EDV1

Max Brede

2021-01-04

Contents

1 Vorwort **5**

2 Lehrplan **7**
2.1 Semesterplan 7
2.2 Übungsformat 7
2.3 Lehrziele für jede Sitzung 7
2.4 Prüfungsleistung 8

3 Vorlesung I - Rste Schritte **9**
3.1 Organisatorisches 9
3.2 Einführung 10
3.3 Grundlegende Rechenoperationen 14
3.4 Ausdrücke, Funktionen, Argumente 16
3.5 Objekte 16
3.6 Hausaufgabe 22

4 elementare Datenverarbeitung **23**
4.1 Organisatorisches 24
4.2 Vektoren 25
4.3 Indizierung 26
4.4 Systematische Wertefolgen erzeugen 29
4.5 Daten transformieren 30
4.6 Einfache deskriptiv-statistische Kennwerte 31

5 `tidyverse` und `tibbles` **33**
5.1 Organisatorisches 33
5.2 Pakete benutzen 34
5.3 Datensätze erstellen und ergänzen 34
5.4 Datensätze sortieren und indizieren 38
5.5 Vorlesung 43

6 Faktoren und Aggregation **45**
6.1 Organisatorisches 45
6.2 Faktoren 46

6.3 Daten einlesen I 49
6.4 deskriptive Kennwerte 50

7 Aggregation und Filemanagement **53**
7.1 Organisatorisches 54
7.2 Aggregation 55
7.3 Daten einlesen II und zusammenfügen 58

Chapter 1

Vorwort

Dieses mit `bookdown` erstellte Dokument ist das über das Wintersemester 2020 hinweg wachsende Skript zur Übung "PSY_B_11-2: Computerunterstützte Datenanalyse I" der CAU zu Kiel.

Chapter 2

Lehrplan

2.1 Semesterplan

2.2 Übungsformat

Die Übung soll zur Hälfte in 45-minütigen Sitzungen im Vorlesungsformat zur Vorstellung der Funktionen und zur anderen Hälfte als 45-minütige praktische Übung stattfinden. Es wird pro Übungs-Sitzung ein Übungszettel ausgegeben, der mit Hilfe der in der Zugehörigen Vorlesung besprochenen Funktionen bearbeitet werden können soll. Diese Zettel sollen nach der jeweiligen Vorlesung für die Übungen vorbereitet werden, in denen der Zettel dann besprochen und mögliche Fragen geklärt werden. Nach den Übungssitzungen haben die Studierenden dann eine Woche Zeit, zusätzliche Hausaufgaben zu bearbeiten.

Eine Ausnahme von diesem Ablauf ist die erste Sitzung, in der organisatorisches und Grundlagen in 90 minütigem Vorlesungsstil besprochen werden sollen. Auch nach dieser Sitzung werden aber Übungszettel und Hausaufgaben ausgegeben.

2.3 Lehrziele für jede Sitzung

Die Studierenden können nach dem Absolvieren der Übung…

Einheit 1

- Vor- und Nachteile von R und RStudio nennen und diese installieren.
- erklären was Funktionen und was Argumente sind.
- die Hilfe benutzen.
- das Environment von R benutzen um Objekte anzulegen und zu löschen.

Einheit 2

- Vektoren erstellen, tranformieren und indizieren.
- verschiedene Datenformate in R erstellen, benutzen und in einander überführen.

Einheit 3

- Pakete installieren und benutzen.
- mit Hilfe des "`tidyverse`" Datensätze erstellen, ergänzen, sortieren und indizieren.

Einheit 4

- Faktoren erstellen.
- Daten auf Gruppenebene aggregieren.
- Häufigkeiten auszählen und tabellarisch darstellen.

Einheit 5

- Daten auf Gruppenebene noch besser aggregieren.
- Datensätze aus verschiedenen Formaten einlesen.

Einheit 6

- Datensätze kombinieren und pivotieren.
- eine Anzahl von Grafiken erstellen.

Einheit 7

- kompliziertere Grafiken erstellen.

Einheit 8

- die Funktionsschreibweise lesen und anwenden.
- erfolgreich an der Klausur teilnehmen.

2.4 Prüfungsleistung

Die Studierenden **müssen** während des Semesters die nach den Übungssitzungen ausgegebenen Hausaufgaben innerhalb einer Woche sinnvoll bearbeitet abgeben.

Mit maximal einer nicht sinnvoll bearbeiteten Serie werden die Studierenden zur Klausur am Ende des Semesters zugelassen.

Chapter 3

Vorlesung I - Rste Schritte

3.1 Organisatorisches

Semesterplan

Einheit	Vorlesung	Übungswoche	Thema
1	2.11.20	keine Übung	Grundlagen und Begriffe
2	16.11.20	KW 48	Vektoren und Indizierung
			Datenformate erstellen und transformieren
3	30.11.20	KW 50	Pakete installieren und benutzen
			Datensätze erstellen und ergänzen können
			Datensätze sortieren und indizieren können
4	14.12.20	KW 1	Faktoren
			deskriptive Kennwerte
			Aggregation I
5	11.01.21	KW 3	Aggregation II
			In- und Export von Datensätzen
6	25.01.21	KW 5	Grafische Darstellungen I
7	08.02.21	KW 7	Grafische Darstellungen II
8	22.02.21	keine Übung	Puffer
			Probeklausur

Übungsablauf

Die Übung wird zur Hälfte als Vorlesung, zur anderen Hälfte in Kleingruppen abgehalten.

Die Daten sind im Kalender und im Semesterplan im Olat ersichtlich.

Prüfungsleistung

Die Prüfungsleistung in dieser Veranstaltung besteht aus:

1. Dem *regelmäßigen Bearbeiten* und *Bestehen* von Hausaufgaben. Diese werden über das OLAT ausgeteilt und abgegeben, zu jeder Veranstaltung wird eine neue Serie herausgegeben. Das Bestehen der Hausaufgaben ist nötig, um zur Klausur zugelassen zu werden.
   - Als *Bestanden* gilt eine Serie, wenn alle Aufgaben **sinnvoll** bearbeitet wurden.
   - Unter *regelmäßigem Bearbeiten* versteht sich das Bestehen aller Serien **mit einer Ausnahme**.

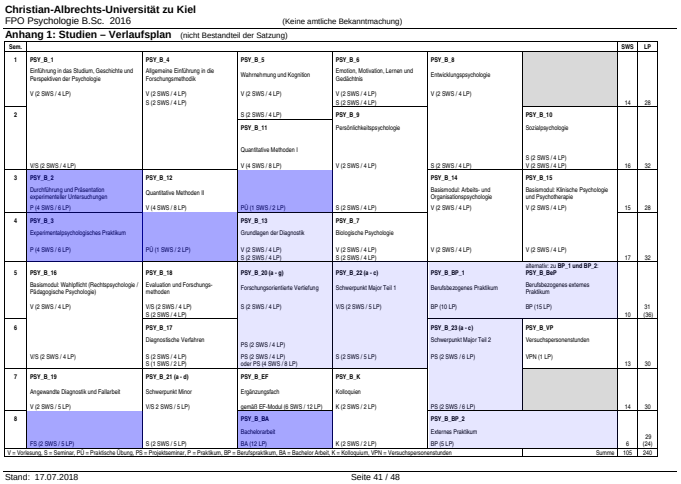
2. Im Klausurzeitraum findet an einem Tag eine praktische Prüfung statt.

3.2 Einführung

Rste Schritte

Diese Veranstaltung und das zugehörige Material sollen Ihnen einen Einstieg in das computergestützte Aufbereiten und Auswerten von empirischen Daten bieten. Dazu werden wir auf die von ihren Autoren als 'software environment for statistical computing and graphics' bezeichnete, freie Umgebung R zurückgreifen.

Wozu brauche ich das?



Christian-Albrechts-Universität zu Kiel
FPO Psychologie B.Sc. 2016 (Keine amtliche Bekanntmachung)

Anhang 1: Studien – Verlaufsplan (nicht Bestandteil der Satzung)

