und sicherheitshalber auch `Rtools` (fragt nicht. Vorerst installiert ihr es einfach, nur für den Fall, das es euch später Probleme erspart).

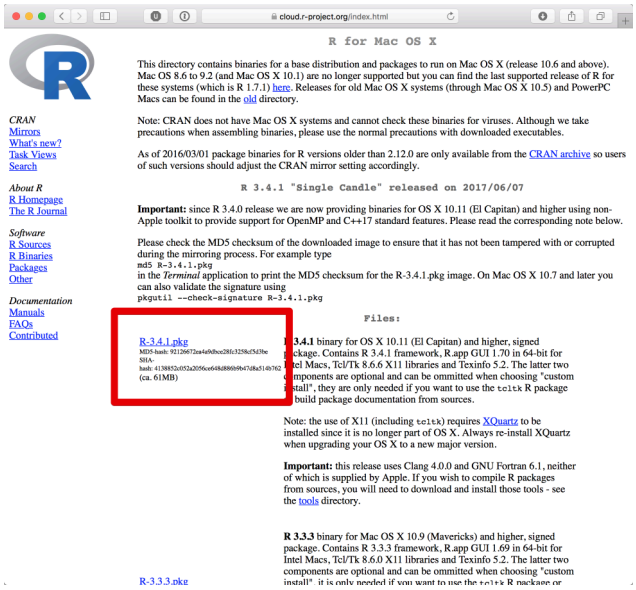


Da ich akut keinen Windows-Computer zur Hand habe, gehe ich einfach davon aus, dass ihr das Prinzip "`.exe`-Datei doppelklicken und alles brav mit `ja` oder `OK` bestätigen" beherrscht. Schafft ihr bei Spotify ja auch.

Das war die R-Installation!
Und *jetzt* installiert ihr RStudio

3.1.2 Mac OS X (macOS)

Klickt auf der oben genannte Seite auf `Download R for (Mac) OS X`. Auch das, vollkommen unerwartet.



- Wenn euer Betriebssystem auf dem aktuellen Stand ist, benutzt die aktuelle Version des R-installers.

- Wenn ihr aus unerfindlichen Gründen noch eine sehr alte OS X version laufen habt:

  a) Möge die Deität eurer Wahl euch beistehen
  b) Aktualisiert euer Betriebssystem, oder wenn das nicht möglich ist…
  c) …müsst ihr vermutlich den Installer mit `snowleopard` im Namen benutzen

- Wenn ihr nicht wisst, welche OS X Version ihr benutzt: Versucht's erstmal mit dem aktuellen Installer, wenn das nicht funktioniert, versucht den älteren. Ansonsten: Googlet.

Ansonsten braucht ihr nichts runterzuladen. Den Installer (die `.pkg`-Datei) einfach doppelklicken und alles brav bejahen.
Fertig.

3.1.3 Linux

Wenn ihr Linux benutzt, solltet ihr unabhängig von der Geschmacksrichtung auch wissen, wie man da Software installiert.

Es gibt diese Info-Seite für Ubuntu: https://cloud.r-project.org/bin/linux/ubuntu/README.html.
Wenn ihr nicht wisst, was `https://<my.favorite.cran.mirror>/bin/linux/ubuntu` sein soll, dann setzt für `<my.favorite.cran.mirror>` am besten einfach `cloud.r-project.org` ein. Die Zeilen unter "Installation" im verlinkten Dokument würden dann so aussehen (für Ubuntu 18.04 bionic):

- `deb https://cloud.r-project.org/bin/linux/ubuntu bionic/`

Zusätzlich solltet ihr vermutlich einige Systempakete (also außerhalb R) installieren, für den Fall der Fälle.

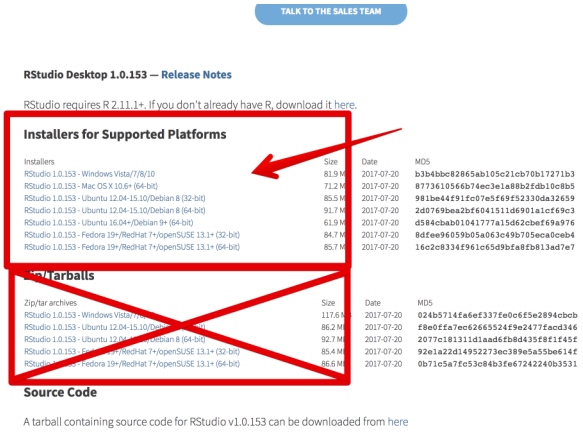
Für Ubuntu zum Beispiel (nur aus eigener Erfahrung, YMMV):

```
sudo apt install libcurl4-openssl-dev curl git libxml2-dev libcairo2-dev
```

Damit installiert ihr ein paar Pakete, die ihr für manche R-packages als dependencies benötigt. Die Liste ist nicht vollständig für *alle* möglichen R packages, aber zumindest alle, die wir im Laufe dieser Intro benutzen oder benutzen *könnten* sollten damit abgedeckt sein.

3.2 RStudio installieren

RStudio bekommt ihr hier: https://www.rstudio.com/products/rstudio/download/#download



Hier dasselbe Spiel:
Installationsdatei für euer Betriebssystem runterladen und installieren traue ich euch zu. Enttäuscht mich nicht.

Nach der Installation solltet ihr jedenfalls folgendes Symbol entweder auf eurem Desktop oder in eurem Programmordner finden:

Das wollt ihr anklicken, dann sollte sich RStudio öffnen und ihr habt erfolgreich R und RStudio installiert.
Gratuliere.

Falls ihr Windows 10 benutzt und sich keine Desktopverknüfung erstellt habt, wollt ihr eure Suchfunktion benutzen und “RStudio” suchen. Vollkommen unerwartet, ich weiß.

3.3 RStudio benutzen

Das ist keine Anleitung, das ist eine Aufforderung.

Ihr benutzt *ausschließlich* RStudio. Alle anderen Desktopverknüpfungen die im Laufe der Installationen erschienen sein sollten könnt ihr getrost löschen, die braucht ihr nicht.
Dazu gehören solche Dinge wie “R Console” oder auch “R GUI” oder ähnliches. Das sind die Verknüpfungen für das reine R, aber wir wollen R ja durch RStudio benutzen.
Trust me, it’s better that way.

In RStudio könnt ihr jetzt vermutlich noch nicht viel machen, außer in der Konsole aus Spaß “`print('Wurstwasser')`” eingeben und Enter drücken.

4 Orientierung

Wenn ihr vorher noch nie eine Programmiersprache benutzt habt… ist das auch eigentlich gar nicht so schlimm, denn R verhält sich sowieso für den Einstieg etwas anders.
"Programmiersprache" klingt so abschreckend, weil es nach kompliziertem Informatikkram klingt, aber wenn ihr es in eurer Schulzeit geschafft habt einen Taschenrechner zu bedienen, dann bekommt ihr auch den Einstieg in R hin.
Was der Begriff "Programmiersprache" für uns heißt ist recht simpel:
R folgt Anweisungen. Anweisungen, die wir entweder in die Konsole (in RStudio das Fenster unten links) schreiben können, und dann mit der Entertaste bestätigt werden. R guckt dann, ob es weiß wovon ihr redet, und rödelt dann los — und wenn es weiß was es mit eurem Befehl anfangen soll, gibt es euch auch direkt eine Antwort.

Gute Praxis[1] ist es übrigens, ein *Script* zu erstellen, das Scriptfenster ist dann in RStudio oben links. Dort könnt ihr einen Befehl pro Zeile schreiben und als Datei abspeichern, so müsst ihr euch bei eurer Arbeit nicht jeden Befehl einzeln merken, sondern könnt einfach das Script wieder neu ausführen und eure Ergebnisse tauchen alle wieder auf.

4.1 RStudio

Wenn ihr RStudio öffnet, seht ihr vermutlich folgendes:

Auf der rechten Seite habr ihr unter dem Reiter "Environment" eine Übersicht eurer Dateien und angelegten Variablen, sowie darunter mit der Hilfe und diversem anderem Krams, der uns zuerst noch nicht interessiert.
Das große Fenster zur linken Seite ist die Konsole, mit der beschäftigen wir uns zuerst.

Bevor wir hier aber irgendwas machen, schaffen wir erstmal ein bisschen Struktur und erstellen ein neues Projekt.
Ein Projekt ist einfach nur ein bestimmter Ordner, in dem ihr arbeiten könnt. Idealerweise ist es auch ein Ordner, den ihr auf eurem Computer einfach wiederfindet.

[1] Wenn wir von "Guter Praxis" reden, dann meint das in der Regel bestimmte Gewohnheiten, die wir euch empfehlen, weil sie euer Leben langfristig einfacher machen. Ihr müsst nicht immer sofort verstehen wieso wir euch irgendetwas empfehlen, aber seid euch sicher, dass da mehrere Jahre Erfahrung hinter stecken.

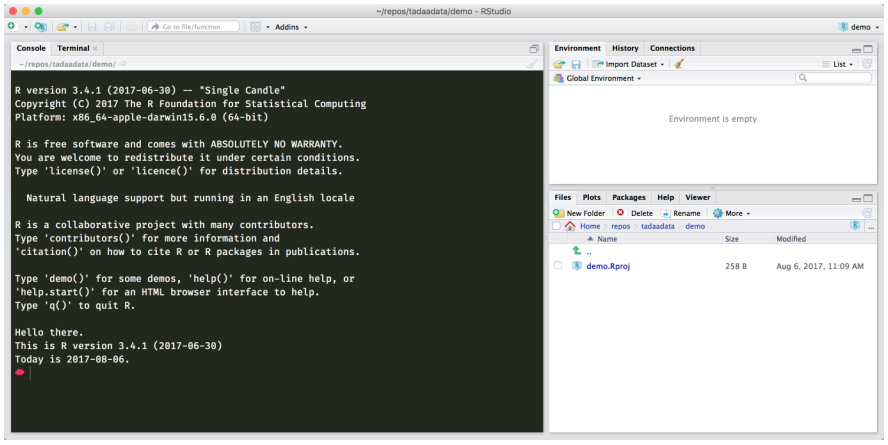


Figure 4.1: Ein frisches RStudio Fenster

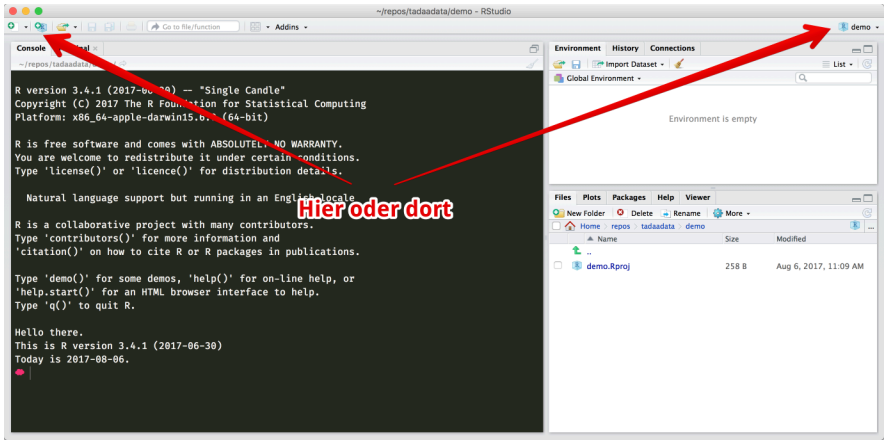
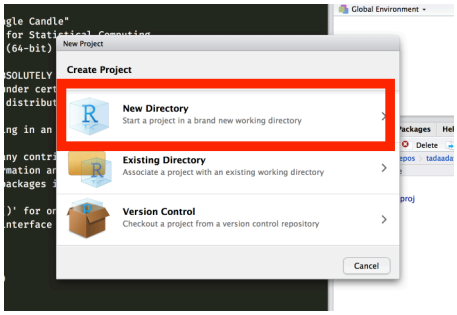
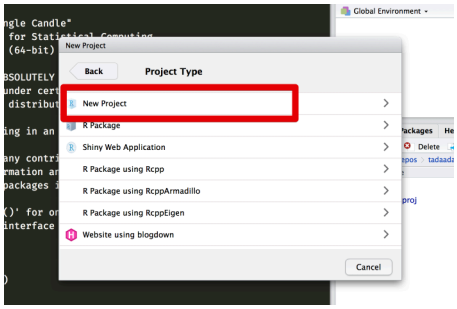
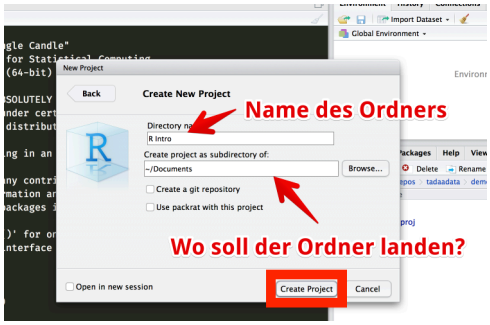


Figure 4.2: Ein neues RStudio Projekt erstellen







Am Ende solltet ihr euer erstes Projekt erstellt haben.
Hier könnt ihr jetzt den Rest der Einführung über bleiben und Scripte erstellen und sowieso und überhaupt, und wenn ihr brav alles gespeichert habt, könnt ihr darauf auch in drei Wochen noch wieder zurückgreifen, indem ihr einfach das Projekt aus der entsprechenden Leiste in RStudio auswählt.

Alternativ könnt ihr auch auf eurem Computer den Ordner mit dem Projekt öffnen[2], dort dann einfach die Datei mit dem RStudio-Logo und der `.Rproj`-Endung öffnen. Falls ihr Windows benutzt, werden die Dateiendungen per Standard nicht angezeigt - die Datei hat aber den gleichen Namen, den ihr auch dem Projekt gegeben habt und das Symbol sieht ungefähr so aus:

[2]Es bietet sich an, RStudio Projekte irgendwo zentral zu organisieren, oder sie zum Beispiel in einem Unterordner für euer Studiumszeug anzulegen. Bei mir war das z.B. sowas wie `Dokumente/Studium/Psychologie/Statistik/`

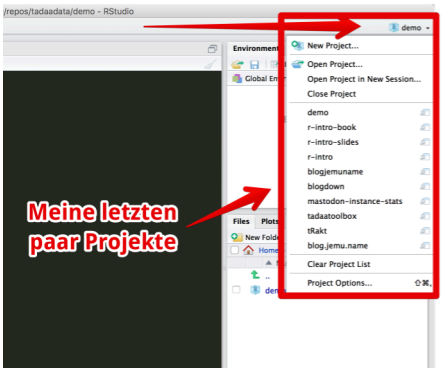


Figure 4.3: RStudio merkt sich eure letzten paar Projekte

Figure 4.4: Ein RStudio Projektordner

Als nächstes können wir uns **Konsole** und **Text Editor** widmen: Ihr könnt in der Regel immer nur in einem der beiden Bereiche schreiben, und erkennt den aktiven Bereich am blinkenden Cursor (ein bisschen wie in Textprogrammen á la Word) — Konsole und Editor sind die wichtigsten Bereiche in RStudio, und dort wird auch der Großteil eurer Arbeit gemacht.

4.2 Konsole

Jetzt habt ihr ein frisches Projekt und könnt loslegen.
Als erstes müssen wir uns mit der Konsole (dem Teil unten links) vertraut machen.
Hier könnt ihr Frage-Antwort-mäßig Befehle eingeben, mit Enter bestätigen, und erhaltet eine Antwort.
Hier kommt die Taschenrechner-Analogie wieder — versucht mal folgendes:

```
928 + 182
```

Die Mathe-Basics in R:

- Addition: `+`: `52 + 365` –> 417
- Subtraktion: `-`: `2017 - 18` –> 1999
- Multiplikation: `*`: `4*21` –> 84
- Division: `/`: `936/12` –> 78

Figure 4.5: Oh Schreck, es funktioniert tatsächlich!

- Exponentiation: `^`: `2^10` –> 1024
- Klammern: `(`, `)`: `3 * (12 - 8) + 2^(5/2)`
  - Wie beim Taschenrechner: Lieber zwei mehr als nötig als eine zu wenig
  - …und ja, jede offene Klammer braucht eine geschlossen Klammer, sonst gibt's Fehler
- Das Dezimaltrennzeichen ist der Punkt `.`: 12,1 wird eingegeben als `12.1`

Ansonsten gibt es etliche weitere mathematischen Funktionen, und wir können das ganze beliebig komplex aussehen lassen:

```
2 + sin((2*pi)/3) * exp(5)
```

Hier sehen wir mehrere neue Dinge:

Erstens: `pi` ist wirklich, naja, π. Das mit dem Kreis. Als Konstante schon in R vorgespeichert, weil ja kein Mensch π auf der Tastatur findet[3].
Zweitens: `sin()` und `exp()` sind **Funktionen**. Die sind ziemlich wichtig, aber denen wenden wir uns erst im nächsten Abschnitt zu.
Was wir da geschrieben haben sieht übersetzt in Mathe übrigens so aus…

$$2 + \sin\left(\frac{2\pi}{3}\right) \cdot e^5$$

… und ergibt etwa 130,5. Aber darum geht's eigentlich gar nicht.

4.2.1 Gängige Probleme

Was den meisten Leuten während der R-Einführung passiert ist, dass sie einen Befehl in die Konsole eingeben und Enter drücken, aber der Befehl nicht richtig abgeschlossen (*korrekt terminiert*) wurde. Das passiert zum Beispiel, wenn ihr eine schließende Klammer vergesst, oder etwa ein `+` am Ende der Zeile habt. In diesen Fällen drückt ihr Enter und R nimmt euren Befehl entgegen, aber es merkt, dass da irgendwas fehlt und wartet auf den Rest des Befehls.

[3]Die spitzfindigen Mac-User finden π mit *alt + p*. Das Zeichen kennt aber R nicht.

Ihr erkennt das daran, dass das Symbol an der linken Seite eurer Konsole auf einmal ein `+`-Symbol statt eines `>` ist und auch wiederholtes Drücken der Entertaste nichts daran ändert.

Figure 4.6: Plus was denn?!

Ihr habt an dieser Stelle zwei Möglichkeiten:

- Drückt Escape (`esc`) um den Befehl abzubrechen und es nochmal zu versuchen
- Führt den Befehl korrekt zu Ende, sprich schließt ggf. offene Klammern etc.

4.3 Text Editor

Alles was in der Konsole passiert ist schön und gut, aber es ist flüchtig. Stellt es euch vor wie eine Timeline auf Twitter oder ein Snapchat… Snapchat Dings oder was auch immer diese jungen Leute heutzutage benutzen.
Sobald ihr mehr als vier oder fünf Befehle eingegeben habt, müsst ihr hochscrollen, um eure alten Ergebnisse wieder zu finden. Das ist vollkommen okay um mal schnell etwas auszuprobieren, aber eher unpraktisch für eure Arbeit, die in der Regel sowas wie Reproduzierbarkeit erfordert.

Dazu gibt es Scripte. Scripte sind im Grunde nur Textdateien, in die ihr R-Befehle eingebt. Schön brav einen Befehl pro Zeile, wie in der Konsole.
Scripte könnt ihr speichern und an andere Leute verschicken oder hochladen oder euch ausdrucken und an die Backe tackern — der Kreativität sind keine Grenzen gesetzt! Scripte schreibt und speichert ihr, um eure Befehle / Auswertung / Code später wiederzufinden und nachvollziehen zu können. Ergebnisse reproduzieren könnt ihr indem ihr den Code aus dem Script nochmal ausführt.

Um euer erstes Script zu erstellen klickt ihr in RStudio oben links den Button, der nach *"neuer Datei"* aussieht:

Danach ploppt das Fenster oben links auf und begrüßt euch mit einem leeren Textfeld:

Im Moment heißt euer Script noch `Untitled1` — das heißt, euer Script hat noch keinen Namen und ist **noch nicht gespeichert**. Letzteres wollt ihr umgehend ändern, weil all eure schönen Befehle für die Katz sind, wenn ihr euren Kram nicht speichert.
Ihr könnt zum speichern entweder den anachronistischen Diskettenbutton klicken und eurem Script einen schönen Namen geben, oder ihr drückt `STRG + S` oder auf dem Mac `cmd + S` —

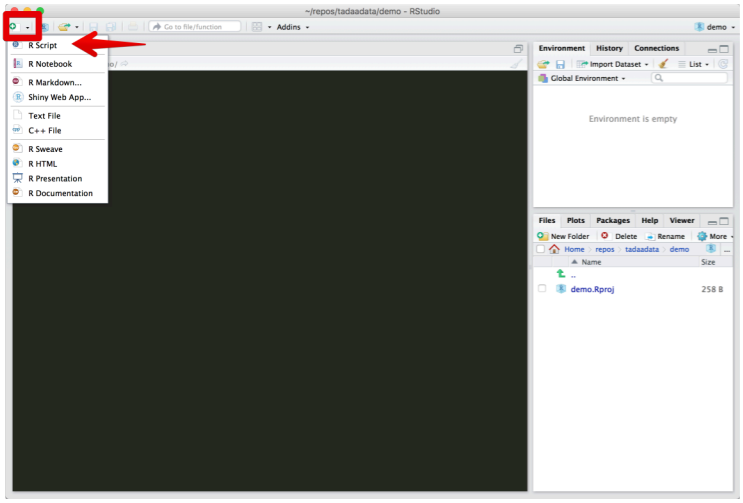


Figure 4.7: Liebes Tagebuch: Heute habe ich einen Button geklickt. Es war sehr schön.

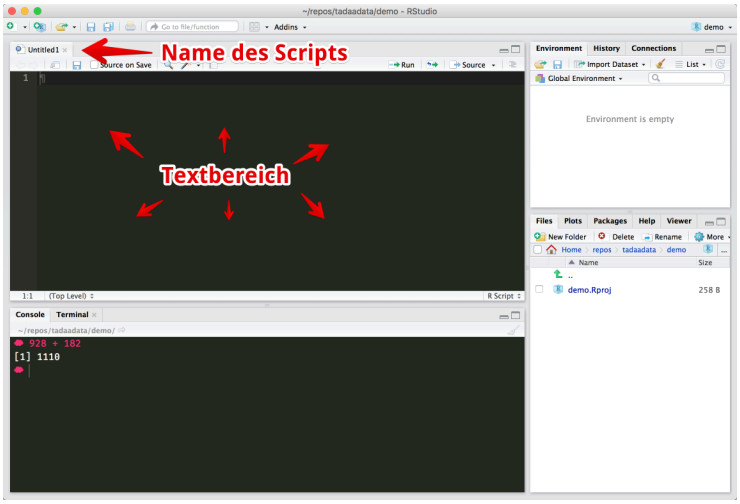


Figure 4.8: Na, auch hier?

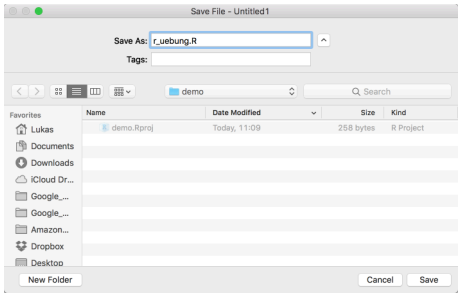


Figure 4.9: Der “Datei speichern…”-Dialog

der wohl wichtigste Keyboard-Shortcut der Welt. Wenn ihr Probleme habt eure Tasten zu finden oder zu benennen, dann guckt im Glossar oder googlet halt.

Gebt eurem Script einen aussagekräftigen Namen. Ihr wollt ja auch in zwei Wochen noch wissen, was ihr da gemacht habt.
Außerdem solltet ihr darauf achten, am besten ausschließlich Zahlen und Buchstaben sowie `-` und `_` zu verwenden. Leerzeichen und Umlaute (äöü) sind zwar *in der Theorie* kein Problem, aber glaubt mir, sobald ihr euer Script an KommilitonInnen mit anderen Betriebssystemen verschickt, kann auch jedes noch so harmlos aussehende *Ü* auf einmal zu einer Reihe von kleinen Problemen führen, deren Ursache ihr erst nach Stundenlanger Probiererei (oder niemals) finden würdet. Das ist im übrigen kein R-Ding, sondern gilt auch für alles andere; Word- & Excel-Dateien, PDFs, Pornovideos…

Etwas ähnliches gilt auch für den Text in eurem Script:
Vermeidet nach Möglichkeit besondere Zeichen wie Emoji (auch wenn die theoretisch korrekt angezeigt werden). Leerzeichen sind kein Problem, und sollten sogar der besseren Lesbarkeit halber großzügig eingesetzt werden.

Was der Lesbarkeit auch sehr hilft: Kommentare.
R ignoriert in Scripten sowie in der Konsole generell alles, was rechts neben einem `#` steht. Wir nennen dieses Zeichen übrigens entweder *Raute*, *Lattenzaun* oder *Octothorpe*. Wer es *hashtag* nennt muss leider 5€ in die Millenial-Dose werfen.

Damit können wir sowas machen:

```
# Wie alt bin ich nochmal?
2022 - 1991

# Wie viele Stunden im Jahr?
24 * 365
```

Damit bekommen eure Befehle Kontext, und sowohl ihr als auch eure KommilitonInnen können leicht rausfinden, was zum Geier ihr euch da eigentlich gedacht habt.
Kommentare sind auch praktisch, wenn ihr ein längere Script ausführt, aber ein Befehl Probleme bereitet. Wenn ihr einfach ein `#` davor setzt, ist die Zeile *auskommentiert*, und wird von R ignoriert.

Wenn ihr dann ein paar Zeilen Code angesammelt habt, könnt ihr euer Script ausführen.
Ein Script wird von oben nach unten (und von links nach rechts) ausgeführt, wenn ihr auf "`Source`" klickt oder `Shift + STRG + Enter` drückt (auf dem Mac `Shift + cmd + Enter`).
Wenn ihr nur die aktuelle Zeile (da wo euer Cursor gerade ist, ist "aktuell") ausführen wollt, reicht `STRG + R` (Mac: `cmd + R`). Auch hier kann das Glossar helfen.

4.4 Und das da rechts?

Auf der rechten Seite in RStudio findet ihr unter Anderem die Hilfe (*Help*), die Dateien in eurem Projektordner (*Files*), eine Übersicht der installierten packages (*Packages*), eine Variablenübersicht (*Environment*) und von euch erstelle Graphiken (*Plots*). Wenn ihr das hier in der richtigen Reihenfolge lest, habt ihr vermutlich keine Ahnung was das alles heißen soll — und genau deswegen wenden wir uns dem Ganzen auch Schritt wir Schritt in späteren Abschnitten zu, wenn ihr ein bisschen mehr Übersicht über die Grundlagen habt.

Part II

Grundlagen

5 Erste Schritte

Im letzten Abschnitt habt ihr R als glorifizierten Taschenrechner gesehen.
Als nächstes schauen wir uns an, was wir sonst so damit anstellen können.

Zuerst ein bisschen Terminologie:

> "To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call."
>
> – John Chambers

Oder in der Sprach-Analogie: Alles was in R existiert (Variablen, Tabellen, etc.) ist ein *Nomen* (Objekt) und alles, was etwas tut, ist ein *Verb* (Funktion).

5.1 Grundfunktionen

Die einfachsten Funktionen haben wir in Form der Rechenzeichen `+ - / *` schon kennengelernt, aber es gibt natürlich noch mehr.
Eine Funktion in R hat sieht immer ungefähr so aus: `sqrt(8)`. Der Name der Funktion, hier `sqrt`, (**immer** ohne Leerzeichen) gefolgt von Klammern, in denen ein oder mehrere *Argumente* stehen. Ein Argument ist das, womit die Funktion arbeiten soll. Eine gängige Veranschaulichung für Funktionen sind **Verben** einer Sprache, denn sie *tun* etwas.