Sem.							SWS	LP
1	**PSY_B_1** Einführung in das Studium, Geschichte und Perspektiven der Psychologie V (2 SWS / 4 LP)	**PSY_B_4** Allgemeine Einführung in die Forschungsmethodik V (2 SWS / 4 LP) S (2 SWS / 4 LP)	**PSY_B_5** Wahrnehmung und Kognition V (2 SWS / 4 LP)	**PSY_B_6** Emotion, Motivation, Lernen und Gedächtnis V (2 SWS / 4 LP) S (2 SWS / 4 LP)	**PSY_B_8** Entwicklungspsychologie V (2 SWS / 4 LP)		14	28
2			S (2 SWS / 4 LP) **PSY_B_11** Quantitative Methoden I	**PSY_B_9** Persönlichkeitspsychologie		**PSY_B_10** Sozialpsychologie S (2 SWS / 4 LP) V (2 SWS / 4 LP)	16	30
	V/S (2 SWS / 4 LP)		V (4 SWS / 8 LP)	V (2 SWS / 4 LP)	S (2 SWS / 4 LP)			
3	**PSY_B_2** Durchführung und Präsentation experimenteller Untersuchungen P (4 SWS / 6 LP)	**PSY_B_12** Quantitative Methoden II V (4 SWS / 8 LP)	PÜ (1 SWS / 2 LP)	S (2 SWS / 4 LP)	**PSY_B_14** Basismodul: Arbeits- und Organisationspsychologie V (2 SWS / 4 LP)	**PSY_B_15** Basismodul: Klinische Psychologie und Psychotherapie V (2 SWS / 4 LP)	15	28
4	**PSY_B_3** Experimentalpsychologisches Praktikum P (4 SWS / 6 LP)	PÜ (1 SWS / 2 LP)	**PSY_B_13** Grundlagen der Diagnostik V (2 SWS / 4 LP) S (2 SWS / 4 LP)	**PSY_B_7** Biologische Psychologie V (2 SWS / 4 LP) S (2 SWS / 4 LP)	V (2 SWS / 4 LP)	V (2 SWS / 4 LP)	17	32
5	**PSY_B_16** Basismodul: Wahlpflicht (Rechtspsychologie / Pädagogische Psychologie) V (2 SWS / 4 LP)	**PSY_B_18** Evaluation und Forschungsmethoden V/S (2 SWS / 4 LP) S (2 SWS / 4 LP)	**PSY_B_20 (a - g)** Forschungsorientierte Vertiefung S (2 SWS / 4 LP)	**PSY_B_22 (a - c)** Schwerpunkt Major Teil 1 V/S (2 SWS / 5 LP)	**PSY_B_BP_1** Berufsbezogenes Praktikum BP (10 LP)	alternativ zu **BP_1 und BP_2**: **PSY_B_BaP** Berufsbezogenes externes Praktikum BP (15 LP)	10	34 (36)
6	V/S (2 SWS / 4 LP)	**PSY_B_17** Diagnostische Verfahren S (2 SWS / 4 LP) S (1 SWS / 2 LP)	PS (2 SWS / 4 LP) PS (2 SWS / 4 LP) oder PS (4 SWS / 8 LP)	S (2 SWS / 5 LP)	**PSY_B_23 (a - c)** Schwerpunkt Major Teil 2 PS (2 SWS / 6 LP)	**PSY_B_VP** Versuchspersonenstunden VPN (1 LP)	13	30
7	**PSY_B_19** Angewandte Diagnostik und Fallarbeit V (2 SWS / 5 LP)	**PSY_B_21 (a - d)** Schwerpunkt Minor V/S 2 SWS / 5 LP)	**PSY_B_EF** Ergänzungsfach gemäß EF-Modul (6 SWS / 12 LP)	**PSY_B_K** Kolloquien K (2 SWS / 2 LP)	PS (2 SWS / 6 LP)		14	30
8	FS (2 SWS / 5 LP)	S (2 SWS / 5 LP)	**PSY_B_BA** Bachelorarbeit BA (10 LP)	K (2 SWS / 2 LP)	**PSY_B_BP_2** Externes Praktikum BP (6 LP)		6	29 (34)
V = Vorlesung, S = Seminar, PÜ = Praktische Übung, PS = Projektseminar, P = Praktikum, BP = Berufspraktikum, BA = Bachelor Arbeit, K = Kolloquium, VPN = Versuchspersonenstunden						Summe	105	240

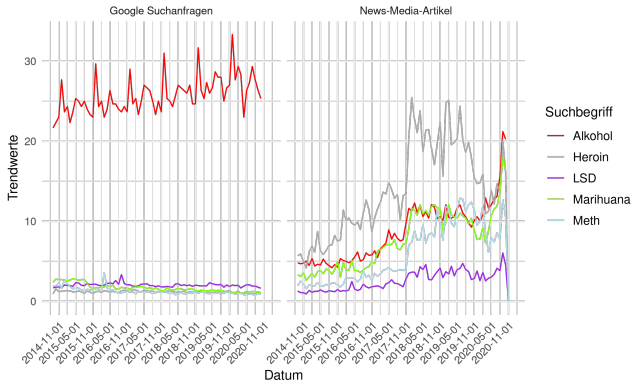
Stand: 17.07.2018 Seite 41 / 48

Warum R ?

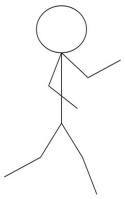
(...und nicht SPSS...)

	SPSS	R
Pro	einfache Bedienung	das 'CRAN' (Comprehensive R Archive Network)
	weit verbreitet	kostenlos
		macht was angewiesen ist
Contra	kann nicht alles	etwas Gewöhnung notwendig
	relativ kostenintensive Lizenzen	
	nimmt vieles ab	
	nicht beliebig erweiterbar	

Aber die viel wichtigeren Argumente: R kann **Alles**



R macht **Spass**



Literatur

Die Veranstaltung orientiert sich an:

1. Wollschläger (2016) . R kompakt.(Link aus dem Uni-Netz).
2. Grolemund and Wickham (2017) . R for Data Science (Link).

Installation & Verwendung

Es wird die Verwendung der grafischen Benutzeroberfläche RStudio empfohlen.

Beachten Sie, dass für die Verwendung von RStudio zuvor eine Basisinstallation von R erfolgen muss:

1. (R) herunterladen und installieren.
2. (RStudio) herunterladen und installieren.

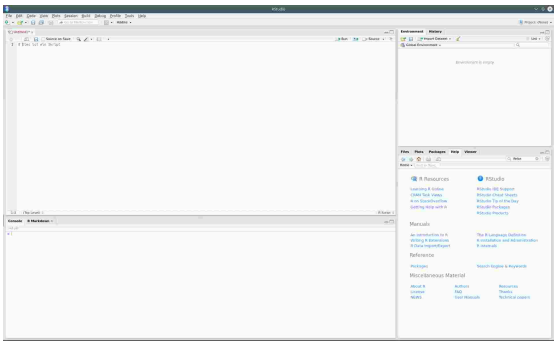


Figure 3.1: Benutzeroberfläche von RStudio. Oben links: Editor; unten links: Konsole; oben rechts: Environment bzw. History; unten rechts: Files, Plots, Help, etc.

Benutzeroberfläche RStudio

Allgemeine Hinweise

- Verwenden Sie die Konsole (unten links) nur für einzeilige Berechnungen beim "Ausprobieren"
- Verwenden Sie stets den Editor (oben links), um mehrzeilige Berechnungen direkt in ein Skript zu schreiben
- Kommentieren Sie Ihren Code ausreichend und sinnvoll mit Hilfe des `#`-Zeichens
- Speichern Sie Ihr Skript unter einem sinnvollen Namen in einem sinnvoll benannten Verzeichnis ab
- Speichern Sie regelmäßig mit Strg+S zwischen
- Eine einzelne Skript-Zeile (diejenige, in der sich der Cursor befindet) oder zuvor markierter Code lassen sich mit Strg+Enter ausführen
- In der Konsole bricht ESC die Eingabe ab

Zum besseren Verständnis

In diesem Skript enthalten die grau hinterlegten Zeilen R-Input, die weiß hinterlegten Zeilen den R-Output. Ein ganz einfaches Beispiel zum Ausprobieren: Die simple Berechnung von 1 + 1.

```
1 + 1
```

```
## [1] 2
```

Ausdrücke in der R-Konsole

Anweisungen in R funktionieren grundsätzlich über das Ausführen von Ausdrücken. Dabei werden Ausdrücke entweder durch Semikolons oder Zeilenum-

brüche beendet.

```
1 + 1; 2 + 2;
```

```
## [1] 2
```

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

```
1+1
```

```
## [1] 2
```

```
2+2
```

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

Kommentare

R bietet außerdem die Möglichkeit, im Code Anmerkungen zu machen, die beim Ausführen ignoriert werden. Diese werden mit einem #-Symbol eingeleitet.

```
1 + 1 ### +1 +1
```

```
## [1] 2
```

```
#Dies ist ein Kommentar
```

Nutzen Sie Kommentare innerhalb Ihrer Skripte, um Arbeitsschritte kenntlich zu machen und zu erklären. Die übersichtliche Gestaltung Ihrer Skripte ist von wirklich großem Vorteil bei der Arbeit mit R. Dies kann nicht oft genug betont werden.

3.3 Grundlegende Rechenoperationen

Addition, Subtraktion

```
2 + 3
```

```
## [1] 5
```

```
28 - 5
```

```
## [1] 23
```

Multiplikation, Division

```
2 * 21
```

```
## [1] 42
```

```
92 / 4
```

```
## [1] 23
```

Rechenregeln

```
1+1*1+1*(1+1)+1
```

```
## [1] 5
```

Wie man sieht, befolgt R die Punkt-vor-Strich-Regel und berücksichtigt Klammerung.

Potenz, Quadratwurzel ("squareroot"), Betrag ("absolute")

```
3^2
```

```
## [1] 9
```

```
sqrt(9)
```

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

```
abs(-42)
```

```
## [1] 42
```

Runden

```
pi
```

```
## [1] 3.141593
```

```
round(pi)
```

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

```
round(pi, digits=2)
```

```
## [1] 3.14
```

```
round(pi, digits=3)
```

```
## [1] 3.142
```

Aufgabe

```
round(pi, digits = 0) * 3 ### + 5
```

Was kommt raus?