Eine der wichtigsten Grundfunktionen ist `c()`, für *combine*. Mit `c` verbindet ihr mehrere Zahlen zu einem **Vektor** (ja, wie in der linearen Algebra. Mathe und so. Wisstschon.). Wenn ihr mehrere Zahlen zu einem Vektor kombiniert habt, könnt ihr damit so spaßige Dinge machen wie Mittelwerte ausrechen, sie aufsummieren oder zwei Vektoren gleicher Länge addieren.
Probiert mal ein paar Beispiele aus:

```
# Ein paar Zahlen
c(1, 1, 2, 3, 5, 8, 13, 21)
#> [1]  1  1  2  3  5  8 13 21

# Was ist der Mittelwert der Zahlen?
```

```r
mean(c(1, 1, 2, 3, 5, 8, 13, 21))
#> [1] 6.75

# Und die Summe?
sum(c(1, 1, 2, 3, 5, 8, 13, 21))
#> [1] 54

# Und wenn wir quadrieren?
c(1, 1, 2, 3, 5, 8, 13, 21)^2
#> [1]   1   1   4   9  25  64 169 441

# Oder die Wurzel ziehen?
sqrt(c(1, 1, 2, 3, 5, 8, 13, 21))
#> [1] 1.000000 1.000000 1.414214 1.732051 2.236068 2.828427 3.605551 4.582576
```

Was wir hier sehen ist der Unterschied zwischen Funktionen, die aus mehreren Zahlen eine machen (`mean`, `sum`), und Funktionen, die auf jeder Zahl einzeln operieren (`sqrt`, `^`).
Was wir außerdem sehen: Jedes mal die Liste von Zahlen `c(1, 1, 2, 3, 5, 8, 13, 21)` kopieren und in eine Funktion einsetzen ist ziemlich unpraktisch. Stellt euch vor, ihr habt eine Reihe von Testergebnissen von hunderten ProbandInnen und müsst da alles einzeln, also, nein, das wäre ja albern.
Für sowas gibt es dann Abstraktionen wie **Variablen** und **Datensätze**, die entweder eine Liste von Werten oder auch eine Liste einer Liste von Werten handlich machen — das sehen wir dann in den nächsten beiden Abschnitten.

Eine weitere praktische Funktion ist `length()`: Sie sagt uns, wie *lang* das Argument ist.
Wenn wir uns also angucken, wie der Mittelwert funktioniert…

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

…und wir das übersetzen in *"Die Summe aller Werte geteilt durch die Anzahl der Werte"*, dann können wir statt `mean` also auch folgendes schreiben:

```r
# in lang
sum(c(1, 1, 2, 3, 5, 8, 13, 21))/length(c(1, 1, 2, 3, 5, 8, 13, 21))
#> [1] 6.75

# in kurz
mean(c(1, 1, 2, 3, 5, 8, 13, 21))
#> [1] 6.75
```

Ihr seht vielleicht so langsam, wieso wir das mit den Klammern und den Leerzeichen für die Lesbarkeit erwähnt haben.
Aber gut, so langsam wird's unübersichtlich, es wird Zeit ein paar Variablen anzulegen.

5.1.1 Funktionsbeispiele

Wenn ihr R lernt, werdet ihr erfahrungsgemäß die meiste Zeit damit verbringen herauszufinden wie bestimmte Funktionen funktionieren und welche Funktion für euer Vorhaben die richtige ist.

Funktionen sind zwar vom Schema immer gleich — ihr steckt irgendwelche Argumente rein, und es kommt irgendein Ergebnis raus — aber wie die Argumente aussehen unterscheidet sich von Funktion zu Funktion.
`sd()` zum Beispiel hat zwei Argumente:

- `x`: Ein Vektor aus Zahlen, aus denen die Standardabweichung berechnet werden soll
- `na.rm`: Für `NA remove`, entweder `TRUE` oder `FALSE`. Wenn `x` fehlende Werte (`NA`) enthält, dann werden diese automatisch ignoriert, wenn `na.rm = TRUE`

```r
zahlen <- c(3, 6, 8, 3, 1, 2, 5, 6, 4, 3, NA, 4, 5, 7, NA, 1, 4)

# Ergibt NA :(
sd(zahlen)
#> [1] NA

# Ergibt ein Ergebnis :)
sd(zahlen, na.rm = TRUE)
#> [1] 2.065591
```

Der *default* für `na.rm` ist bei den meisten Funktionen (z.B auch `mean`) `FALSE`, das heißt fehlende Werte werden nicht automatisch ignoriert. Wenn euer *input* aber `NA` enthält, dann lässt sich daraus nicht sauber ein Mittelwert oder eine Standardabweichung berechnen, weil wir nichts über `NA` wissen (wir widmen uns `NA` im Kapitel zu Datentypen).
Nicht jede Funktion hat ein Argument namens `na.rm`, aber ihr werdet im Laufe der Zeit lernen, bei welchen Funktionen ihr darauf achten müsst, wie mit fehlenden Werten umgegangen wird.

Einige andere Funktionen, die insbesondere zum Lernen und Ausprobieren praktisch sind, werden zur Erstellung von **Sequenzen** benutzt — also Reihen von Zahlen in einem bestimmten Muster:

```r
# Die Zahlen von 1 bis 100
1:100
#>   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
#>  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
#>  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
#>  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
#>  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
#>  [91]  91  92  93  94  95  96  97  98  99 100

# 10 bis 15 in 0.5er-Schritten
seq(10, 15, 0.5)
#>  [1] 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5 15.0

# Äquivalent, mit explizit benannten Argumenten:
seq(from = 10, to = 15, by = 0.5)
#>  [1] 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5 15.0

# Von 5 bis -5 in ganzen Schritten
seq(5, -5)
#>  [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5

# Sequenz von 1 mit Länge 5, dasselbe wie 1:5
seq_len(5)
#> [1] 1 2 3 4 5
seq_len(length.out = 5)
#> [1] 1 2 3 4 5
```

Das `:` ist eine einfache Funktion und kann gelesen werden wie "Von … bis … in ganzen Schritten". Zusötzlich gibt es die Funktion `seq()`, die etwas flexibler ist. Außerdem gibt es diverse Funktionen für zufallsgenerierte Zahlen:

```r
# Normalverteilte Zufallszahlen (10 Stück, Mittelwert 5, Standardabweichung 2)
rnorm(n = 10, mean = 5, sd = 2)
#>  [1] 5.576295 3.998968 3.146825 5.536502 6.091464 6.022787 3.412928 4.386278
#>  [9] 5.441071 6.073732

# Gleichverteilte Zufallszahlen (sprich "r unif", nicht "run if")
runif(n = 10)
#>  [1] 0.41068732 0.87902860 0.61701363 0.70995547 0.06679693 0.31640234
#>  [7] 0.65379679 0.43960666 0.90341179 0.31753057
```

```r
# Münzwurf (Binomialverteilte Ergebnisse), 10 Stück
rbinom(n = 10, size = 1, prob = 0.5)
#>  [1] 0 1 0 1 0 0 0 0 0 0

# Würfel (W6) (10 Stück)
sample(x = 1:6, size = 10, replace = TRUE)
#>  [1] 4 1 2 3 4 4 5 3 5 1
```

5.2 Variablen

Wenn Funktionen wie Verben sind, dann sind Variablen wie **Nomen**. Sie haben einen Namen, und mit ihnen kann man Dinge tun. Oder sogar Sachen machen.
Variablen werden durch eine *Zuweisung* (*Assignment*) erstellt, was in R traditionell via `<-` passiert.
In den meisten anderen Programmiersprachen benutzt man dafür `=`, aber nun ja, R ist historisch gewachsen[1], also nehmt für den Anfang einfach mal hin, dass das nunmal so ist.
Wir speichern also mal ein paar Dinge:

```r
# Speichern in "fib"
fib <- c(1, 1, 2, 3, 5, 8, 13, 21)

# Ausgeben lassen
fib
#> [1]  1  1  2  3  5  8 13 21

# Mittelwert…
mean(fib)
#> [1] 6.75

# …funktioniert immer noch. Angenehm.
# Und wenn wir…
fib + fib
#> [1]  2  2  4  6 10 16 26 42

# Abgefahrener Kram.
```

[1] Money quote: "[…] the reason we use `<-` for assignment is it made sense in a programming language written before the incorporation of Apple Computers, because it made sense in a programming language written before the moon landings."

Hier haben wir die Zahlen `1, 1, 2, 3, 5, 8, 13, 21` in die Variable `fib` gespeichert, weil es die ersten paar Fibonacci-Zahlen sind, und wir unsere Variablen immer so benennen sollten, dass wir später noch wissen wofür sie da sind. Viele Tutorials beginnen damit, Variablen wie `x`, `y` und `z` anzulegen, aber da blickt ja irgendwann kein Mensch mehr durch.
Wenn ihr jetzt mit euren Zahlen arbeiten wollt, könnt ihr einfach in jeder Funktion `fib` statt der Liste mit `c(…)` einsetzen, und alles funktioniert wie vorher. Das liegt daran, dass R bei jedem Befehl erstmal nachschaut, ob ihr eine Variable benutzt (alles was Text ohne Anführungszeichen ist), und ob es die Variable findet. Wenn es die Variable gefunden hat, guckt es nach, was da drinsteht, in diesem Fall also `c(1, 1, 2, 3, 5, 8, 13, 21)`, dann benutzt R den Inhalt der Variablen.

An dieser Stelle bietet es sich an, einen neuen Typ einzuführen: Den String (oder auch `character`). Als String bezeichnet man im Kontext von, naja, Computerkram generell eigentlich, alles was als Text durchgeht. Sobald wir etwas nicht mehr nur durch Zahlen darstellen können, ist es ein String.
Strings stehen in R immer in Anführungszeichen, entweder in "doppelten" oder in 'einfachen'.

Wichtig dabei ist, dass sich Anführungszeichen ähnlich verhalten wie Klammern. Wenn wir einen String mit " beginnen, müssen wir ihn auch wieder mit " schließen, ansonsten wartet R brav darauf, dass endlich das zweite " kommt und verläuft sich.
Ein Beispiel:

```
# Vollkommen okay
namen <- c("Tobi", "Lukas", "Nadja", "Christoph")

# Auch okay, aber inkonsistent und daher eher unschön
namen <- c('Tobi', "Lukas", 'Nadja', "Christoph")

# Tod und Verderben (=> funktioniert nicht)
namen <- c("Tobi", 'Lukas", 'Nadja, Christop")
```

Wir können auch Zahlen in `""` setzen — das ist kein Problem, aber dann sind es nunmal keine Zahlen in diesem Sinne mehr, es sind *Strings*, und mit Strings können wir nicht rechnen.
Probiert folgendes aus:

```
# Okay
5 + 5

# Hä?
5 + "5"
```

Ihr sehr jetzt vermutlich die Meldung `Error in 5 + "5" : non-numeric argument to binary operator`.
Das *non-numeric argument* hier ist die `"5"`. Merken: `5` ist **numerisch**, aber `"5"` ist ein **character** (=> **String**).
Der *binary operator* an dieser Stelle ist übrigens das `+`. Ein **Operator**, weil es, äh… operiert? Naja, es tut Dinge, und wenn etwas in R Dinge tut, ist es meistens ein Operator in irgendeinem Sinne. Das *binary* heißt, dass es **zwei** (bi, binär, binary, zwei halt) Argumente nimmt.
Wie schon gesagt, Argumente sind die Dinge, die wir an Funktionen übergeben, und wenn wir an eine Funktion wie `+` oder auch `mean()` ein Argument übergeben, mit denen sie nichts anfangen können, dann beschwert sich R weil es nicht weiß was zum Geier ihr da vorhabt.

```
namen <- c("Tobi", "Lukas", "Nadja", "Christoph")

fib <- c(1, 1, 2, 3, 5, 8, 13, 21)

# Alles knorke
mean(fib)
#> [1] 6.75

# Alles CHAOS UND UNHEIL
mean(namen)
#> Warning in mean.default(namen): argument is not numeric or logical:
#> returning NA
#> [1] NA
```

Was es mit `NA` auf sich hat, und was es noch so für Datentypen gibt, sehen wir dann im Abschnitt zu Datentypen.

Eine letzte Sache noch: Strings sind "dominanter" als Zahlen, das heißt, wir können zwar Zahlen verbinden zu `c(1, 2, 3)`, und Strings zu `c("A", "B", "C")`, aber wenn wir `c("A", 2, "C", 4)` schreiben, dann behandelt R einfach alle Elemente des Vektors (=> Das, was in `c(…)`) steht, als wären es `character`-Werte.

> **Note**
>
> Note that there are five types of callouts, including: Merke: Ein Vektor in R muss immer Elemente des gleichen Typs haben, Zahlen und Buchstaben zusammen werden zu Strings konvertiert!

5.3 Tabellen

Jetzt haben wir schonmal das Vokuabular an der Hand um Zahlen und beliebige Strings in R zu verarbeiten, aber noch ist das alles etwas unahndlich um damit *richtig* zu arbeiten.

Stellt euch vor, wir wollen einen kleinen Datensatz erstellen über die Statistiktutorien in QM mit Variablen wie *Namen*, *Alter*, und vielleicht sowas wie *Beliebtheit* auf einer Skala von 1-10. Wir könnten sowas machen:

```
namen <- c("Tobi", "Christoph", "Nadja", "Lukas")
alter <- c(20, 35, 30, 12) # (Nicht alle diese Werte sind korrekt)

mean(alter)
#> [1] 24.25
```

Schön und gut, aber das ist ja unhandlich. Was, wenn wir die Namen aller TutorInnen haben wollen, die jünger als 30 sind? Alles was wir mit `alter` machen, passiert unabhängig von `namen`.

Um mehrere Variablen in Kontext zu setzen, gibt es tabellarischen Datenstrukturen, namentlich nennt sich sowas in R dann `data.frame`. Letztendlich ist das nichts anderes als eine Tabelle, aber für R ist eine Tabelle praktische eine Liste von Vektoren mit gleicher länge:

```
leute <- data.frame(name = c("Tobi", "Christoph", "Nadja", "Lukas"),
                    alter = c(20, 35, 30, 12),
                    beliebtheit = c(9, 10, 8, 3))

# Anzeigen lassen
leute
#>        name alter beliebtheit
#> 1      Tobi    20          9
#> 2 Christoph    35         10
#> 3     Nadja    30          8
#> 4     Lukas    12          3
```

Was haben wir da gemacht?

- Wir haben einen `data.frame` mit der gleichnamigen Funktion erstellt
- Die **Argumente** der Funktion haben die Form `Spaltenname = Werte der Spalte`
- Mehrere Argumente werden mit `,` getrennt und optional mit einem Zeilenumbruch übersichtlich gehalten

Das Ergebnis ist eine Variable `leute`, die drei Spalten mit je vier Werten hat.
Jede Spalte ist eine Variable, und jede Zeile der Tabelle kann als eine *Beobachtung* betrachtet werden.
Eine Beobachtung (*Observation*) sind alle Werte, die wir zu einem Untersuchungsobjekt haben, also in diesem Beispiel eine Person. Wenn wir uns nur die erste Zeile anschauen, sehen wir nur die Werte, die zu Tobi gehören, in der zweiten Zeile sehen wir die Werte zu Christoph etc.

Tabellen, und damit `data.frames`, sind für uns die wichtigsten Objekte in R, weil wir fast ausschließlich mit Datensätzen in dieser Form arbeiten werden um unsere Statistik da draufzuwerfen.
Wie können wir jetzt mit einzelnen Variablen arbeiten?

```
# Die Variable "name" ausgeben lassen
leute$name
#> [1] "Tobi"      "Christoph" "Nadja"     "Lukas"

# Den Mittelwert von "alter" bestimmen
mean(leute$alter)
#> [1] 24.25

# Die Standardabweichung von "beliebtheit"
sd(leute$beliebtheit)
#> [1] 3.109126

# Was auch funktioniert:
leute[["name"]]
#> [1] "Tobi"      "Christoph" "Nadja"     "Lukas"
leute[["alter"]]
#> [1] 20 35 30 12
```

Das mit den "Levels" wird im Abschnitt zu Datentypen erklärt

Was wir hier benutzen nennt sich *Subsetting*, also im Grunde nur einen Teil von etwas rausholen. Hier also einen Teil der Tabelle on Form einer einzelnen Spalte.
Spalten können wir mit `$` oder `[[ ]]` direkt aus einem `data.frame` ansteuern, was unser Leben gleich viel einfacher macht. Strenggenommen sidn `$` und `[[` auch eigene Funktionen, aber dazu vielleicht später mehr, im Moment ist für uns nur wichtig, dass wir einzelne Spalten (Variablen) einer Tabelle (`data.frame`) einfach adressieren und genauso behandeln können wie die einzelnen Variablen `name` und `alter`, die wir weiter oben erstellt haben.

5.4 Umgang mit Datensätzen

Da wir noch nicht an dem Punkt sind, wo wir beliebige Daten einlesen können, und wir natürlich zu faul sind uns eine größere Tabelle selber zu schreiben, greifen wir zu Übungszwecken mal auf einen Datensatz zurück, der bei R von Haus aus mitgeliefert wird: `sleep`.

Dieser Datensatz beinhaltet die Daten aus einer Medikamentenstudie, bei der es um Schlafgewinn bzw. -verlust ging. Die Tabelle hat drei Spalten (Variablen) zu 10 Personen:

- `extra`: Schlafzuwachs in Stunden, positiv oder negativ für mehr bzw. weniger Schlaf als vorher
- `group`: Die Versuchsgruppe, sprich welches Medikament die Person bekam, `1` oder `2`
- `ID`: Die Identifikationsnummer der Person. Es ist gängig, ProbandInnen pseudonymisiert durchzunummerieren, der Zuordnung unt des Datenschutzes wegen als Zahlen.

```
# Mit head() lassen wir uns die ersten paar Zeilen (den "Kopf") der Tabelle anzeigen
head(sleep)
#>   extra group ID
#> 1   0.7     1  1
#> 2  -1.6     1  2
#> 3  -0.2     1  3
#> 4  -1.2     1  4
#> 5  -0.1     1  5
#> 6   3.4     1  6
```

Wie viele Zeilen hat die Tabelle?

```
nrow(sleep)
#> [1] 20
```

Die *number of rows* bekommen wir mit `nrow()` — ihr dürft jetzt raten, wie wir uns die Anzahl der Spalten (*columns*) anzeigen lassen können.

```
ncol(sleep)
#> [1] 3
```

Surprise!

Okay, aber was interessiert uns an diesem Datensatz jetzt? Wie wäre es mit dem durchschnittlichen Schlafzuwachs:

```
mean(sleep$extra)
#> [1] 1.54
```

Schön und gut, aber wir wollen ja vermutlich die beiden Gruppen (Medikamente) vergleichen, also was tun?
Subsetting to the rescue /o/

```
gruppe1 <- sleep[sleep$group == 1, ]
gruppe2 <- sleep[sleep$group == 2, ]

# Mittelwert der ersten Gruppe
mean(gruppe1$extra)
#> [1] 0.75

# Mittelwert der zweiten Gruppe
mean(gruppe2$extra)
#> [1] 2.33
```

Okay, Schritt für Schritt.
Hier haben wir unseren ersten logischen vergleich benutzt, um eine Teilmenge der Tabelle zu extrahieren.
Das klingt fancy, ist aber ziemlich simpel.
In Worten heißt die Zeile `gruppe1 <- sleep[sleep$group == 1]` lediglich:
"*Nimm die Tabelle* ***`sleep`*** *und filtere daraus alle Zeilen, die zu der Gruppe* ***`1`*** *gehören, und speichere sie in die Variable* ***`gruppe1`***" Das Resultat sind zwei Variablen, die einen Teil der Tabelle `sleep` enthalten, und zwar jeweils zu einer der beiden Gruppen.

Wieso dann eigentlich noch diese `,`-Sache am Ende der eckigen Klammern?
Das gehört zur Art, wie R Tabellen *indiziert*, sprich wie man einzelne Bereiche der Tabelle ansteuert:

```
# Die erste Spalte
sleep[1]
#>    extra
#> 1    0.7
#> 2   -1.6
#> 3   -0.2
#> 4   -1.2
#> 5   -0.1
#> 6    3.4
#> 7    3.7
#> 8    0.8
#> 9    0.0
#> 10   2.0
#> 11   1.9
```

```
#> 12   0.8
#> 13   1.1
#> 14   0.1
#> 15  -0.1
#> 16   4.4
#> 17   5.5
#> 18   1.6
#> 19   4.6
#> 20   3.4

# Die erste Zeile
sleep[1, ]
#>   extra group ID
#> 1   0.7     1  1

# Die erste Zeile und die dritte Spalte
sleep[1, 3]
#> [1] 1
#> Levels: 1 2 3 4 5 6 7 8 9 10
```

Die allgemeine Form ist `tabelle[Zeilennummer, Spaltennumer]`, und jetzt fragt ihr euch vermutlich, wieso wir vorhin `[[ ]]` benutzt haben, und jetzt `[ ]` — die kurze Antwort ist: Das ist halt was anderes. Die Details sind erstmal nicht so wichtig, was ihr euch vorerst merken solltet ist folgendes:

- `sleep[1]` ergibt einen `data.frame` mit **nur einer Spalte**
- `sleep[1, ]` ergibt einen `data.frame` mit **nur einer Zeile**
- `sleep[[1]]` und `sleep$extra` sind **dasselbe** (weil `extra` die erste Spalte ist) und ergeben die erste Spalte als **Vektor**
- `sleep$extra[[2]]` und `sleep$extra[2]` sind hier **dasselbe**: Das zweite Element im Vektor `sleep$extra`

Bei einer Tabelle ist es nützlich mit Zeilen und Spalten zu arbeiten, um die gewünschten Werte rauszuholen, aber bei einem Vektor gibt es in diesem Sinne nur eine Dimension.
Sinn der Sache ist, dass wir Funktionen wie `mean` oder `sd` nur auf Vektoren anwenden können, was auch intuitiv irgendwie sinnvoll scheint, denn der Mittelwert einer ganzen Tabelle mit mehreren Variablen ist ja konzeptionell etwas… schwierig.

```
# Okay
mean(sleep$extra)
#> [1] 1.54
```

```r
# Das selbe Ergebnis
mean(sleep[[1]])
#> [1] 1.54

# Auch okay!
mean(sleep[["extra"]])
#> [1] 1.54

# Das hier nicht so
mean(sleep[1])
#> Warning in mean.default(sleep[1]): argument is not numeric or logical:
#> returning NA
#> [1] NA
```

`sleep[1]` gibt euch zwar auch die Spalte `extra`, aber wie schon gesagt, in `data.frame`-Form, und nicht als Vektor.

Vermutlich verwirrt euch das ganze Geklammere jetzt mehr oder weniger stark, aber glaubt mir, wenn wir erstmal ein Gefühl dafür habt ist es sehr viel Wert diese Grundlagen auf dem Schirm zu haben (oder sie zumindest nachlesen zu können), denn in der ersten Zeit eurer R-Nutzung werdet ihr massenhaft kleinere und größere Fehler in dieser Art machen, wo ihr zwar das richtige *meint*, aber R nicht das richtige *sagt*.
Die andere Sache ist, dass ihr das mit den eckigen Klammern gar nicht so häufig brauchen werdet, wenn ihr euch erstmal an das *tidyverse* und *dplyr* gewöhnt habt, aber dazu später mehr.

Wir schneiden das Ganze Thema *Subsetting* hier auch erstmal nur an, aber wenn ihr's jetzt schon ganz genau wissen wollt, könnt ihr die Details hier nachlesen

5.5 Logische Vergleiche

Logik! Eine Welt des Spaßes, der internen Konsistenz[^Naja, fast. Aber Gödel lassen wir mal aus.] und der unendlichen Anwendbarkeit in allen Bereichen.
Was ihr intuitiv als Logik kennt ist alleridngs etwas anderes als *formale Logik*, also das, was Computer verstehen.
Wir brauchen zum Glück nicht all zu viel davon, nur den Standardkram und nichtmal das volle Spekrum Bool'scher Algebra.

Wir brauchen Logik in R in erster Linie zum *indizieren* von Objekten. Das heißt, wenn wir alle Zeilen einer Tabelle haben wollen, für die eine bestimmte Variable einen bestimmten Wert hat oder eine *Bedingung* erfüllt, dann drücken wir das durch Logik aus. Dasselbe funktioniert

natürlich auch bei Vektoren (und strenggenommen funktioniert Tabellenindizierung sowieso über Vektorindizierung). Man nehme folgendes Beispiel:

```
sleep[sleep$extra > 3, ]
#>    extra group ID
#> 6    3.4     1  6
#> 7    3.7     1  7
#> 16   4.4     2  6
#> 17   5.5     2  7
#> 19   4.6     2  9
#> 20   3.4     2 10
```

Das heißt: *"Nimm die Tabelle* `sleep` *und gib mir alle Zeilen (das mit den eckigen Klammen), für die die Variable* `sleep$extra` größer als 3 *ist"*.

Das Ergebnis eines logischen Vergleichs ist immer entweder `TRUE` oder `FALSE` für wahr oder falsch.
Wenn wir in R indizieren wollen, können wir dafür auch direkt `TRUE` und `FALSE` statt eines Vergleichs benutzen:

```
fib <- c(1, 1, 2, 3, 5, 8, 13, 21)

fib[c(TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE)]
#> [1] 1 1
```

Hier haben wir uns effektiv nur die ersten beiden Werte des Vektors `fib` ausgeben lassen, weil R alles ausgibt, was mit `TRUE` indiziert ist und alles weglässt, was mit `FALSE` indiziert ist.
`c(TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE)` ist hier ein *logischer Vektor*, also ein Vektor mit, naja, logischen Werten, der praktisch der Reihe nach jedes Element im Vektor `fib` entweder *an* oder *aus* schaltet, wie Lichtschalter. Wir müssen dafür nicht unbedingt einen logischen Vektor der gleichen Länge (Anzahl der Elemente) wie unser Zielvektor (der, den wir indizieren/filtern wollen) benutzen, aber es bietet sich für den Einstieg an so genau wie möglich zu sein.
Wir könnten aber auch sowas machen:

```
# Immer abwechselnd TRUE und FALSE, also jedes zweite Element
fib[c(TRUE, FALSE)]
#> [1]  1  2  5 13

# Schema TRUE TRUE FALSE FALSE ginge auch
fib[c(TRUE, TRUE, FALSE, FALSE)]
#> [1] 1 1 5 8
```

Wichtig hierbei ist, dass der logische Vektor ein *ganzer Faktor* des Zielvektors ist, das heißt, dass die Anzahl der Element im Zielvektor *ganz* durch die Anzahl der Elemente im logischen Vektor teilbar sein muss (also ohne Rest), ansonsten bekommen wir potenziell schwer vorhersagbare Ergebnisse.
Was R hier macht nennt sich *Recycling*: In der ersten Zeile im letzten Beispiel wird der Vektor `c(TRUE, FALSE)` solange *recyclet*, also wiederverwendet, bis der ganze Zeilvektor "abgedeckt" ist. Wenn der logische Vektor nicht sauber in den Zielvektor passt (in Bezug auf die Anzahl der Elemente), dann bleibt entweder was übrig oder es reicht nicht. Das wäre schade.

5.5.1 Operatoren

Es gibt eine Reihe *logischer Operatoren* wie hier `>` für *"ist größer als"*, die wichtigsten in Übersicht:

- `==` (doppeltes Gleichheitszeichen **ohne Leerzeichen dazwischen**)
  - "Ist gleich"
  - `1 == 2` –> `FALSE`
  - `3 == 3` –> `TRUE`
  - `3 == "Hallo"` –> `FALSE`
- `!=`
  - "ist ungleich"
  - Die *Negation* von `==`, also immer da wo `==` euch `TRUE` zeigt, gibt `!=` euch `FALSE` und andersherum.
  - `pi != 3` –> `TRUE`
  - `"Psychologiestudium" != "Voll gute Idee"`
- `!` (ja, ein einfaches Ausrufezeichen)
  - "Negation"
  - Dreht ein `FALSE` zu einem `TRUE` um und andersherum. Wichtig: Klammern!
  - `!(2 == 3)`
  - `TRUE == !FALSE`
- `>` und `<` (spitze Klammern)
  - "ist größer/kleiner als"
  - Da wo die Klamer spitz ist, soll das *kleinere* sein
  - `5 > 4` –> `TRUE`
  - `2^10 < 1000` –> `FALSE`
  - `2 < 5 < 4` –> *Funktioniert nicht!*
- `>=`, `<=`
  - "Größer gleich bzw. kleiner gleich"

  - Ist *Entweder* `a > b` *oder* `a == b`?
  - `5 >= 4`
  - `16 <= 2^4`

Diese Ausdrücke können wir auch auf bestimmte Arten *verknüpfen*:

- `&` (oder auch `&&`)
  - "Und"
  - Ist `TRUE`, wenn beide Seiten `TRUE` sind
  - `(1 < 3) & (5 < 10)` –> `TRUE`
  - `(5 < 2) & (2 < 10)` –> `FALSE`
  - 1 und 0 werden zu `TRUE` bzw. `FALSE` übersetzt:
    - `1 & (2 == 2)` –> `TRUE`
    - `!(0 & FALSE)` –> `TRUE`
- `|` (oder auch `||`)
  - "Oder"
  - Ist *entweder* A *oder* B *oder* beides `TRUE`?
  - Da `|` auch wahr ist, wenn nur eine Seite wahr ist, ist es auch Grundlage etlicher Mathe-/Logikwitze
  - `(1 < 3) | (5 < 10)` –> `TRUE`
  - `(5 < 2) | (2 < 10)` –> `TRUE`
- `xor()`
  - "Entweder … oder …" (*ausschließend*!)
  - Wenn euch `|` zu unspezifisch ist
  - Ist *nur* war, wenn *eins von beidem* wahr ist, aber nicht, wenn beides wahr ist
  - **Kein** *binärer Operator* wie die anderen, sondern eine R-Funktion mit Klammern und so
  - `xor(TRUE, FALSE)` –> `TRUE`
  - `xor(TRUE, TRUE)` –> `FALSE`
  - `xor((1 < 3), (5 < 10))` –> `FALSE`
  - `xor((5 < 2), (2 < 10))` –> `TRUE`