A) `pi`
B) `14`

C) eine Fehlermeldung
D) `9`
E) `NULL`

3.4 Ausdrücke, Funktionen, Argumente

Funktionen & Argumente

In R werden sehr häufig *Funktionen* verwendet. Diese repräsentieren eine Reihe von Anweisungen, die beim Aufrufen mit spezifischen Parametern ausgeführt werden sollen. Diese Parameter werde in Form von *Argumenten* übergeben. Beispielsweise enthält die Funktion `round()` die nötigen Anweisungen, um eine Zahl zu runden. Hierfür erwartet `round()` die zu rundende Zahl und die Anzahl an Nachkommastellen auf die zu runden ist. Man schreibt immer *Funktionsname*(*Argumentliste*). Bei Funktionen müssen *immer* runde Klammern vorhanden sein, auch wenn keine einzelnen Argumente vorgegeben werden.

Es gibt *obligatorische Argumente*, ohne deren Übergabe das Aufrufen einer Funktion zu einer Fehlermeldung führt:

```
round(pi)
```

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

```
round() ### Funktionsaufruf ohne Argument
```

```
## Error in eval(expr, envir, enclos): 0 arguments passed to 'round' which requires 1 or 2 arguments
```

… und *optionale Argumente*:

```
round(pi, digits=3)
```

```
## [1] 3.142
```

```
round(pi, digits=pi)
```

```
## [1] 3.142
```

```
round(pi, digits=15)
```

```
## [1] 3.141593
```

Gibt man den Namen eines Arguments nicht an, entscheidet die Position in der Liste über die Interpretation des Arguments durch R. **Achtung: Fehlerquelle!**

```
round(1/42, 3)
```

```
## [1] 0.024
```

```
round(3, 1/42)
```

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

3.5 Objekte

Objekte sind für den späteren Gebrauch mit einem Namen versehene und im Arbeitsspeicher abgelegte Ergebnisse von Ausdrücken. Dabei ist Objekt der Überbegriff für eine Vielzahl von möglichen Datenstrukturen.

Ein paar Beispiele für Datenstrukturen in R:

- eindimensionale Vektoren (vector)

- mehrdimensionale Matrizen (matrix)
- Funktionen(function)

Objekte benennen

Wählen Sie kurze, aber aussagekräftige Objektnamen! Objektnamen dürfen dabei enthalten: Buchstaben, Zahlen, Punkte, Unterstriche

Achtung:

- Immer mit einem Buchstaben beginnen
- Groß-/Kleinschreibung ist relevant
- Keine anderen Sonderzeichen
- Keine durch R reservierte Namen von Funktionen, Konstanten, etc. (z.B. “mean”, “pi”, “if”, etc.) (im Zweifel Überprüfen mit `exists()`)

Hier nochmal der nachdrückliche Hinweis: Tun Sie sich selbst den Gefallen, Ihre Objekte eindeutig und nachvollziehbar zu benennen!

Zuweisungen an Objekte

Ergebnisse von Ausdrücken können benannten Objekten zugewiesen werden.

Dabei sind folgende Ausdrücke äquivalent:

```
firstObject = 42
42 -> firstObject
firstObject <- 42
```

Die letzte Möglichkeit stellt dabei die Beste im Hinblick auf Übersichtlichkeit und Eindeutigkeit dar.

Verwenden von Objekten:

Die Objektnamen können dann synonym zu ihrem Inhalt verwendet werden.

```
firstObject + 1; 42 + 1;
```

```
## [1] 43
```

```
## [1] 43
```

Objekte ausgeben

Um diese Ausgabe nachzuholen gibt es folgende Möglichkeiten:

```
print(firstObject)
```

```
## [1] 42
```

```
firstObject
```

```
## [1] 42
```

Diese beiden Versionen sind faktisch dieselbe, da das einfache Aufrufen eines Variablennamens implizit als ein Aufruf von `print()` interpretiert wird.

```
(object2 <- firstObject^2)
```

```
## [1] 1764
```

Bei Setzen eines Befehls in Klammern wird die durch ihn ausgelöste Änderung ausgegeben, im Beispiel die Zuweisung des Ergebnisses zum neuen Objekt `object2`.

Diese Methode ist eine gute Variante, Zwischenergebnisse regelmäßig zu kontrollieren.

Objekte anzeigen lassen

Alle Objekte im Workspace anzeigen lassen:

```
ls()
```

```
## [1] "a"           "firstObject" "object2"
## [4] "plan"
```

Diese Operation braucht man später nicht unbedingt, da alle angelegten Objekte auch im Environment-Tab in RStudio einsehen kann. Am Anfang kann diese Funktion aber helfen, sich über die Abläufe klar zu werden.

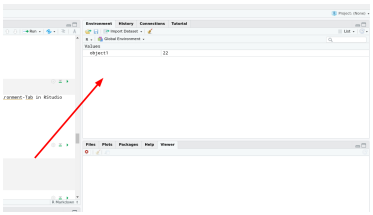


Figure 3.2: Environment

Objekte entfernen

Vorhandene Objekte lassen sich dann wie folgt entfernen:

```
ls()
```

```
## [1] "a"           "firstObject" "object2"
## [4] "plan"
```

```
rm(object2)
ls()
```

```
## [1] "a"           "firstObject" "plan"
```

Mit `rm(list=ls())` lassen sich alle Objekte aus dem Workspace entfernen.

```
ls()
```

```
## [1] "a"           "firstObject" "plan"
```

```
rm(list=ls())
ls()
```

```
## character(0)
```

Datentypen

In R, wie in so gut wie jeder anderen Sprache, werden Objekte in unterschiedliche Subtypen gegliedert, die sich auf die in ihnen gespeicherten Informationen beziehen:

Beschreibung	Beispiel	Datentyp
leere Menge	‘NULL‘	‘NULL‘
logische Werte	‘TRUE, FALSE, T, F‘	‘logical‘
ganze und reelle Zahlen	‘42‘	‘numeric‘
Buchstaben- o. Zeichenfolgen (immer in Anführungszeichen)	‘beware of the leopard.‘	‘character‘

Dabei ist das hier keine vollständige Liste, für den Anfang reicht sie aber.

`mode()` gibt den Datentyp des übergebenen Arguments aus (braucht man selten, hier nur für das Beispiel):

```
mode(answer)
```

```
## [1] "numeric"
```

```
mode('answer')
```

```
## [1] "character"
```

Datentypen konvertieren

`as.character(answer)` konvertiert den Datentyp des Objekts von `numeric` nach `character` ohne den ursprünglichen Eintrag von `answer` zu überschreiben.

```
mode(answer)
```

```
## [1] "numeric"
```

```
as.character(answer)
```

```
## [1] "42"
```

```
mode(answer)
```

```
## [1] "numeric"
```

Um das zu erreichen muss das Objekt überschrieben werden:

```
answer <- as.character(answer)
mode(answer)
```

```
## [1] "character"
```

Mit `answer` als character-Element lässt sich nicht mehr rechnen:

```
answer * 2
```

```
## Error in answer * 2: non-numeric argument to binary operator
```

Um das dann wieder zu ermöglichen muss das Objekt zurück nach numeric konvertiert werden:

```
answer <- as.numeric(answer)
mode(answer)
```

```
## [1] "numeric"
```

```
answer * 2
```

```
## [1] 84
```

Weitere Beispiele für Konvertierung:

```
as.numeric("42") ### konvertiert character nach numeric
```

```
## [1] 42
```

```
as.numeric(TRUE) ### konvertiert logical nach numeric
```

```
## [1] 1
```

```
as.logical(0)  ### konvertiert numeric nach logical
```

```
## [1] FALSE
```

```
as.logical(1)  ### konvertiert numeric nach logical
```

```
## [1] TRUE
```

```
as.logical(23)  ### konvertiert numeric nach logical
```

```
## [1] TRUE
```

```
as.logical("true")  ### konvertiert character nach logical
```

```
## [1] TRUE
```

Logische Werte, Operatoren und Verknüpfungen

Logische Vergleiche, Verknüpfungen und andere Operatoren:

Operator	Operation
‘==‘	ist gleich
‘!=‘	ist ungleich
>	ist größer
>=	ist größer gleich
<	ist kleiner
<=	ist kleiner gleich
‘¡	logisches NICHT
&	logisches UND
‘|‘	logisches ODER
‘isTRUE()‘	gibt an, ob übergebenes Argument TRUE ist

Das Ergebnis eines logischen Vergleichs sind logische Werte:
WAHR: `TRUE = T = 1`
FALSCH: `FALSE = F = 0`

Beispiele:

```
1 == 2
```

```
## [1] FALSE
```

```
1 != 2
```

```
## [1] TRUE
```

```
1 < 2
```

```
## [1] TRUE
```

```
1 >= 2
```

```
## [1] FALSE
```

```
1>2 & 1<=3
```

```
## [1] FALSE
```

```
2>1 | 1!=1
```

```
## [1] TRUE
```

```
6>5 & !(2<=1)
```

```
## [1] TRUE
```

```
isTRUE(1 == 1)
```

```
## [1] TRUE
```

```
(1 == 1)
```

```
## [1] TRUE
```

Aufgabe

Was kommt raus?

```
(2 > 1 & 1 < 3) | 1 != 1
```

A) TRUE
B) FALSE
C) NULL

Umgang mit Dezimalzahlen:

Was kommt hier raus?

```
0.1 + 0.2 == 0.3
```

A) TRUE
B) FALSE
C) NULL

```
0.1 + 0.2 == 0.3
```

```
## [1] FALSE
```

0.1 + 0.2 != 0.3?

'Falsches' Ergebnis ist Resultat von Repräsentation von Gleitkommazahlen im Speicher des Rechners.
Die Funktion `all.equal()` löst dieses Problem.

```
all.equal(target=0.1+0.2, current=0.3)
```

```
## [1] TRUE
```

Mit dem `tolerance`-Argument lässt sich der Bereich der akzeptabeln Unterschiede in Dezimalstellen angeben.

```
all.equal(target = 0.424242, current = 0.424243,
          tolerance = 1e-5)
```

```
## [1] TRUE
```

```
all.equal(target = 0.424242, current = 0.424243,
          tolerance = 1e-6)
```

```
## [1] "Mean relative difference: 2.357145e-06"
```

Hierbei fällt auf, dass bei Ungleichheit nicht `FALSE` sondern die Abweichung ausgegeben wird.

Um `all.equal` sinnvoll in logischen Operationen benutzen zu können wird `isTRUE` benötigt:

```
isTRUE(all.equal(target = 0.424242,
                 current = 0.424243,
                 tolerance = 1e-6))
```

```
## [1] FALSE
```

3.6 Hausaufgabe

Hausaufgabe: Erstellen eines R-Skripts

Schreiben Sie den dem folgenden Ablauf entsprechenden Code in ein R-*Skript* und führen Sie ihn von dort in der Konsole aus:

Erstellen Sie drei Objekte wie folgt:

- Als erstes ein Objekt namens *whatDoIDoThis* mit der Zahl 4 als Inhalt.
- Als zweites ein Objekt namens *text* mit dem Inhalt : “i_like_snake_case_better”.
- Als drittes ein Objekt namens *myFavouriteNumber* mit einer Zahl Ihrer Wahl als Inhalt.

Berechnen Sie nun den Mittelwert der Objekte mit numerischem Inhalt und legen Sie diesen in einem weiteren Objekt namens *manualMean* ab.

Lassen Sie sich in der Konsole durch eine Zeile in Ihrem Skript den Text `'I learned about the most important bugfixing tool'` ausgeben.

Speichern Sie anschließend das R-*Skript* unter ‘.R’ ab.

Chapter 4

elementare Datenverarbeitung

4.1 Organisatorisches

4.1.1 Semesterplan

Einheit	Vorlesung	Übungswoche	Thema
1	2.11.20	keine Übung	Grundlagen und Begriffe
2	16.11.20	KW 48	Vektoren und Indizierung
			Datenformate erstellen und transform
3	30.11.20	KW 50	Pakete installieren und benutzen
			Datensätze erstellen und ergänzen kö
			Datensätze sortieren und indizieren k
4	14.12.20	KW 1	Faktoren
			deskriptive Kennwerte
			Aggregation I
5	11.01.21	KW 3	Aggregation II
			In- und Export von Datensätzen
6	25.01.21	KW 5	Grafische Darstellungen I
7	08.02.21	KW 7	Grafische Darstellungen II
8	22.02.21	keine Übung	Puffer
			Probeklausur

4.2 Vektoren

4.2.1 Begriff

Im Kontext von R ist ein Vektor als eine sequentiell geordnete Menge von Werten und nicht als das gleichnamige mathematische Konzept zu verstehen.

4.2.2 Vektoren erzeugen

Leere Vektoren eines bestimmten Typs lassen sich mit dem Namen des Typs als Funktion und der Anzahl der gewünschten Stellen als Argument erstellen. z.B.:

```
numeric(5);
character(4);
logical(3);
```

```
## [1] 0 0 0 0 0
## [1] "" "" "" ""
## [1] FALSE FALSE FALSE
```

Um mehrere Daten in einer eindimensinalen Anordnung zu verketten wird die `c()`-Funktion benutzt. Die Argumente werden in Reihenfolge der Eingabe hintereinander angeordnet und können beliebigen Datentypen angehören.

```
c(1,2,3,4);
c('dies', 'ist', 'ein', 'Vektor');
c(T, F, T, F);
```

```
## [1] 1 2 3 4
## [1] "dies"   "ist"    "ein"    "Vektor"
## [1]  TRUE FALSE  TRUE FALSE
```

Das Ergebnis kann dann wie gewohnt in ein Objekt abgelegt werden.

```
numericVector <-c(4,2,4242,42)
```

Außerdem lassen sich mit dem `c()`-Operator mehrere Vektoren kombinieren.

```
additionToNumericVector <-  c(424242, 42400, 42000,
                              4224, 24)
(numericVector <- c(numericVector,
                    additionToNumericVector))
```

```
## [1]      4      2   4242     42 424242  42400  42000
## [8]   4224     24
```

4.2.3 Vektoren verwenden

Die Länge eines Vektors lässt sich mit der Funktion `length()` ausgeben.

```
length(numericVector)
```

```
## [1] 9
```

4.2.4 Datentypen in Vektoren

Bei dem Versuch Vektoren aus verschiedenen Datentypen anzulegen werden die Daten in den allgemeinsten Datentyp umgewandelt. Dabei gilt im Rahmen der Komplexität für die bisher vorgestellten Datentypen:

```
logical < numeric < character
```

```
mode(c(T, T, F));
mode(c(T, T, 0));
mode(c(T, 0, 'false'));
```

```
## [1] "logical"
## [1] "numeric"
## [1] "character"
```

In einem Vektor ist im Allgemeinen also immer nur ein Typ an Daten vertreten.

4.2.5 Aufgabe

Wenn ich `mode(c('TRUE',FALSE,1))` eingebe, dann...

A) ... wird `logical` ausgegeben
B) ... wird `vector` ausgegeben
C) ... wird `numerical` ausgegeben
D) ... wird `character` ausgegeben

4.3 Indizierung

4.3.1 Elemente indizieren

Die beim Erstellen eines Vektors angelegten Positionen der Werte werden in R implizit mit fortlaufenden Indizes versehen und gespeichert. Diese Indizes starten bei jedem Vektor mit 1 und enden mit der Länge desselben. Die einzelnen Elemente eines Vektors lassen sich über ihren Index mit dem `[]` - Operator aufrufen.

```
numericVector[4] ## 4. Element des Vektors numericVector.
```

```
## [1] 42
```

Wird ein Index über dem des letzten Eintrags eines Vektors aufgerufen, wird `NA` zurückgegeben.

```
numericVector[length(numericVector)+1]
```

```
## [1] NA
```

4.3.2 mehrere Elemente gleichzeitig indizieren

Es lassen sich auch mehrere Werte eines Vektors über die Indizierung über Zuhilfenahme eines anderen Vektors aufrufen. Dabei kann der Index-Vektor als Objekt vordefiniert oder dem `[]`-Operator direkt übergeben werden.

```
idx <- c(1,2,3,8)
numericVector[idx]
```

```
## [1]    4    2 4242 4224
```

```
numericVector[c(4,5,6,7)]
```

```
## [1]     42 424242  42400  42000
```

Der Index-Vektor kann dabei auch länger als der ursprüngliche Vektor sein, da mehrfacher Aufruf eines Index möglich ist.

```
numericVector ## 9 Werte
```

```
## [1]      4      2   4242     42 424242  42400  42000
## [8]   4224     24
```

```
idx <- c(1,1,2,2,3,3,4,4,5,5,6,6)
## 12 Aufrufe über die Indizes
numericVector[idx]
```

```
##  [1]      4      4      2      2   4242   4242     42
##  [8]     42 424242 424242  42400  42400
```

4.3.3 Elemente ausschließen

Durch das verwenden negativer Indizes wird das entsprechende Element von der Ausgabe ausgeschlossen.

```
numericVector[-3] ## Vektor ohne drittes Element
```

```
## [1]      4      2     42 424242  42400  42000   4224
## [8]     24
```

```
idx <- c(1,3,5,7,9)
## Vektor mit Ausnahme der in idx abgelegten Indizes:
numericVector[-idx]
```

```
## [1]     2    42 42400  4224
```

4.3.4 Elemente austauschen

Die Indizierung kann außerdem genutzt werden um Elemente eines Vektors zu ersetzen oder als alternative Methode zu oben vorgestelltem Kombinieren von Vektoren via `c(<Vektor1>, <Vektor2>)`.

```
numericVector
```

```
## [1]      4      2   4242     42 424242  42400  42000
## [8]   4224     24
```

```
numericVector[1] <- 12
numericVector
```

```
## [1]     12      2   4242     42 424242  42400  42000
## [8]   4224     24
```

```
numericVector[idx] <-idx
numericVector
```

```
## [1]     1     2     3    42     5 42400     7  4224
## [9]     9
```

```
length(numericVector)
```

```
## [1] 9
```

```
numericVector[c(10,11,12,13,14)] <- idx
numericVector
```

```
##  [1]     1     2     3    42     5 42400     7  4224
##  [9]     9     1     3     5     7     9
```

4.3.5 Elemente löschen

Elemente eines Vektors lassen sich nicht im eigentlichen Sinne löschen, man kann aber sehr wohl das Objekt in dem der Vektor abgelegt ist mit einer verkürzten Version überschreiben.

```
numericVector <- numericVector[-idx]
numericVector
```

```
## [1]     2    42 42400  4224     1     3     5     7
## [9]     9
```

4.3.6 Logische Operatoren

Verarbeitungsschritte mit logischen Operatoren treten häufig bei der Auswahl von Teilmengen von Daten sowie der Recodierung von Datenwerten auf, zwei häufigen Prozeduren in der statistischen Auswertung

4.3.7 Logischer Vergleich von Vektoren

Wie vorher Einzelwerte kann man auch Vektoren in logischen Vergleichen verwenden.

```
age <- c(17, 30, 30, 24, 23, 21)
age < 24
```

```
## [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

```
age >= 18
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Dabei werden logische Werte als Ergebnis für den Vergleich jeden Wertes ausgegeben.