5.5.2 Spezielle Tests

Die obigen Operatoren können wir für getrost für Vektorvergleiche benutzen, aber es gibt noch ein paar Sonderfälle. Was zum Beispiel, wenn wir nur generell wissen wollen, ob ein Element wie eine Zahl oder ein String in einem Vektor enthalten ist? Oder was, wenn wir wir auf spezielle Typen oder Klassen testen wollen? Was das im Detail heißt sehen wir in den entsprechenden Abschnitten zu Datentypen noch einmal, aber hier schonmal eine Kurzreferenz:

- `%in%` (auch hier, *ohne Leerzeichen* dazwischen!)
  - "Ist in"
  - Mengentheoretisch ist das $x \in X$
  - Ist Element `a` in Menge `b`?
  - `5 %in% c(1, 4, 5, 3)` –> `TRUE`
  - `"B" %in% c("a", "b", "c")` –> `FALSE`
  - `"B" %in% c("a", "B", "c")` –> `TRUE`
  - `c(1, 2) %in% 1:5` –> `TRUE`
- `is.na()`: Testet auf fehlende Werte (*missing values*, `NA`)
- `is.null()`: Testet auf leere Werte (`NULL`)
- `is.nan()`: Testet auf `NaN` (*Not a Number*)

5.5.3 Indexing: Beispiele

Das war jetzt relativ viel Information, und ihr müsst euch das auch nicht alles sofort merken, sondern nur wissen, wo ihr's bei Bedarf nachschlagen könnt.
Die Motivation hinter dem Logikram ist wie erwähnt primär das Filtern von Tabellen und Vektoren, was wir nunmal relativ häufig brauchen um zum Beispiel bestimmte Untergruppen in unseren Datensätze zu analysieren, zum Beispiel Personen älter als 35 (z.B. `age > 35`) oder Menschen, die sowohl weiblich sind als auch Medikament B bekommen haben (z.B. `geschlecht == "weiblich" & drug == "B"`).

5.5.3.1 Vektoren

Vektoren werden immer *elementweise* verglichen, das heißt, dass das Ergebnis von `c(1, 2) == 1` nicht `FALSE` oder `TRUE` ist, sondern der *logische Vektor* `TRUE FALSE`. Dadurch entsteht durch den logischen Vergleich eines Vektors ein Vektor aus `TRUE` und `FALSE`, den wir zum indizieren benutzen können, wie wir weiter oben schon gesehen haben.

```
# Irgendwelche Zahlen
x <- c(4, 7, 2, 1, 7, 9, 6, 5, 4, 3, 3, 2, 2, 5, 8, 9, 31)

# Alle Zahlen größer 4: "Gib mir x, wo x > 4"
x[x > 4]
#> [1]  7  7  9  6  5  5  8  9 31

# Die Logik dahinter
x > 4
#>  [1] FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
#> [13] FALSE  TRUE  TRUE  TRUE  TRUE
```

```r
# Alle Zahlen größer 5 und kleiner 20
x[x > 5 & x < 20]
#> [1] 7 7 9 6 8 9
```

Das Ganze lässt sich natürlich beliebig komplex aussehen lassen, weshalb zu viele Bedingungen in Kombination etwas verwirrend aussehen können.
Weiterhin können wir einen Vektor natürlich auch durch einen Vergleich eines anderen Vektors indizieren.

```r
x <- c(4, 7, 2, 1, 7, 9, 6, 5, 4, 3)
y <- c(6, 7, 10, 1, 9, 3, 6, 5, 6, 3)

# x, wo x größer gleich 4 ist *und* y kleiner 6
x[x >= 4 & (y < 6)]
#> [1] 9 5

# x, wo x und y identisch sind
x[x == y]
#> [1] 7 1 6 5 3

# x, wo x kleiner y ist
x[x < y]
#> [1] 4 2 7 4
```

5.5.3.2 Tabellen data.frame

Tabellenindexing ist nichts anderes als Vektorindexing mit einer anderen Struktur. Alle Regeln zum Vektorindexing, die wir bisher gesehen haben, gelten auch so für `data.frames`, nur dass wir hier jetzt auf einmal in Spalten und Zeilen denken müssen, anstatt in Vektoren.

6 Datentypen

Es gibt eine Reihe von Unterscheidungsmöglichkeiten zwischen verschiedenen Arten von Daten. Neben der naheliegenden Unterscheidung zwischen "Zahlen" und "Buchstaben" gibt es diverse andere Typen, die R verwendet.
Technisch gesehen müssten wir hier noch zwischen Typen und Klassen unterscheiden, aber für die meisten unserer normalen Anwendungszwecke ist es nicht unbedingt notwendig Typen und Klassen auseinanderhalten zu können, weshalb wir hier auch mehr oder weniger beides gleichzeitig abhandeln.

"Und wieso sollte uns interessieren, wie R da unterscheidet?"
R ist verwirrt, wenn wir Buchstaben in eine Funktion stecken, die Zahlen erwartet. Genauso ist R verwirrt, wenn wir den Mittelwert aus einer Tabelle berechnen wollen. Mittelwerte sind nur dann sinnvoll, wenn wir sie aus einem Vektor aus numerischen Werten berechnen. Da eure Daten in verschiedenen Formaten ankommen, und unterschiedliche Repräsentationen unterschiedliche Vor- und Nachteile haben, ist es wichtig, dass ihr im Zweifelsfall herausfinden könnt was ihr da vor der Nase habt und wie ihr damit arbeiten könnt.
Was eine R-Funktion mit einem Objekt anstellt hängt von der *Klasse* des Objekts ab, das ganze fällt vermutlich irgendwo unter *"object oriented programming"*, und wenn ihr InformatikerInnen kennt und die euch Fragen, ob R eine funktionale oder objektorientierte Frage ist, könnt ihr getrost "ja" sagen[1].

Die wichtigste Funktion für diesen Abschnitt ist `class()`, was euch sagt was R unter einem bestimmten Objekt versteht (die *Klasse* des Objekts):

```
class(4)
#> [1] "numeric"
class(c(1, 2, 3))
#> [1] "numeric"
class(c("hallo", "welt"))
#> [1] "character"
class(sleep)
#> [1] "data.frame"
class(sleep$extra)
#> [1] "numeric"
```

[1] Das ist einer von diesesn Logikwitzen basierend auf dem Umstand, dass das logische *oder* (`a | b`) auch *wahr* ist, wenn sowohl `a` als auch `b` *wahr* ist.

```
class(sleep$group)
#> [1] "factor"
```

(Was es mit `factor` auf sich hat sehen wir ein paar Abschnitte weiter)

Zusätzlich gibt es `typeof()`, eine Funktion, die so speziell ist, dass ich sie in meinen ~4 Jahren R erst neulich entdeckt habe, weil der exakte *Typ* eines Objekts meistens weniger relevant ist als die *Klasse*:

```
typeof(4)
#> [1] "double"
typeof(c(1, 2, 3))
#> [1] "double"
typeof(c("hallo", "welt"))
#> [1] "character"
typeof(sleep)
#> [1] "list"
typeof(sleep$extra)
#> [1] "double"
typeof(sleep$group)
#> [1] "integer"
```

6.1 Numeric (Zahlen und so)

Zahlen in R (und in den meisten anderen Programmiersprachen, beziehungsweise generell irgendwo, wo Maschinen rechnen) kommen in zwei Geschmacksrichtungen: **Integer** (ganze Zahlen) und **double** (Dezimalzahlen, Fließkommazahlen, *floating point numbers*).
Der Grund dafür hat damit zu tun, wie Computer intern Zahlen abbilden, binäres Zahlensystem, Bits, ihr wisst schon — komplizierter Kram wo sich kluge Menschen Dinge ausdachten, mit denen wir arbeiten können, wir aber nicht im Detail verstehen müssen.
Fließkommazahlen sind so gängig, dass R sogar eine einfache ganze Zahl wie `2` erstmal als *double*, also praktisch als `2.0` interpretiert, und wir explizit `2L` schreiben müssen, wenn wir *"2, aber als integer"* meinen. Wieso wir dafür `L` brauchen sei dahingestellt, aber nun ja, der Unterschied ist da:

```
# Beides "numeric"
class(2)
#> [1] "numeric"
class(2.5)
#> [1] "numeric"
```

```
# Aber…
typeof(2L)
#> [1] "integer"

# …und
typeof(2.5)
#> [1] "double"
```

Integers sind ziemlich unproblematisch, werden aber in der Praxis nicht häufig explizit genutzt. Fließkommazahlen (*double*) hingegen tauchen häufiger auf, weil Computer in den letzten Jahrzehnten *echt verdammt gut* darin geworden sind, mit Fließkommazhalen zu rechnen. Arithmetik mit integers ist auch okay, aber wenn eure Datensätze riesig und eure Statistik komplex ist, dann ist Geschwindigkeit von Rechenoperationen auf ein mal ein wichtiger Faktor.
Das Problem an der Sache ist nur leider, dass Fließkommazahlen seltsam sind. Nicht nur vom initialen Verständnis her, dazu empfehle ich euch herzlichst dieses schöne Video von Tom Scott, sondern auch für ganz reale Konsequenzen, über die wir stolpern können, wenn wir nicht aufpassen:

```
# Wurzel aus 2, ganz harmlos
sqrt(2)
#> [1] 1.414214

# Quadrierte Wurzel aus 2 ergibt 2, ja, kommt hin
sqrt(2)^2
#> [1] 2

# Das sollte ja dann…
sqrt(2)^2 == 2
#> [1] FALSE
```

Wait, what?

Und das ist der Grund warum Fließkommazahlen (*double*) seltsam sind.
Die kurze Version: $\sqrt{2}$ ist eine *irrationale Zahl*, das heißt sie hat *unendlich viele Nachkommastellen*. Computer können nur eine begrenzte Anzahl an Nachkommastellen speichern, weshalb das Resultat von einer Berechnung wie $\left(\sqrt{2}\right)^2$ zwar für alle praktischen Zwecke immer noch 2 ist, aber *irgendwie auch nicht*. Wenn wir mit Datensätzen arbeiten und darin rumrechnen dann sind solche kleinen Rundungsfehler egal, aber wenn wir uns auf Operatoren wie `==` verlassen, um berechnete Werte zu vergleichen, dann müssen wir vorsichtig sein.

An dieser Stelle ein kurzer Exkurs in die Numerik:
Die absolut kleinste Toleranz, die euer Computer für Fließkommazahlen berücksichtigt, könnt ihr euch mit `.Machine$double.eps` anzeigen lassen[2]. `.Machine` ist ein besonderes Objekt in R, dass Informationen zu eurem Computer (sprich eurer *Maschine*) sammelt.

```
# Wie groß ist die Abweichung vom erwarteten Ergebnis?
sqrt(2)^2 - 2
#> [1] 4.440892e-16
```

`4.4408920985e-16` ist Computer für $4.4408920985 \cdot 10^{-16}$, also ungefähr…

$$\frac{4.4408920985}{10000000000000000} \approx 0.00000000000000044$$

Das ist… ziemlich wenig, und im Alltag auch ziemlich egal, aber wie gesagt: Für R ist das ein Unterschied.

```
# Wie groß ist die Toleranz?
.Machine$double.eps
#> [1] 2.220446e-16

# Moment mal…
.Machine$double.eps * 2
#> [1] 4.440892e-16

# Wenn jetzt…
(.Machine$double.eps * 2) == (sqrt(2)^2 - 2)
#> [1] TRUE
```

Tatsache.
Wir könnten noch weiter damit rumspielen, aber als Lektion sollte eigentlich nur hängenbleiben, dass Zahlen in R gerne mal mehr sind, als euch in der Konsole angezeigt wird.

> **Note**
>
> One does not simply *round* floating point numbers — Programmer Boromir

Wenn ihr mal auf sowas stoßen solltet, dann verwendet am besten einfach die Funktion `round()` um eure Werte auf eine sinnvolle Anzahl Nachkommastellen zu runden:

[2] Habt ihr mal im Kontext von Computern oder Betriebssystemen/Software von "32bit" und "64bit" gehört? Da geht's tatsächlich genau um dieses Ding mit den Fließkommazahlen. 64 bit kann einfach mehr Nachkommastellen speichern als 32bit. Vergleiche dazu auch `.Machine$double.digits` auf einem 32bit gegen ein 64bit-Betriebssystem

```r
# Auf 5 Stellen gerundete Wurzel 2
round(sqrt(2), digits = 5)
#> [1] 1.41421

# Gerundetes Ergebnis von "Wurzel 2 hoch 2"
round(sqrt(2)^2, digits = 5)
#> [1] 2

# Literally close enough.
round(sqrt(2)^2, digits = 5) == 2
#> [1] TRUE
```

> **Note**
>
> Im Zweifelsfall einfach Genauigkeit opfern um den Verstand zu behalten

Theoretisch ist "numeric" für Zahlen eine *Klasse*, und *integer* und *double* sind die beiden *Typen*, aus denen die Klasse besteht.

6.2 Character (Buchstabenzeugs)

Characters sind *Strings* sind *irgendwas was aus mehr als nur Zahlen besteht* (zumindest meistens). Die Unterscheidung zwischen *numeric* und *character* ist intuitiv ziemlich einfach, und in eurer statistischen Praxis werdet ihr vermutlich meistens auf *numerics* treffen, wobei *characters* dann meistens nur für nominale Variablen (Gruppenzugehörigkeiten, Entscheidungen für A, B, C) gebraucht werden. Tatsächlich werden eure nominalen Variablen sogar eher als `factor` daherkommen, dazu dann der nächste Abschnitt.