Die vorher gezeigten logischen Verknüpfungen lassen sich genauso anwenden

```
 ## Alle Werte die mindestens 18 und kleiner als 30 sind
(age >= 18) & (age < 30)
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
## Alle Werte die kleiner als 18 oder mindestens 30 sind
(age < 18) | (age >= 30)
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

4.3.8 Logische Vektoren

Die mit `sum()` gebildete Summe eines logischen Vektors gibt einem die Anzahl der wahren Werte im Vektor aus, da `TRUE` für die Berechnung in eine 1 und `FALSE` in eine 0 transformiert wird.

```
res <- !((age < 18) | (age >= 30))
sum(res)
```

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

4.3.9 Logische Vergeiche von Vektoren

Zwei **gleichlange** Vektoren lassen sich auch mit Hilfe logischer Operatoren vergleichen.

```
age2 <- c(19, 31, 29, 24, 30, 22)
age == age2
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE
```

Und natürlich lassen sich hier alle vorher besprochenen logischen Operatoren anwenden:

```
age == age2
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
age < age2
```

```
## [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE
```

```
age != age2
```

```
## [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
```

4.3.10 Logische Indizierung

Indizierung funktioniert auch mit logischen Vektoren. Dabei wird im Indexvektor für jeden Wert des indizierten Vektors angegeben, ob dieser ausgewählt derden soll oder nicht. Eine einfach Methode zur Auswahl von Teilmengen von Elementen die einem bestimmten Kriterium entsprechen.

```
(res <- age < 24)
```

```
## [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

```
age[res]
```

```
## [1] 17 23 21
```

Der Indexvektor muss nicht vorher als Objekt angelegt werden.

```
age[age<24]
```

```
## [1] 17 23 21
```

4.3.11 Logische Indizierung und fehlende Werte

Versucht man Vektoren mit fehlenden Werten zu erzeugen, stößt man auf folgendes Problem:

```
age3 <-  c(20, 23, 32, NA, 19, 27)
(idx <- age3 < 24)
```

```
## [1]  TRUE  TRUE FALSE    NA  TRUE FALSE
```

```
age3[idx]
```

```
## [1] 20 23 NA 19
```

4.3.12 Umgang mit fehlenden Werten bei logischer Indizierung

Der fehlende Wert wird in den Index-Vektor und die Indizierung weitergetragen.
Umgehen lässt sich dieses Problem mit der `which()`-Funktion, die die Positionen aller `TRUE`-Werte des ihre übergebenen Arguments als numerischen Vektor ausgibt.

```
(idx <- which(idx))
```

```
## [1] 1 2 5
```

Dieser kann dann wieder zur Indexierung benutzt werden.

```
age3[idx]
```

```
## [1] 20 23 19
```

4.4 Systematische Wertefolgen erzeugen

4.4.1 Numerische Sequenzen erstellen

In R lassen sich durch einen Doppelpunkt Zahlensequenzen in Einserschritten zwischen einem Start- und Endwert erstellen.

```
1:20
20:1
-10:10
-(1:20)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
## [17] 17 18 19 20
##  [1] 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5
## [17]  4  3  2  1
##  [1] -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1
## [13]   2   3   4   5   6   7   8   9  10
##  [1]  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10 -11 -12
## [13] -13 -14 -15 -16 -17 -18 -19 -20
```

Sequenzen mit anderen Schrittgrößen lassen sich mit der `seq()`-Funktion erstellen.

Dabei lässt sich entweder die Schrittgröße angeben:

```
seq(from = 0, to = 42, by = 6)
```

```
## [1]  0  6 12 18 24 30 36 42
```

```
seq(from = 0, to = 42, by = 5) ## Endpunkt nicht erreicht
```

```
## [1]  0  5 10 15 20 25 30 35 40
```

Oder die gewünschte Anzahl der Werte in der Sequenz:

```
seq(from = 0, to = 42, length.out = 8)
```

```
## [1]  0  6 12 18 24 30 36 42
```

```
seq(from = 0, to = 42, length.out = 6)
```

```
## [1]  0.0  8.4 16.8 25.2 33.6 42.0
```

Bei zweiterer Methode oder bei Angabe einer nicht-ganzzahligen Schrittgröße können auch nicht zur Indizierung geeignete Dezimalzahlen entstehen.

```
seq(from = 0, to = 1, by = 0.1)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Mit dem `along` Argument lässt sich außerdem eine Sequenz in der Länge eines übergebenen Vektors erstellen.

```
age <-c(16, 43, 30, 22,  7, 36)
seq(along = age)
```

```
## [1] 1 2 3 4 5 6
```

```
seq(from = 10, to = 100, along = age)
```

```
## [1]  10  28  46  64  82 100
```

4.4.2 Wertefolgen wiederholen

```
someValues <- c(42, 16, 12)
 ## Vektor 5-mal wiederholen
rep(someValues, times = 5)
```

```
##  [1] 42 16 12 42 16 12 42 16 12 42 16 12 42 16 12
```

```
 ## Jeden Wert des Vektors 5 mal wiederholen
rep(someValues, each = 5)
```

```
##  [1] 42 42 42 42 42 16 16 16 16 16 12 12 12 12 12
```

```
 ## Jeden Wert so oft wie in times individuell angegeben wiederholen
rep(someValues, times = c(42, 1, 5))
```

```
##  [1] 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
## [17] 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
## [33] 42 42 42 42 42 42 42 42 42 42 16 12 12 12 12 12
```

4.5 Daten transformieren

4.5.1 Werte sortieren

`sort()` sortiert je nach Angabe in auf- oder absteigender Reihenfolge

```
sort(someValues, decreasing = FALSE)
```

```
## [1] 12 16 42
```

```
sort(someValues, decreasing = TRUE)
```

```
## [1] 42 16 12
```

`order()` macht im Prinzip dasselbe, gibt aber statt des sortierten Vektors die Indizes in entsprechender Ordnung aus.

```
someValues
```

```
## [1] 42 16 12
```

```
(idx <- order(someValues, decreasing = F))
```

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

```
someValues[idx]
```

```
## [1] 12 16 42
```

Dies funktioniert auch für `character`-Vektoren in alphabetischer Ordnung

```
(someCharacters <-  c("Z", "D", "L", "O", "l", "n", "e", "N", "t", "R"))
```

```
##  [1] "Z" "D" "L" "O" "l" "n" "e" "N" "t" "R"
```

```
sort(someCharacters, decreasing = F)
```

```
##  [1] "D" "e" "l" "L" "n" "N" "O" "R" "t" "Z"
```

Sind Zahlen an der ersten Stelle in `character`-Vektoren vertreten, werden diese vor das Alphabet sortiert.

```
someCharacters <- c(
  "42 is fairly overused",
  "India",
  "Zulu",
  "Whiskey",
  "42",
  "a string of characters",
  "Tango",
  "not a number",
  "1"
)
sort(someCharacters, decreasing = F)
```

```
## [1] "1"                     "42"
## [3] "42 is fairly overused" "a string of characters"
## [5] "India"                 "not a number"
## [7] "Tango"                 "Whiskey"
## [9] "Zulu"
```

4.6 Einfache deskriptiv-statistische Kennwerte

```
age <- c(6, 60, 44, 56, 8, 58, 87, 8, 55, 83)
IQ <- c(91, 104, 109, 92, 90, 101, 99, 93, 89, 118)

mean(age)  ## Mittelwert
var(age)  ## Varianz (korrigiert)
sd(age) ## Streuung (korrigiert)
```