Characters verhalten sich im Grunde wie Worte. Wir können sie aneinanderhängen, wir können sie vergleichen, aber wir können zum Beispiel keine Berechnungen damit durchführen:

```r
namen <- c("Lukas", "Tobias", "Christoph")
mean(namen)
#> Warning in mean.default(namen): argument is not numeric or logical:
#> returning NA
#> [1] NA

# Zahlen != Buchstaben
5 == "5"
#> [1] TRUE
```

```r
# Groß- / Kleinschreibung ist wichtig!
"Lukas" == "Lukas"
#> [1] TRUE
"Lukas" == "LUKAS"
#> [1] FALSE

# Strings aneinanderhängen
paste("Lukas", "hat", "Spass", sep = "_")
#> [1] "Lukas_hat_Spass"
```

Die Funktionen `paste` und `paste0` sind ziemlich praktisch wenn ihr mit Strings arbeitet, die werdet ihr früher oder später mal brauchen.
Ansonsten dürfte euch aufgefallen sein, dass man *Spaß* mit *ß* schreibt. Das ist korrekt. Allerdings zählt *ß* als Sonderzeichen, genauso wie Umlaute (*üöä*). R kann damit zwar prinzipiell umgehen, solange ihr das richtige *Encoding* verwendet, aber dennoch bietet es sich an auf Sonderzeichen in R-Code zu verzichten, um Inkompatibilität mit anderen vorzubeugen.

Um eure Einstellungen anzupassen und auf Nummer sicher zu gehen, öffnet die Einstellungen von RStudio und setzt die folgende Einstellung auf **UTF-8** (Unicode):

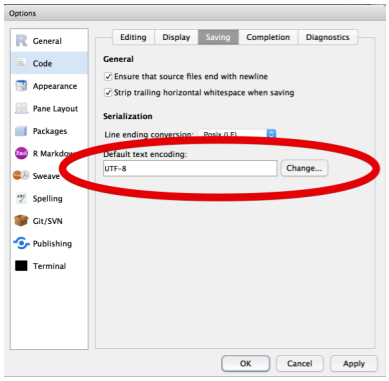


Figure 6.1: RStudio > Optionen > Code > Saving: UTF-8

Encoding ist so der einfachste Grund aus dem eure Scripte und Dokumente auf einmal kaputt aussehen, wenn ihr sie von einem Windows-Rechner an einen Mac oder eine Linux-Kiste schickt. Mac und Linux können sich wenigstens meistens auf Unix-Standards und Unicode eignen, aber Windows… Windows ist seltsam.

> **Note**
>
> Encoding ist kodifizierter Selbsthass, aber Unicode ist großer Spaß

6.3 Factor (Here be dragons)

Okay, der haarige Teil.
Die `factor`-Klasse in R ist unheimlich praktisch, aber auch ziemlich unintuitiv bei der ersten Verwendung. Das liegt nicht zuletzt daran, dass ein `factor` von aussehen meistens einfach aussieht wie ein `character`, aber nunmal kein `character` ist.
Factors haben zwei Bestandteile:

- `level`: Die *Merkmalsausprägung*, so wie R den `factor` sieht. Meistens *numeric*.
- `label`: (*Optional*) Die *Bezeichnung der Merkmalsausprägungen*, meistens `character`, für die bessere Lesbarkeit.

Ein Beispiel aus dem `sleep`-Datensatz:

```
# Die Vairable "group"
sleep$group
#>  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
#> Levels: 1 2

# …hat die Klasse "factor"
class(sleep$group)
#> [1] "factor"

# Und die levels…
levels(sleep$group)
#> [1] "1" "2"
```

Wir sehen, dass `group` die Merkmalsausprägungen (*levels*) `1` und `2` hat, aber das ist für uns möglicherweise nicht wirklich aussagekräftig. Wir können die Variable modifizieren, und einen schöneren `factor` daraus machen:

```
# Wir modifizieren nur die labels, nicht die level
sleep$group <- factor(sleep$group, levels = c(1, 2), labels = c("Medikament A", "Medikamen

# Jetzt werden uns unsere Labels angezeigt
sleep$group
#>  [1] Medikament A Medikament A Medikament A Medikament A Medikament A
#>  [6] Medikament A Medikament A Medikament A Medikament A Medikament A
#> [11] Medikament B Medikament B Medikament B Medikament B Medikament B
#> [16] Medikament B Medikament B Medikament B Medikament B Medikament B
#> Levels: Medikament A Medikament B
```

```r
# Und unsere unveränderten Levels
levels(sleep$group)
#> [1] "Medikament A" "Medikament B"

# Aber wir können den factor immer noch wie Zahlen behandeln
as.numeric(sleep$group)
#>  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
```

Mit `factor` können wir praktisch zwei Lagen an Informationen in nur einer Variable speichern, einmal numerische *levels* und einmal character *labels*. Die Levels sind die eigentlich wichtige Information und die Labels sind praktisch nur für uns zur besseren Lesbarkeit da, zum Beispiel bei Tabellen oder Grafiken.

6.4 Besondere Typen

Eure Daten kommen meistens von anderen, zumindest in den ersten Semestern eures Studiums. Meistens kommen eure Daten auch mit Fehlenden oder irgendwie kaputten Werten, mit denen ihr ohne Weiteres nichts anfangen könnt.

6.4.1 Fehlende Werte: `NA`

Vermutlich der wichtigste Datentyp, der euch begegnen wird. `NA` steht für *Not Available* und heißt, dass es an dieser Stelle einfach keinen Wert gibt. In einem Fragebogen wäre das zum Beispiel eine nicht ausgefüllte Frage, und das heißt für euch, dass ihr ohne Weiteres keine Annahme über diese Wert machen könnt. `NA` heißt nicht "da ist nichts", sondern eher "da könnte was sein, aber ich weiß nicht".
Das ist auch der Grund, warum der Mittelwert nicht funktioniert, wenn da `NA` drinstecken:

```r
mean(c(1, 2, NA, 4, 5, NA, 7))
#> [1] NA
```

Klar könnten wir einfach annehmen, dass die fehlenden Werte `3` und `6` sind, aber das wissen wir nunmal nicht, und da R in der Regel nicht rät, sagt es halt auch "weiß nicht" in Form von `NA`.
In solchen Fällen müsst ihr explizit `NA` ignorieren:

```r
mean(c(1, 2, NA, 4, 5, NA, 7), na.rm = TRUE)
#> [1] 3.8
```

Ihr könnt auch mit der Funktion `is.na` prüfen, ob ihr fehlende Werte habt. Beachtet, dass `==` zum vergleichen nicht funktioniert!

```
zahlen <- c(1, 2, NA, 4, 5, NA, 7)

is.na(zahlen)
#> [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE

# Alle 'zahlen', für die is.na() _nicht_ TRUE ist
zahlen[!is.na(zahlen)]
#> [1] 1 2 4 5 7

# An welcher Position sind die NAs?
which(is.na(zahlen))
#> [1] 3 6
```

6.4.2 Leere Werte: `NULL`

Wenn `NA` fehlende Werte sind, was soll dann `NULL` sein?
Naja, ich merke mir das immer ungefähr so:

- `NULL`: Da ist *nichts*, also so wirlich nichts, und ich weiß das auch!
- `NA`: Da ist zwar *nichts*, aber ich hab keine Ahnung ob da nicht doch was sein sollte ¯_( )_/¯

`NULL` wird euch vermutlich weniger häufig begegnen als `NA`, zumindest in Datensätzen. Ansonsten taucht `NULL` eher bei R-Funktionen als *default argument* auf, also ein Argument einer Funktion, das nicht gesetzt ist, außer ihr setzt es explizit. Das klingt jetzt etwas abstrakt, aber wir werden im Laufe dieser Einführung vermutlich noch Beispiele dafür sehen.

> **Note**
>
> `NULL` ist leer, und zwar mir Sicherheit
> `NA` ist leer, aber man weiß es nicht so recht

6.4.3 To `Inf` and BeyoNaNd!

Habt ihr schonmal durch 0 geteilt? Oder überlegt was 0^0 ist?
Das ist die Ecke, in der `Inf` und `NaN` auftauchen.

`Inf` und `-Inf` stehen erstmal nur für ∞ und $-\infty$ und sind Rs Weg euch zu sagen, dass ihr da gerade den Bereich der alltagstauglichen Zahlen überschritten habt.
Nehmt mal folgendes Beispiel:

```
# 2 hoch 10… geht noch
10^10
#> [1] 1e+10

# Auch das…
10^100
#> [1] 1e+100

# Okay, aber jetzt…
10^1000
#> [1] Inf
```

Das ist R einfach zu viel, bzw. es ist eurem Computer generell zu viel.
Kurzer reminder: `1e10` ist Computer für $1 \cdot 10^{10}$, also eine 1 mit 10 Nullen, also…

$$1e10 = 1 \cdot 10^{10} = 10000000000$$

Dementsprechend könnt ihr euch vorstellen, wie groß 10^{1000} wäre, und R macht solche Späße nicht mit und sagt einfach `Inf`.
Wenn ihr `Inf` oder `-Inf` in euren Ergebnissen seht, dann solltet ihr nur wissen, dass es da ein entweder *viel zu großes* oder *viel zu kleines* Ergebnis gab.

Und dann ist da noch die Sache mit `NaN`.
`NaN` steht für *not a number* und passiert dann, wenn ihr irgendwas mathematisch fragwürdiges macht, wie zum Beispiel 0 durch 0 teilen:

```
0 / 0
#> [1] NaN
```

Das ist mathematisch nicht definiert, und wieso das so ist und mehr dazu findet ihr zum Beispiel bei Numberphile gut erklärt.
Hier solltet ihr auch nur wissen, dass es das gibt und dass ihr es im Zweifelsfall vermeiden wollt, wenn es in euren Ergebnissen auftaucht.

6.5 Tabellen: `data.frame`

All eure Datensätze im Studium kommen in Tabellenform.
Tabellen in R sind im Grunde nichts anderes als Listen von Vektoren mit gleiche Länge: Der `sleep`-Datensatz zum Beispiel besteht aus drei Vektoren der Länge 20, und jede Spalte verhält sich wie ein Vektor mit bestimmten Typen.
Um sich einen Überblick über einen Datensatz zu verschaffen empfiehlt sich die Funktion `str` (lies *structure*), oder auch `head`:

```
# Die ersten paar Zeilen
head(sleep)
#>   extra        group ID
#> 1   0.7 Medikament A  1
#> 2  -1.6 Medikament A  2
#> 3  -0.2 Medikament A  3
#> 4  -1.2 Medikament A  4
#> 5  -0.1 Medikament A  5
#> 6   3.4 Medikament A  6

# Nur die ersten 2 Zeilen
head(sleep, n = 2)
#>   extra        group ID
#> 1   0.7 Medikament A  1
#> 2  -1.6 Medikament A  2

# Struktur des Datensatzes
str(sleep)
#> 'data.frame':    20 obs. of  3 variables:
#>  $ extra: num  0.7 -1.6 -0.2 -1.2 -0.1 3.4 3.7 0.8 0 2 ...
#>  $ group: Factor w/ 2 levels "Medikament A",..: 1 1 1 1 1 1 1 1 1 1 ...
#>  $ ID   : Factor w/ 10 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
```

Das Output von `str` sagt euch alles, was ihr braucht:

- Die *Klasse* des Objekts, hier ein `data.frame`, das Tabellenformat
  - Die Anzahl der Zeilen (`20 obs.`), und Spalten (`3 variables`)
- Die Spalten der Tabelle mit den ersten Werten
  - `extra`: Numerisch (`num`)
  - `group`: `factor` mit 2 Merkmalsausprägungen (`w/ 2 levels`), die *Labels* und die *Levels*
  - `ID`: `factor` mit 10 *Labels* (`"1", "2", "3" …`) und *Levels* (`1 2 3 4 …`)

Später werden wir noch andere Klassen für Tabellen sehen, die `data.frame` erweitern bzw. etwas aufhübschen, namentlich wird das `tbl_df` bzw. `tibble` sein, aber dazu müssen wir uns erst *Packages* ansehen.

6.6 Prüfen & Konvertieren

Was wir im Abschnitt zu `factor` am Ende mit `as.numeric` gemacht haben fällt unter *Coercion*, und heißt, dass Werte eines Typs in einen anderen Typ konvertiert werden sollen. Das Gegenstück dazu wäre `is.numeric`, was nachsieht, ob eine Variable bereits *numeric* ist.

```
as.numeric("5")
#> [1] 5
as.numeric(5)
#> [1] 5

as.character(c(2, 5, 4, 3))
#> [1] "2" "5" "4" "3"
```

Es gibt etliche solcher Konvertierungsfunktionen in R, manche mehr oder weniger nützlich, aber nun gut, sie sind da:

```
# Römische Zahlen
as.roman(2017)
#> [1] MMXVII

# Hexadezimal
as.hexmode(255)
#> [1] "ff"

# Logische Werte
as.logical(0)
#> [1] FALSE
as.logical(1)
#> [1] TRUE
```

7 Packages

Fast jede Software hat Erweiterungen irgendeiner Art. Manche haben *extensions*, andere *plug-ins*, wieder andere haben *add-ons*. Unterschiedliche Terminologie für dasselbe Prinzip: Mehr Features durch Erweiterungen anderer Leute.
Bei Programmiersprachen heißt sowas meistens *library* oder *package*[1].

R hat sowas natürlich auch, als populäre open-source Software. Hier heißt sowas *packages* und besteht aus Funktionen, die andere Leute für bestimmte Anwendungsfälle geschrieben haben, und durch ein Verteilungssystem verfügbar machen, sodass wir alle sie benutzen können.
Der *Kern* von R wird auch **base** genannt und umfasst die wichtigstens Grundfunktionen — mit denen kommen wir auch schon relativ weit, wir können zum Beispiel problemlos diverse Statistiken berechnen und sogar Visualisierungen machen, aber wir wollen natürlich mehr, einfacher, schneller und besser.

7.1 Installieren, Laden, Updaten

Packages ladet ihr aus dem Internet runter, woraufhin sie ggf. kompiliert und in eure R-library eingepflegt werden müssen.
Das klingt kompliziert, und deshalb passiert das auch alles automatisch!
N Wir installieren ein package mit einem einfachen Befehl in der Konsole:

```
install.packages("ggplot2")
```

> **Note**
>
> Windows-BenutzerInnen: Wenn ihr Antivirus-Software benutzt (Norton, Kaspersky, whatever) **kann es sein** dass diese euch beim installieren stört. Ihr bekommt dann wenig aussagekräftige Fehler. **Schaltet eure Antivirus-Software aus** wenn ihr Probleme bei der Installation habt.