```
## [1] 46.5
## [1] 895.6111
## [1] 29.92676
```

```
N <- length(age)
sd(age)/sqrt(N) ## SEM
```

```
## [1] 9.463673
```

```
sqrt((N-1) / N) * sd(age) ## unkorrigierte Streuung
```

```
## [1] 28.39102
```

```
cov(x=age, y=IQ, method="pearson")  ## Kovarianz
```

```
## [1] 167.6667
```

```
cor(x=age, y=IQ, method="pearson")  ## Korrelation
```

```
## [1] 0.5875238
```

Chapter 5

`tidyverse` und `tibbles`

5.1 Organisatorisches

5.1.1 Semesterplan

Einheit	Vorlesung	Übungswoche	Thema
1	2.11.20	keine Übung	Grundlagen und Begriffe
2	16.11.20	KW 48	Vektoren und Indizierung
			Datenformate erstellen und transformieren
3	30.11.20	KW 50	Pakete installieren und benutzen
			Datensätze erstellen und ergänzen können
			Datensätze sortieren und indizieren können
4	14.12.20	KW 1	Faktoren
			deskriptive Kennwerte
			Aggregation I
5	11.01.21	KW 3	Aggregation II
			In- und Export von Datensätzen
6	25.01.21	KW 5	Grafische Darstellungen I
7	08.02.21	KW 7	Grafische Darstellungen II
8	22.02.21	keine Übung	Puffer
			Probeklausur

5.2 Pakete benutzen

5.2.1 das CRAN und Paketinstallation

Wie in der ersten Sitzung schon angedeutet, ist eine der größten wenn nicht *die* größte Stärke von R das Comprehensive R Archive Network.

Zum einen werden dort die Installationsdateien für die neuesten R-Versionen gehostet, für uns viel praktischer ist aber, dass dort auch fast unzählige Pakete zugänglich gemacht werden. Mit diesen Paketen oder auch `packages` lässt sich der Funktionsumfang von R beliebig erweitern.

Diese Pakete lassen sich mit der naheliegend benannten Funktion `install.packages` installieren. Ein für uns wichtiges Paket, das eine ganze Sammlung von nützlichen Paketen beinhaltet, die wir ab jetzt regelmäßig benutzen werden, ist das `tidyverse`.

Bitte installieren Sie diese Sammlung mit dem folgenden Code:

```
install.packages('tidyverse')
```

5.2.2 Pakete laden

Wenn das Paket installiert ist, lässt es sich ganz einfach mit `library` in den genutzten Namensraum laden.

Pakete, die nicht vor der Benutzung geladen wurden, können nicht benutzt werden!

Da wir es gleich benutzen werden, laden wir schon einmal das `tidyverse`

```
library(tidyverse)
```

5.3 Datensätze erstellen und ergänzen

5.3.1 Datensätze

Bisher haben wir nur einfache Vektoren benutzt um Daten auszudrücken. Das geht zwar, solange man nur Daten eines Formats und in überschaubarer Anzahl betrachtet, sobald wir aber an richtige experimentelle Kontexte denken, reicht das nicht mehr.

Deswegen gibt es in R so genannte *'rechteckige Datensätze'*, die Vektoren als Spalten zu einer Tabelle kombinieren. Die Spalten können dabei unterschiedliche Datenformate haben.

5.3.2 Datensätze erstellen

Datensätze werden in R `data.frames` genannt und können mit `data.frame()` erstellt werden.

Hier werden wir aber direkt auf die Basis-R Funktion verzichten und die etwas hübschere Funktion `tibble()` aus dem `tidyverse` nutzen.

Ein `tibble` ist prinzipiell ein `data.frame`, hat aber ein paar zusätzliche quality-of-life-features, die den Umgang mit ihnen vereinfachen.

5.3.3 Beispiel-Datensatz

Wir erstellen jetzt erstmal unseren ersten Datensatz:

```
my_1st_tibble <- tibble(
  index = 1:10,
  name = c('Agnes Nitt',
           'Samuel Vimes',
           'Esme Weatherwax',
           'Gytha Ogg',
           'Horace Worblehat',
           'Mustrum Ridcully',
           'Fred Colon',
           'Leonard of Quirm',
           'Havelock Vetinari',
           'Reg Shoe'
  ),
  group = c(1,2,1,1,3,3,2,2,2,2)
)
```

5.3.4 `tibble`

Der Datensatz sieht dann so aus:

```
my_1st_tibble
```

```
## # A tibble: 10 x 3
##    index name              group
##    <int> <chr>             <dbl>
##  1     1 Agnes Nitt            1
##  2     2 Samuel Vimes          2
##  3     3 Esme Weatherwax       1
##  4     4 Gytha Ogg             1
##  5     5 Horace Worblehat      3
##  6     6 Mustrum Ridcully      3
##  7     7 Fred Colon            2
##  8     8 Leonard of Quirm      2
##  9     9 Havelock Vetinari     2
## 10    10 Reg Shoe              2
```

Also: Jedes Argument von eben ist jetzt eine Spalte, jeder Wert in allen Vektoren jetzt ein Zeile.

5.3.5 Überblick über den Datensatz verschaffen

Zwar ist in unserem Fall das `tibble` ziemlich überschaubar, sollten wir uns aber trotzdem einen Überblick verschaffen wollen, können wir die `glimpse`-Funktion benutzen:

```
glimpse(my_1st_tibble)
```

```
## Rows: 10
## Columns: 3
## $ index <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
## $ name  <chr> "Agnes Nitt", "Samuel Vimes", "Esme...
## $ group <dbl> 1, 2, 1, 1, 3, 3, 2, 2, 2, 2
```

5.3.6 Spalten hinzufügen

Wir stellen uns jetzt vor, dass unser Datensatz die Anmeldeliste für ein Experiment darstellt.
Nach der Anmeldung haben die Probanden das Experiment durchgeführt und an zwei Testzeitpunkten in einem von uns designeten Aufmerksamkeitsparadigma die folgenden Punkte erhalten:

name	points_t1	points_t2
Agnes Nitt	4.8	2.8
Samuel Vimes	3.2	5.8
Esme Weatherwax	3.5	4.8
Gytha Ogg	3.1	5.3
Horace Worblehat	4.2	5.8
Mustrum Ridcully	4.7	5.0
Fred Colon	3.4	4.7
Leonard of Quirm	2.8	6.2
Havelock Vetinari	4.2	4.0
Reg Shoe	4.0	5.2

Diese Daten wollen wir jetzt unserem Datensatz hinzufügen, um sie für unsere Auswertungen verwenden zu können (das Beispiel ist ein bisschen künstlich, später werden wir einfach Datensätze aus externen Dateien einlesen. Um das Prinzip zu demonstrieren, ergibt es hier aber Sinn).

Um dies zu tun, benutzen wir die `mutate`-Funktion (auf deutsch 'ändern' oder 'mutieren').

```
mutate(my_1st_tibble,
       points_t1 = c(4.8, 3.2, 3.5, 3.1, 4.2, 4.7, 3.4, 2.8, 4.2, 4),
         points_t2 = c(2.8, 5.8, 4.8, 5.3, 5.8, 5, 4.7, 6.2, 4, 5.2))
```

```
## # A tibble: 10 x 5
##    index name              group points_t1 points_t2
##    <int> <chr>             <dbl>     <dbl>     <dbl>
##  1     1 Agnes Nitt            1       4.8       2.8
##  2     2 Samuel Vimes          2       3.2       5.8
##  3     3 Esme Weatherwax       1       3.5       4.8
##  4     4 Gytha Ogg             1       3.1       5.3
##  5     5 Horace Worblehat      3       4.2       5.8
##  6     6 Mustrum Ridcully      3       4.7       5
##  7     7 Fred Colon            2       3.4       4.7
##  8     8 Leonard of Quirm      2       2.8       6.2
##  9     9 Havelock Vetinari     2       4.2       4
## 10    10 Reg Shoe              2       4         5.2
```

`mutate` funktioniert wieder wie alle bisher gesehenen Funktionen, wenn wir das Objekt nicht überschreiben, wird der Output nicht gespeichert.

```
glimpse(my_1st_tibble)
```

```
## Rows: 10
## Columns: 3
## $ index <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
## $ name  <chr> "Agnes Nitt", "Samuel Vimes", "Esme...
## $ group <dbl> 1, 2, 1, 1, 3, 3, 2, 2, 2, 2
```

Wir könnten jetzt einfach den Datensatz wie gewohnt überschreiben, nutzen aber die Gelegenheit um ein weiteres wichtiges feature des `tidyverse` einzuführen.

`%>%` ist die *'pipe'*, ein Operator, der den Output des links genannten Ausdrucks als erstes Argument an den rechts genannten Ausdruck weitergibt.

Mit diesem Operator können wir das Beispiel von eben wie folgt umformulieren:

```
my_1st_tibble <- my_1st_tibble %>%
  mutate(points_t1 = c(4.8, 3.2, 3.5, 3.1, 4.2, 4.7, 3.4, 2.8, 4.2, 4),
         points_t2 = c(2.8, 5.8, 4.8, 5.3, 5.8, 5, 4.7, 6.2, 4, 5.2))
```

Nur noch schnell gucken ob das geklappt hat:

```
my_1st_tibble
```

```
## # A tibble: 10 x 5
##    index name              group points_t1 points_t2
##    <int> <chr>             <dbl>     <dbl>     <dbl>
##  1     1 Agnes Nitt            1       4.8       2.8
##  2     2 Samuel Vimes          2       3.2       5.8
##  3     3 Esme Weatherwax       1       3.5       4.8
##  4     4 Gytha Ogg             1       3.1       5.3
##  5     5 Horace Worblehat      3       4.2       5.8
##  6     6 Mustrum Ridcully      3       4.7       5
##  7     7 Fred Colon            2       3.4       4.7
##  8     8 Leonard of Quirm      2       2.8       6.2
##  9     9 Havelock Vetinari     2       4.2       4
## 10    10 Reg Shoe              2       4         5.2
```

Die `mutate`-Funktion wird richtig nützlich, wenn wir dem Datensatz Spalten hinzufügen wollen, die aus den anderen Spalten zusammengesetzt sind.

So könnten wir in unserem Beispiel überlegen, die Veränderung zum zweiten Messzeitpunkt als Spalte hinzufügen zu wollen. Dafür können wir die nötige arithmetische Operation einfach in der mutate-Funktion ausführen:

```
my_1st_tibble <- my_1st_tibble %>%
  mutate(change = points_t2 - points_t1)

my_1st_tibble
```

```
## # A tibble: 10 x 6
##    index name         group points_t1 points_t2 change
```

```
##    <int> <chr>        <dbl>     <dbl>     <dbl>  <dbl>
##  1     1 Agnes Nitt       1       4.8       2.8 -2
##  2     2 Samuel Vimes     2       3.2       5.8  2.60
##  3     3 Esme Weathe~     1       3.5       4.8  1.30
##  4     4 Gytha Ogg        1       3.1       5.3  2.20
##  5     5 Horace Worb~     3       4.2       5.8  1.60
##  6     6 Mustrum Rid~     3       4.7       5    0.300
##  7     7 Fred Colon       2       3.4       4.7  1.3
##  8     8 Leonard of ~     2       2.8       6.2  3.4
##  9     9 Havelock Ve~     2       4.2       4   -0.2
## 10    10 Reg Shoe         2       4         5.2  1.2
```

5.3.7 Spalten verändern

Es funktionieren natürlich nicht nur arithmetische Operatoren, wir können auch alle anderen vektorisierten Funktionen benutzen. Vektorisiert heißt ganz einfach gesagt, dass ein Vektor eingegeben wird und ein genauso langer Vektor ausgegeben wird[1].

In unserem Beispiel könnten wir überlegen, dass wir statt der Differenz die Wurzel aus der quadrierten Differenz als Änderung haben wollen. Da wir die Differenz schon zum Datensatz hinzugefügt haben, können wir sie einfach überschreiben.

```
my_1st_tibble <- my_1st_tibble %>%
  mutate(change = sqrt(change^2))

my_1st_tibble
```

```
## # A tibble: 10 x 6
##    index name         group points_t1 points_t2 change
##    <int> <chr>        <dbl>     <dbl>     <dbl>  <dbl>
##  1     1 Agnes Nitt       1       4.8       2.8  2
##  2     2 Samuel Vimes     2       3.2       5.8  2.60
##  3     3 Esme Weathe~     1       3.5       4.8  1.30
##  4     4 Gytha Ogg        1       3.1       5.3  2.20
##  5     5 Horace Worb~     3       4.2       5.8  1.60
##  6     6 Mustrum Rid~     3       4.7       5    0.300
##  7     7 Fred Colon       2       3.4       4.7  1.3
##  8     8 Leonard of ~     2       2.8       6.2  3.4
##  9     9 Havelock Ve~     2       4.2       4    0.2
## 10    10 Reg Shoe         2       4         5.2  1.2
```

5.4 Datensätze sortieren und indizieren

5.4.1 Datensätze sortieren

Der erste Schritt bei vielen Auswertungen ist es, sich die größten und kleinsten Werte einer Gruppe oder Variable anzugucken. Das kann schnell erreicht werden, indem man den gegebenen Datensatz anhand einer Varaible sortiert.

[1] Das ist nicht ganz richtig, aber für hier ausreichend nah genug an der richtigen Aussage.

Wir benutzen weiter unseren Datensatz von eben und wollen zuerst absteigend nach den Änderungen sortieren. Dafür können wir die `arrange`-Funktion und unseren pipe-Operator benutzen:

```
my_1st_tibble %>%
  arrange(change)
```

```
## # A tibble: 10 x 6
##    index name         group points_t1 points_t2 change
##    <int> <chr>        <dbl>     <dbl>     <dbl>  <dbl>
##  1     9 Havelock Ve~     2       4.2       4    0.2
##  2     6 Mustrum Rid~     3       4.7       5    0.300
##  3    10 Reg Shoe         2       4         5.2  1.2
##  4     3 Esme Weathe~     1       3.5       4.8  1.30
##  5     7 Fred Colon       2       3.4       4.7  1.3
##  6     5 Horace Worb~     3       4.2       5.8  1.60
##  7     1 Agnes Nitt       1       4.8       2.8  2
##  8     4 Gytha Ogg        1       3.1       5.3  2.20
##  9     2 Samuel Vimes     2       3.2       5.8  2.60
## 10     8 Leonard of ~     2       2.8       6.2  3.4
```

Wie man unschwer erkennen kann, ist hier der Standard, den Datensatz in aufsteigender Reihenfolge der Variable zu sortieren. Um das umzukehren, können wir die `desc`-Funktion nutzen, deren Name für 'descending', also absteigend steht:

```
my_1st_tibble %>%
  arrange(desc(change))
```

```
## # A tibble: 10 x 6
##    index name         group points_t1 points_t2 change
##    <int> <chr>        <dbl>     <dbl>     <dbl>  <dbl>
##  1     8 Leonard of ~     2       2.8       6.2  3.4
##  2     2 Samuel Vimes     2       3.2       5.8  2.60
##  3     4 Gytha Ogg        1       3.1       5.3  2.20
##  4     1 Agnes Nitt       1       4.8       2.8  2
##  5     5 Horace Worb~     3       4.2       5.8  1.60
##  6     7 Fred Colon       2       3.4       4.7  1.3
##  7     3 Esme Weathe~     1       3.5       4.8  1.30
##  8    10 Reg Shoe         2       4         5.2  1.2
##  9     6 Mustrum Rid~     3       4.7       5    0.300
## 10     9 Havelock Ve~     2       4.2       4    0.2
```

5.4.2 mehrere Sortierschlüssel

Wenn wir der `arrange`-Funktion mehr als eine Variable übergeben, werden die zusätzlichen Variablen als zusätzliche Sortierschlüssel interpretiert.

Wir könnten zum Beispiel aufsteigend nach der Gruppe und dann pro Gruppe absteigend nach der Veränderung sortieren wollen. Das könnte dann so aussehen:

```
my_1st_tibble %>%
  arrange(group, desc(change))
```

```
## # A tibble: 10 x 6
##    index name         group points_t1 points_t2 change
##    <int> <chr>        <dbl>     <dbl>     <dbl>  <dbl>
##  1     4 Gytha Ogg        1       3.1       5.3  2.20
##  2     1 Agnes Nitt       1       4.8       2.8  2
##  3     3 Esme Weathe~     1       3.5       4.8  1.30
##  4     8 Leonard of ~     2       2.8       6.2  3.4
##  5     2 Samuel Vimes     2       3.2       5.8  2.60
##  6     7 Fred Colon       2       3.4       4.7  1.3
##  7    10 Reg Shoe         2       4         5.2  1.2
##  8     9 Havelock Ve~     2       4.2       4    0.2
##  9     5 Horace Worb~     3       4.2       5.8  1.60
## 10     6 Mustrum Rid~     3       4.7       5    0.300
```

5.4.3 Auswahl von Spalten

Es passiert öfter, dass wir eine oder mehrere Variablen nicht mehr benötigen und der Übersichtlichkeit halber aus dem Datensatz entfernen wollen.

Das `tidyverse` bietet dafür mit der `select`-Funktion eine sehr intuitiv zugängliche Syntax an, die wir zur Auswahl und zum Ausschluss von Spalten nutzen können. Dazu pipen wir einfach wieder den Datensatz in die Funktion und listen diejenigen Variablen auf, die wir behalten können.

In unserem Beispiel könnten wir uns überlegen, den Index und die beiden Punkt-Spalten nicht mehr zu benötigen. Wir geben also einfach den Dantensatz in die `select`-Funktion und listen die restlichen Variablen auf:

```
my_1st_tibble %>%
  select(name, group, change)
```

```
## # A tibble: 10 x 3
##    name              group change
##    <chr>             <dbl>  <dbl>
##  1 Agnes Nitt            1  2
##  2 Samuel Vimes          2  2.60
##  3 Esme Weatherwax       1  1.30
##  4 Gytha Ogg             1  2.20
##  5 Horace Worblehat      3  1.60
##  6 Mustrum Ridcully      3  0.300
##  7 Fred Colon            2  1.3
##  8 Leonard of Quirm      2  3.4
##  9 Havelock Vetinari     2  0.2
## 10 Reg Shoe              2  1.2
```

Hier funktioniert auch die schon bei der numerischen Vektor-Indizierung vorgestellte Methode zum Ausschluss von Werten. Durch `-`-eingeleitete Variablennamen lassen sich einfach ausschließen:

```
my_1st_tibble %>%
  select(-index, -points_t1, -points_t2)
```

```
## # A tibble: 10 x 3
##    name              group change
##    <chr>             <dbl>  <dbl>
##  1 Agnes Nitt            1  2
##  2 Samuel Vimes          2  2.60
##  3 Esme Weatherwax       1  1.30
##  4 Gytha Ogg             1  2.20
##  5 Horace Worblehat      3  1.60
##  6 Mustrum Ridcully      3  0.300
##  7 Fred Colon            2  1.3
##  8 Leonard of Quirm      2  3.4
##  9 Havelock Vetinari     2  0.2
## 10 Reg Shoe              2  1.2
```

Wir können die Spalten-Auswahl auch benutzen, um den Datensatz für folgende Funktionen vorzubereiten. Ein Beispiel ist die schon aus der letzen Sitzung bekannte Funktion `cor`, die neben Vektoren auch einen Datensatz mit numerischen Spalten als Input versteht.

Wir könnten uns zum Beispiel die Frage stellen, ob die Aufmerksamkeitswerte vor und nach dem Training miteinander korreliert sind.