Achtet darauf, dass ihr **Groß- und Kleinschreibung beachtet** habt, und dass der Name des packages in `" "` steht (wie ein `character`).
Wenn ihr Enter gedrückt habt sollte R anfangen loszurödeln, vielelicht gehen auch einige

[1] Die Ruby-Leute und ihre *Gems* seien mal dahingestellt

Fenster mit Fortschrittsbalken auf, wenn ihr Windows benutzt.
Das schöne an diesem Befehl ist, dass es auch direkt alle packages installiert, die wir in unserem package benutzen: Sogenannte *dependencies*. Eine dependency (*Abhängigkeit*) ist in diesem Kontext ein package, das von einem anderen package gebraucht wird um zu funktionieren. Wir benutzen im package `tadaatoolbox` zum Beispiel auch die packages `dplyr`, `pixiedust` und `sjlabelled`, deswegen sollte der Befehl diese packages auch gleich mitinstallieren.

Alternativ könnt ihr rechts in RStudio im "Packages"-Tab den "Install"-Button drücken, den namen des packages (`tadaatoolbox`) eingeben, und dann macht RStudio im Hintergrund genau dasselbe Spielchen mit `install.packages()`.

In diesem Fenster ist auch von *Repositories* die Rede. Damit ist der Web-Adresse gemeint, von der die packages geladen werden sollen. RStudio sollte da automatisch die schnellste Quelle auswählen, aber wenn ihr mal in die Verlegenheit kommt euch entscheiden zu müssen, versucht am besten folgende Adresse:

`https://cloud.r-project.org`

Und da packages auch nur Software sind, und Software auf dem aktuellen Stand gehalten werden will, bietet es sich an sporadisch (spätestens alle paar Monate) mal den Update-Button zu drücken und einfach alles zu aktualisieren, was aktualisiert werden kann.
Das könnt ihr auch aus der Konsole heraus machen indem ihr den Befehl `update.packages(ask = FALSE)` ausführt. R fragt euch dann, ob ihr sicherheitshalber R neustarten wollt, das könnt ihr tun oder auch nicht, aber wenn ihr es *nicht* tut, dann solltet ihr *auf jeden Fall* die R-Session neu starten nachdem ihr alle Updates gemacht habt (`RStudio -> Session -> Restart R`).

Wieso? Nun ja, packages werden von R *geladen*, das heißt verfügbar gemacht, und wenn ihr ein package ersetzt (was beim Update passiert), dann zieht ihr damit R praktisch den Boden unter den Füßen weg und es ist sauer weil Dinge anders sind, als sie eben noch waren.

Dieses "verfügbar machen" sieht übrigens so aus:

```
library(ggplot2)
```

Wenn ihr diesen Befehl ausgeführt hat, dann läd R für die aktuelle Session das package und ihr könnt die Funktionen darin benutzen.
Normalerweise beginnen eure R-Scripte mit einer Reihe von `library()`-Befehlen um eure Analyse vorzubereiten und alle benötigten packages zu laden, da ihr diesen Schritt jedes mal wiederholen müsst, wenn ihr eine neue R-Session starten (z.B. beim Neustart von RStudio, Computerneustart etc.).
Außerdem gebt ihr so euren KommilitonInnen eine gute Gelegenheit abzuschätzen, was in eurem Script so passiert, wenn sie direkt erkennen können, welche packages ihr dafür benutzt habt.

Es gibt *tausende* R-packages, und die meisten davon sind für euch vollkommen uninteressant, aber einige wiederum sind so dermaßen praktisch, dass wir sie uns hier im Detail anschauen.

7.1.1 Quellen

Die wichtigste Quelle für R-packages ist das erwähnte *CRAN*, kurz für *"**C**omprehensive **R** **A**rchive **N**etwork"*. Packages müssen diverse Anforderungen erfüllen, um auf CRAN publiziert zu werden, was eine gewisse Hürde darstellt. Deswegen gibt es diverse packages, die gerade in frühen, potenziell noch nicht ausgereiften Versionen an anderen Stellen verfügbar gemacht werden.
Die wohl populärster dieser Sekundärquellen ist **GitHub**.

Optional könnt ihr packages auch direkt von *GitHub* installieren, was insbesondere dann interessant ist, wenn das package noch jung und experimentell ist.
In euren normalen Projekten solltet ihr euch nicht auf GitHub-packages verlassen, sondern nach Möglichkeit ausschließlich packages von CRAN benutzen, aber wenn euch nach Abenteuer ist, dann fühlt euch frei:

```
# Wir brauchen das devtools package
install.packages("remotes")

# remotes package laden
library(remotes)

# install_github ist eine Funktion aus dem remotes package
install_github("tadaadata/tadaatoolbox")
```

7.1.2 Maintenance

Packages installieren ist einfach, aber wie jede Software wollen auch R-packages auf dem neusten Stand gehalten werden. Oder zumindest auf einem "nicht total veraltet"-Stand.
Updaten ist ziemlich einfach:
Entweder ihr klickt im "Packages"-Tab von RStudio auf den "Update"-Button, wählt alle Packages aus und installiert die Updates so, oder ihr gebt in der Konsole folgendes ein:

```
update.packages(ask = FALSE)
```

…dann rödelt R die Updates durch. RStudio wird euch an dieser Stelle fragen, ob ihr die R-Session vorher neustarten wollt – das könnt ihr tun, aber er könnt damit auch warten bis ihr alle Updates installiert habt anstatt für jedes einzelne Update die Session neuzustarten.
Wichtig ist auf jeden Fall, dass ihr die R-Session neustartet bevor ihr weiter arbeitet.

> **Note**
>
> Wenn ein package geladen (`library()`) ist während ihr Updates durchführt, **muss** die R-Session danach neugestartet werden! Wenn ein package aktualisiert wird während es geladen ist macht das R sehr traurig und Fehler treten auf.
> RStudio –> Session –> Restart R

Solltet ihr mal ein packages löschen wollen, aus welchem Grund auch immer, dann geht das entweder auch über den Package-Tab in RStudio (das Kreuzchen rechts neben dem Namen des packages), oder ihr gebt folgendes in der Konsole ein:

```
remove.packages("tadaatoolbox")
```

...um das package `tadaatoolbox` zu deinstallieren.
Aber wieso solltet ihr das tun wollen.
Wir haben so viel Arbeit in das package gesteckt.
Wieso nur.
Ihr Monster.

7.1.2.1 Tabellen

Für Kreuztabellen, wie wir sie relativ häufig in QM1 brauchen für unsere nominal- und ordinalskalierten Statistiken:

Oder einfache Häufigkeitstabellen:

Und für lineare Modelle, in deutlich schöner als `summary()`:

```
model <- lm(zufrieden ~ alter * berufsvorstellung, data = qmsurvey)
```

7.1.2.2 Plots

Klassiker: Histogramm mit Normalverteilungskurve:

7.2 Das `tidyverse`

Das *tidyverse* ist eine Ansammlung von packages, die alle mehr oder weniger gut miteinander auskommen und auf ähnliche Art zu benutzen sind.
Eine Übersicht und ganz viel Dokumentation und Beispiele findet ihr auf https://tidyverse.org.

Es gibt ein *catchall* R-package, das die wichtigstens packages für euch installiert:

```
install.packages("tidyverse")
```

Ihr könnt danach entweder das package laden und habt damit die wichtigsten Funktionen parat, oder ihr ladet die packages nach Bedarf einzeln.
Ich würde zu letzterem raten, weil ihr so eher ein Gefühl dafür bekommt welches package für welche Funktionen zuständig ist, und nebenbei geht es auch ein bisschen schneller.

Hier eine kurze Übersicht über die wichtigsten *tidyverse*-packages/Funktionen für alltägliche Aufgaben:

- `%>%`: Der *pipe*-Operator aus *magrittr*, der von den meisten der packages *re-exportiert* wird
- **ggplot2**: Für Visualisierungen
- **dplyr**: Datenmanipulation
  - `mutate`: Neue Variablen erstellen/ bestehende Verändern
  - `select`: Variablen auswählen
  - `filter`: Datensätze filtern
  - `group_by`: Gruppieren…
  - `summarize`: Und zusammenfassen
- **tidyr**: Datenmanipulation
  - `gather`: Konvertiert von *wide* in *long* format (mehrere Spalten zu zwei zusammenfassen)
  - `spread`: Gegenstück, konvertiert von *long* in *wide* format

7.3 Die `tadaatoolbox`

Die `tadaatoolbox` ist das Produkt von Lukas' und Tobis Frustration mit diversem Kleinkram in R, wie etwa der Optik von statistischen Testoutput.
Im Fokus steht in erster Linie schnelles, einfaches Output mit möglichst wenig Aufwand. Zusätzlich bietet die Toolbox einige Sammelfunktionen um beispielsweise diverse nominal- oder ordinalskalierte Statistiken auf einen Rutsch anzuzeigen, optimiert auf Output in RMarkdown-Dokumenten (Siehe [Berichte]).

7.4 Addendum: "Installier mal alles"

Hier ein Stück Code, das ihr im Zweifelsfall einfach copypasten könnt.
Es sollte euch so ziemlich alles oder zumindest das meiste an (vorerst) relevanten packages installieren, und dient mehr so der Vollständigkeit.

```
install.packages("magrittr")
install.packages("ggplot2")
install.packages("dplyr")
install.packages("tidyr")
install.packages("stringr")
install.packages("devtools")
install.packages("forcats")
install.packages("readr")
install.packages("lubridate")
install.packages("purrr")
install.packages("readxl")
install.packages("haven")
install.packages("rvest")
install.packages("scales")

install.packages("cowplot")
install.packages("ggrepel")
install.packages("psych")
install.packages("DescTools")
install.packages("ggthemes")
install.packages("hrbrthemes")
install.packages("rmarkdown")
install.packages("viridis")
install.packages("RColorBrewer")
install.packages("nortest")
install.packages("plotly")
install.packages("car")
install.packages("afex")
```

8 Die Hilfe

Wir haben bereits Seiten wie rdrr.io erwähnt, wo ihr euch Dokumentation anschauen könnt. Unter *Dokumentation* verbirgt sich übrigens nichts weltbewegendes, in der Regel geht es nur darum wie einzelne Funktionen zu benutzen sind. Diese Seiten sind im Wesentlichen aufgebohrte Versionen der Hilfe, die R von Haus aus mitbringt — allerdings mit bells & whistles in Form von schönerer Optik, erweiterter Suchfunktion, Verlinkungen etc.

Diie Hilfe findet ihr auf der rechten Seite in RStudio, unter dem `"Help"`-Tab:

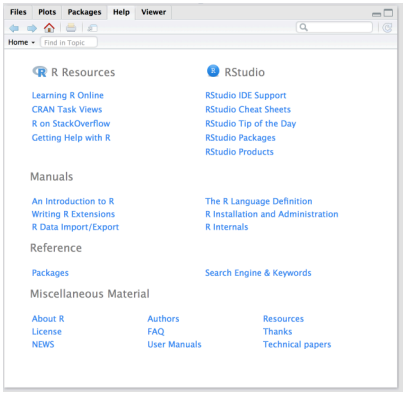


Figure 8.1: Die Startseite der R-Hilfe

In die Suchleiste oben rechts könnt ihr Suchbegriffe eingeben. Denkt daran, dass die Hilfe in erster Linie auf englisch verfügbar ist.

8.1 Format

Die Dokumentation von Funktionen hat immer das gleiche Format mit bestimmtem Pflichtbereichen und mehreren optionalen Abschnitten.

- **Description**: Was macht die Funktion?
- **Usage**: Wie benutzt man die Funktion?
- **Value**: Was kommt dabei raus?
- **See Also**: Welche anderen Funktionen sind verwandt?
- **Examples**: Anwendungsbeispiele

8.2 Sprache

Die R-Hilfe ist primär auf englisch verfügbar.
Wenn euch das überrascht, dann geht mal vor die Tür.
Zusätzlich ist die Hilfe meistens von den AutorInnen der jeweiligen Packages/Funktionen geschrieben, was zwar den Vorteil hat, dass sie in der Regel zumindest inhaltlich korrekt ist, aber für Laien manchmal nur so mäßig verständlich ist, was da jetzt eigentlich genau gesagt wird.

Häufig werden Begriffe verwendet, die für euch womöglich eher fremd erscheinen, weil ihr vermutlich keinen Hintergrund in der Programmierung / Numerik / whatever habt, aber keine Angst. Früher oder später lernt ihr die Begriffe, die relevant sind, und wenn ihr mal was so *gar nicht* versteht, dann fragt ihr eben einfach bei TutorInnen nach, googlet den Quatsch, oder ignoriert es einfach und versucht halt so weiter zu kommen.

9 Datenimport

Im Laufe eures Studiums (und vermutlich darüberhinaus) werdet ihr sehr viel Zeit damit verbringen Daten aus verschiedenen Quellen in die Statistiksoftware eurer Wahl (oder auch nicht eurer Wahl, aber der eurer Arbeitstselle) zu quetschen.
Das funktioniert mal mehr und mal weniger einfach, denn je nachdem wie die Originaldaten aussehen, kann das mitunter anstrengend bis deprimierend werden.
Saubere (*tidy*) Daten sehen immer gleich aus, aber unsaubere Daten sind alle auf ihre eigene Art unsauber. Mal fehlen Variablenbeschriftungen, mal sind da Umlaute durch Encoding kaputtgegangen, manchmal werden numerische Werte als *character* interpretiert und manchmal sind fehlende Werte mit irgendwelchen willkürlichen Werten kodiert (z.B. `-99` anstatt `NA`).

Vielleicht wundert ihr euch auch, wieso dieses Kapitel erst so spät in dieser Einführung auftaucht. Das liegt primär daran, dass ihr unter Umständen ein ausreichend großes Repertoire an Grundlagen braucht, um Daten auf alle Fälle sauber eingelesen zu bekommen.
In vielen Fällen, und besonders in QM, ist das Ganze noch relativ überschaubar und eure TutorInnen können entsprechende Hilfestellung bieten, aber *irgendwann* seid ihr auf euch allein gestellt, und dann macht ein bisschen Bonus-Wissen hier und da den Unterschied zwischen einem anstrengenden Nachmittag voller Leid und Schmerz oder 10 Minuten Probiererei und schnellem Erfolg.

Wenn ihr Daten von eurer Festplatte einlesen wollt, und ihr keine Ahnung habt wie Dateipfade funktionieren, was euer *Home Ordner* ist, was beispielsweise `~/Documents` sein soll oder wie ihr rausfindet, *wo* ihr gerade auf euren Computern seid, dann lest euch das bitte selber an.
Auch hier liefert RStudio jedenfalls im "Files"-Tab entsprechende Orientierungshilfe:

Das rot umrandete ist der Pfad zum Projektordner, in R würde ich den also so eingeben müssen:

```
"~/repos/tadaadata/r-intro-book"
```

Wobei die Tilde `~` eine Abkürzung für das Home-Verzeichnis ist.

9.1 Quellen

Da in QM nur SPSS und R benutzt werden, werdet ihr vermutlich meistens auf Datensätze aus SPSS (`.sav`) stoßen. R kann zwar SPSS-Daten einlesen, aber SPSS kann mit R-Daten

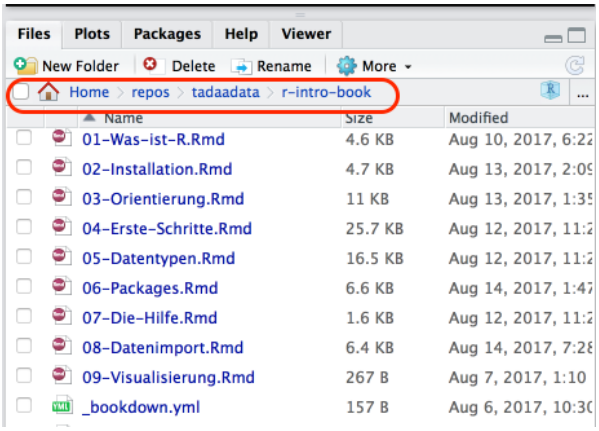


Figure 9.1: RStudio Filebrowser im Projekt dieser Einführung

nichts anfangen. Außerdem beinhalten SPSS-Datensätze auch ein bisschen Metadaten, wie zum Beispiel Labels für eure Variablen oder nominalskalierte Variablen, die wir in R dann für bessere Optik benutzen können — andere Formate wie Textdateien (`.csv`, `.txt`, *plain text*) sind spartanischer und haben sowas nicht.

Die einfachste Option ist meistens die RStudio-Funktio zum Datenimport, aber auch hier solltet ihr erstmal wissen, wo eure Daten herkommen und ggf. über die ein oder andere Eigenart bescheid wissen.

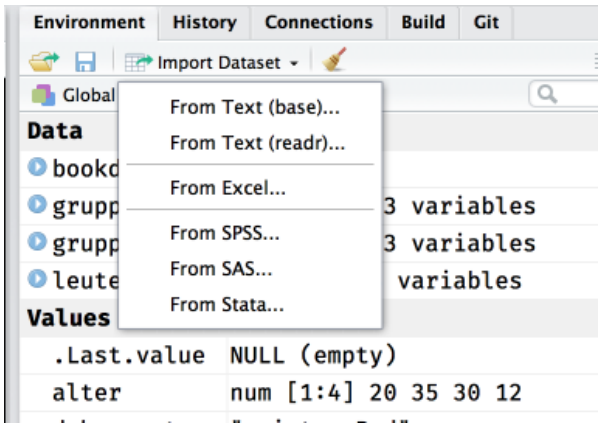


Figure 9.2: RStudio Import-Tool

Bei den Textdateien sind mit `base` und `readr` die beiden unterschiedlichen Möglichkeiten gemeint, mit denen wir Daten einlesen können, aber mehr dazu im entsprechenden Abschnitt.

Eine Sache noch zum Encoding: Um kaputte Umlaute und andere Krämpfe zu vermeiden bietet es sich an, **überall alles immer** auf Unicode bzw. **UTF-8** zu stellen wenn ihr *irgendwo* nach

Encoding gefragt werdet.

9.1.1 Roher Text (`.csv`, `.txt`)

- Benötigte packages: `readr`
- Anstrengend? Entweder alles super oder Riesenkrampf

Einfacher Text (*plain text*) ist die einfachste Möglichkeit Datensätze zu speichern bzw. zu übertragen, da Text so ziemlich der kleinste gemeinsame Nenner jeder gängigen Software ist. **CSV** heißt auch nur *"comma separated values"*, und wird euch vermutlich noch häufiger begegnen. Eine CSV-Datei könnt ihr mit jedem beliebigen Texteditor öffnen (ihr müsst dafür kein Office rauskramen, auf Windows tut es auch das Notepad), und ihr seht dann vermutlich sowas in der Art:

```
"extra","group","ID"
0.7,"1","1"
-1.6,"1","2"
-0.2,"1","3"
-1.2,"1","4"
-0.1,"1","5"
3.4,"1","6"
```

Das Prinzip ist ziemlich einfach: In der ersten Zeile stehen die Variablennamen, und in jeder folgenden Zeile steht jeweils der Wert der zugehörigen Variable, getrennt durch ein Komma.
Die Schwierigkeit kommt dann, wenn die Werte zum Beispiel Text enthalten, der wiederum ein Komma enthalten kann, und der Wert nicht richtig in Anführsungszeichen gesetzt ist. Es gibt auch noch Varianten mit Tabs statt Kommata als Trennzeichen, das wäre dann strenggenommen *TSV* (ihr dürft raten wofür das *T* steht).
Textformate sind also ziemlich einfach um eure Daten zu speichern oder zu verschicken, aber es ist auch sehr fragil, sobald mal irgendwo ein `"` fehlt oder zu viel ist, wird's kompliziert.

Zum lesen und schreiben empfehle ich herzlichst das `readr`-package mit den Funktionen `read_csv`, `write_csv` etc. zu benutzen, die sind weniger anfällig für Murks als die base-Standardfunktionen mit gleichem Namen aber `.` statt `_` (`read.csv`, `write.csv`).

Als Beispiel laden wir mal diesen schönen *Game of Thrones*-Datensatz:

```
library(readr)
gotdeaths <- read_csv("data/got_deaths.csv")

head(gotdeaths)
```

Hier habe ich `col_types = cols()` nur benutzt, um das Output zu unterdrücken. Ihr könnt über dieses Argument aber auch manuell spezifizieren, welchen Typ jede Spalte haben soll, damit eure Daten explizit so eingelesen werden, wie ihr sie erwartet.
Die schmutzigen Details gibt's natürlich in der Hilfe: `?read_csv` und online.

9.1.2 SPSS (`.sav`)

- Benötigte packages: `haven`.
- Anstrengend? Manchmal.

```
library(haven)

ngo <- read_spss("data/NGO.SAV")
```

Die Funktionen `read_sav` und `read_spss` sind identisch.

9.1.3 R (`.rds`, `.rda` & `.RData`)

- Benötigte packages: Keins (Base R reicht, optional `readr` als Alternative)
- Anstrengend? Nope, alles tutti.

Der wohl einfachste und dankbarste Anwendungsfall: Von R zu R.
Hier habt ihr zwei Möglichkeiten: `.rds` und `.rda` (auch `.RData`): Generell scheint `.rds` die präferierte Option zu sein.

9.1.3.1 `.rds`

Daten einlesen ist simpel:

```
qmsurvey <- readRDS("data/qm_survey_ss2017.rds")

tibble::as_tibble(qmsurvey)
```

Wir benutzen hier das `tibble` package nur, damit der Datensatz kompakter angezeigt wird. Das ist für euch keine Notwendigkeit, aber ich empfehle es in der Regel gerne, weil euch so nicht die Konsole vollgeklatscht wird, wenn ihr euch euren Datensatz mal schnell anschauen wollt.

Daten speichern auch:

```r
saveRDS(datensatz, "pfad/zur/datei.rds")
```

Hier zum Beispiel der Datensatz zur Tutoriumsteilnahme, den ihr von https://public.tadaa-data.de/data/participation.rds runterladen könnt:

```r
participation <- readRDS("~/Downloads/participation.rds")
```

Alternativ könnt ihr das `readr`-package benutzen. Die Funktion daraus sieht fast gleich aus, und macht auch exakt das gleiche wie `readRDS`. Der einzige Grund für `read_rds` ist die Konsistenz der Funktionsnamen.

```r
library(readr)

participation <- read_rds("~/Downloads/participation.rds")
```

9.1.3.2 `.rda`, `.RData`

Nein, ihr benutzt nicht `save` und `load` für einzelne Datensätze.

Bei `.rda` bzw. `.RData`-Dateien ist zu beachten, dass diese den Namen des Objekts gleich mitspeichern, das heißt ihr müsst den eingelesenen Datensatz keinen Namen geben — der kommt schon mit der Datei.
Theoretisch kann so eine Datei auch mehrere Variablen enthalten, und wenn ihr zum Beispiel RStudio schließt und wieder öffnet, dann werden in der Zwischenzeit auch eure Variablen der aktuellen Session in Form einer `.RData`-Datei im Projektordner abgelegt und beim nächsten Start wieder eingelesen.

```r
load("pfad/zur/datei.rda")

# Oder…
load("pfad/zur/datei.RData")
```

Speichern:

```r
save(datensatz, file = "pfad/zur/Datei.rda")
```

9.1.4 Excel (`.xlsx`)

- Benötigte packages: `readxl`
- Anstrengend? Manchmal. Aber wenn, dann *richtig*.

Wenn ihr ein sauberes (i.e. ohne Schnickschnak) Spreadsheet habt in dem auch wirklich nur eure Werte drinstehen, dann ist das Ganze recht simpel und ihr seid mit `readxl` auch gut bedient.
Wenn ihr einen dreckigen Haufen dampfender Menschenverachtung vor euch habt, dann… viel Spaß.
Es gibt da das ein oder andere Projekt für die komplexeren Fälle, aber wenn ihr die Möglichkeit habt, macht es euch so einfach (und rechteckig) wie möglich.

9.1.5 Google Sheets

- Benötigte packages: `googlesheets`
- Anstrengend? Meistens geht's ganz gut.

Wenn euch Excel zu unpraktisch ist, dann bietet sich Google Sheets an. Es ist kostenlos, einfach und ausreichend mächtig für alles, was ihr so vorhaben könntet — mitunter weil ihr für alle komplexeren Sachen sowieso R benutzen wollt. Sheets ist praktisch wie Excel, nur halt in der CloudTM und von Google, aber für überschaubare Datensammlungen reicht's auf alle Fälle. Die Tutoriumsteilnahmedaten haben wir da auch gesammelt, und da das Sheet immer an der selben Stelle ist muss man einfach nur den Code zum einlesen und auswerten erneut ausführen und schon hat man eine mehr oder weniger selbstupdatende Analyse. Nett.

In besagtem Projekt sieht das zum Beispiel so aus:

```
participation_1 <- gs_title("Tutoriumsteilnehmer") %>%
  gs_read(ws = "WS1516", range = cellranger::cell_cols(1:7))
```

Zuerst müsst ihr aber die Authentifizierung mit eurem Google-Account abhandeln:

```
library(googlesheets)

gs_ls()
```

Mit diesem Befehl zeigt euch das package all eure Google Sheets an nachdem es euch nach einem Login gefragt hat, von da aus könnt ihr dann weiterarbeiten. Mehr Informationen und Beispiele gibt's in der Vignette.

9.2 Daten angucken & sauber machen

> Happy families are all alike; every unhappy family is unhappy in its own way.
>
> — Leo Tolstoy
>
> and every messy data is messy in its own way - it's easy to define the characteristics of a clean dataset (rows are observations, columns are variables, columns contain values of consistent types). If you start to look at real life data you'll see every way you can imagine data being messy (and many that you can't)!
>
> — Hadley Wickham (answering 'in what way messy data sets are messy') R-help (January 2008)

Um festzustellen, ob eure frisch eingelesenen Daten auch brauchbar sind, empfiehlt sich ein Blick in die Daten via `View(daten)` bzw. Über einen Klick auf den Datensatz im *Environment*-Tab von RStudio (das da oben rechts).
Zusätzlich ist auch hier natürlich `str()` praktisch, um zum Beispiel schnell zu überprüfen, ob eure Variablen auch alle die Klasse haben, die ihr erwartet (alle Zahlen sind *numeric* und Nominaldaten sind *factor* oder wenigstens *character*).

Es gibt da kein one-size-fits-all Rezept zur Datenbereinigung, denn jeder Datensatz ist auf seine eigene Art dreckig. Ihr könnt nur darauf hoffen, dass euer konkreter Anwendungsfall gut googlebar ist, oder ihr sowas ähnliches schonmal gemacht habt.
Die gängigsten Probleme sind recoding, umbenennen oder zusammenfassen, und das meiste lässt sich entweder mit `dplyr`, `sjmisc` oder ggf. `tidyr` erledigen. Versucht es erstmal innerhalb des *tidyverse*, das ist vermutlich angenehmer als zusammengehackte Google-Lösungen. Aber auch hier, wie gesagt: Je nachdem was ihr vorhabt. Mehr dazu findet sich in [Data Munging].

Datenbereinigung ist entweder sehr einfach oder sehr komplex, oder irgendwo dazwischen. In diesem Sinne: Learning by example und so.

Quellen

A Ressourcen

A.1 Datensätze

Es gibt viele freie Quellen für schöne Datensätze zum Üben oder rumspielen, und einige packages bringen auch entsprechende Datensätze mit, die geeignet sind um bestimmte Funktionen auszuprobieren.

A.1.1 In R / Packages

- Gängige Standarddatensätze:
  - `mtcars`
  - `sleep`
  - `attitude`
- `palmerpenguins::penguins`: Pinguine!
- `dplyr::starwars`: *Star Wars*-stuff mit *list-columns*
- `babynames::babynames`: Naja, baby names.
- `tadaatoolbox::ngo`: NGO-Datensatz aus QM (bzw. aus dem *Kähler*)

A.1.2 In der freien Wildbahn

- data.world: Hier kommt z.B. der *Game of Thrones* deaths-Datensatz her

A.1.3 Tadaa, Datasets!

Hier liegen diverse Datensätze aus unseren Projekten, wie z.B. die QM-Surveys und die Tutorienteilnahme.

- data.tadaa-data.de

A.2 Bücher

- "R for Data Science" – Hadley Wickham
- "Advanced R" – Hadley Wickham
- ggplot2 – Hadley Wickham
- "Tidyverse Cookbook"
- "R Programming for Data Science" – Roger Peng
- "Exploratory Data Analysis with R" – Roger Peng
- "Top 50 ggplot2 Visualizations"
- "R Cookbook"
- Mehr Bücher

A.2.1 Kollaboration und Organisation

A.2.1.1 Git & GitHub

- happygitwithr.com – Jenny Bryan
- ohshitgit

A.3 Kurse & Workshops

A.3.1 Kostenlose Tutorials/Videos

- **RStudio Webinars**
- **Stat545**
- **LOTR Data**

A.4 Blogs

- Aggregator: R-Bloggers

Siehe auch: Blogs in `blogdown`

A.5 Community

- Stackoverflow
- Twitter: `#rstats`

A.6 Dokumentation

- R help (inoffizielle bookdown-Version)
- rdrr.io
- Package-specific
  - tidyverse.org