```
my_1st_tibble %>%
  select(points_t1, points_t2) %>%
  cor()
```

```
##            points_t1  points_t2
## points_t1  1.0000000 -0.6449612
## points_t2 -0.6449612  1.0000000
```

Das `tidyverse` bietet wie gesagt viele features, die unser Leben als empirische Forscher leichter machen, darunter eine Reihe von `selection helpers`.

Der erste, den wir hier verwenden wollen ist die `contains`-Funktion. Eine Funktion in der wir Teile von Spaltennamen suchen können, bei uns zum Beispiel `'points'`, um den Aufruf von eben ein bisschen zu vereinfachen:

```
my_1st_tibble %>%
  select(contains('points')) %>%
  cor()
```

```
##            points_t1  points_t2
## points_t1  1.0000000 -0.6449612
## points_t2 -0.6449612  1.0000000
```

Der zweite helper, den wir ausprobieren ist `where`, ein helper, der uns erlaubt, mit Hilfe einer Funktion Spalten auszuwählen, auf die eine Bedingung zutrifft.

In unserem Beispiel wollen wir alle numerischen Variablen miteinander korrelieren, indem wir `where` mit `is.numeric` kombinieren:

```
my_1st_tibble %>%
  select(where(is.numeric)) %>%
  cor()
```

```
##                index      group   points_t1
## index      1.00000000  0.4227600 -0.06127845
## group      0.42276002  1.0000000  0.28205162
## points_t1 -0.06127845  0.2820516  1.00000000
## points_t2  0.24860383  0.4354397 -0.64496117
## change    -0.32568808 -0.3039316 -0.65762107
##            points_t2     change
## index      0.2486038 -0.3256881
## group      0.4354397 -0.3039316
## points_t1 -0.6449612 -0.6576211
## points_t2  1.0000000  0.4254564
## change     0.4254564  1.0000000
```

5.4.4 logische Auswahl von Zeilen

Um Zeilen aus einem Datensatz auszuwählen gibt es sowohl die Möglichkeit über `filter` mit logischen Angaben Zielen auszuwählen, als auch mit `slice` eine numerische Auswahl zu treffen.

Logische Indizierung kann praktisch sein, wenn man die Einträge sehen möchte, die zum Beispiel unterdurchschnittlich viel Veränderung gezeigt haben. Das könnte so aussehen:

```
my_1st_tibble %>%
  filter(change < mean(change))
```

```
## # A tibble: 6 x 6
##   index name          group points_t1 points_t2 change
##   <int> <chr>         <dbl>     <dbl>     <dbl>  <dbl>
## 1     3 Esme Weather~     1       3.5       4.8  1.30
## 2     5 Horace Worbl~     3       4.2       5.8  1.60
## 3     6 Mustrum Ridc~     3       4.7       5    0.300
## 4     7 Fred Colon        2       3.4       4.7  1.3
## 5     9 Havelock Vet~     2       4.2       4    0.2
## 6    10 Reg Shoe          2       4         5.2  1.2
```

5.4.5 numerische Auswahl von Zeilen

Zusätzlich zur logischen Indizierung bietet das `tidyverse` mit der `slice`-Funktion auch die Möglichkeit numerisch zu indizieren.

Das könnte in unserem Beispiel exemplarisch hilfreich sein, um die drei Versuchspersonen ausgeben zu lassen, die die größte Veränderung gezeigt haben.

```
my_1st_tibble %>%
  arrange(desc(change)) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 6
##   index name          group points_t1 points_t2 change
##   <int> <chr>         <dbl>     <dbl>     <dbl>  <dbl>
## 1     8 Leonard of Q~     2       2.8       6.2   3.4
```

```
## 2     2 Samuel Vimes      2       3.2       5.8   2.60
## 3     4 Gytha Ogg         1       3.1       5.3   2.20
```

5.5 Vorlesung

5.5.1 Skript aus der Vorlesung:

```
library(tidyverse)

a <- data.frame(x = 1:100,
                y = 201:300)

b <- tibble(x = 1:100,
            y = 201:300)

tibble(x = 1,
       y = 1:3,
       z = '10')
```

```
## # A tibble: 3 x 3
##       x     y z
##   <dbl> <int> <chr>
## 1     1     1 10
## 2     1     2 10
## 3     1     3 10
```

```
tibble(iris)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width
##           <dbl>       <dbl>        <dbl>       <dbl>
##  1          5.1         3.5          1.4         0.2
##  2          4.9         3            1.4         0.2
##  3          4.7         3.2          1.3         0.2
##  4          4.6         3.1          1.5         0.2
##  5          5           3.6          1.4         0.2
##  6          5.4         3.9          1.7         0.4
##  7          4.6         3.4          1.4         0.3
##  8          5           3.4          1.5         0.2
##  9          4.4         2.9          1.4         0.2
## 10          4.9         3.1          1.5         0.1
## # ... with 140 more rows, and 1 more variable:
## #   Species <fct>
```

```
glimpse(iris)
```

```
## Rows: 150
## Columns: 5
## $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4...
## $ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9...
## $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7...
```

```
## $ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4...
## $ Species      <fct> setosa, setosa, setosa, seto...
```

```
a <- iris

a <- mutate(a, new_column = 1)


df <- tibble(id = sample(letters, 20, replace = TRUE),
             spr_iq = sample(80:115, 20, T),
             log_iq = sample(80:115, 20, T),
             smokes = sample(c(T,F), 20, T))

df %>%
  mutate(ges_iq = (spr_iq + log_iq) / 2) %>%
  arrange(ges_iq) %>%
  filter(ges_iq > 100) %>%
  select(contains('iq')) %>%
  cor()
```

```
##            spr_iq     log_iq    ges_iq
## spr_iq  1.0000000 -0.5354699 0.4656018
## log_iq -0.5354699  1.0000000 0.4981101
## ges_iq  0.4656018  0.4981101 1.0000000
```

Als ergänzende Nebenbemerkung:

mit `skim` aus dem `skimr`-Paket lässt sich der Überblick noch schöner gestalten:

```
skimr::skim(iris)
```

Chapter 6

Faktoren und Aggregation

6.1 Organisatorisches

6.1.1 Semesterplan

Einheit	Vorlesung	Übungswoche	Thema
1	2.11.20	keine Übung	Grundlagen und Begriffe
2	16.11.20	KW 48	Vektoren und Indizierung
			Datenformate erstellen und transformieren
3	30.11.20	KW 50	Pakete installieren und benutzen
			Datensätze erstellen und ergänzen können
			Datensätze sortieren und indizieren können
4	14.12.20	KW 1	Faktoren
			deskriptive Kennwerte
			Aggregation I
5	11.01.21	KW 3	Aggregation II
			In- und Export von Datensätzen
6	25.01.21	KW 5	Grafische Darstellungen I
7	08.02.21	KW 7	Grafische Darstellungen II
8	22.02.21	keine Übung	Puffer
			Probeklausur

6.2 Faktoren

6.2.1 Gruppierungsfaktoren

Mit der Klasse `factor` können die Eigenschaften kategorialer Variablen abgebildet werden. Sie wird insbesondere für Gruppierungsfaktoren im versuchsplanerischen Sinn verwendet und kann bei statistischen Auswertung und Darstellungen hilfreich sein.

Eine Möglichkeit, ein Objekt der Klasse `factor` zu erstellen, ist die `factor()`-Funktion.

Die Stufen eines ungeordneten Faktors haben keine hierarchische Ordnung - Beispiel "Geschlecht":

```
sex <- c("m", "f", "f", "m", "m", "m", "f", "f")
class(sex)
```

```
## [1] "character"
```

```
sexFac  <-  factor(sex)
class(sexFac)
```

```
## [1] "factor"
```

```
sexFac
```

```
## [1] m f f m m m f f
## Levels: f m
```

Ebenso funktioniert dies mit numerischen Faktorstufen.

```
factor(c(1, 1, 3, 3, 4, 4))
```

```
## [1] 1 1 3 3 4 4
## Levels: 1 3 4
```

`factor()` bietet außerdem die Möglichkeit, nicht im Ursprungsvektor definierte Faktorstufen zu definieren:

```
numericfactor <- factor(c(1, 1, 3, 3, 4, 4), levels=1:5)
numericfactor
```

```
## [1] 1 1 3 3 4 4
## Levels: 1 2 3 4 5
```

6.2.2 Mit Faktorstufen arbeiten

```
## 0=Mann, 1=Frau
(sexNum <- sample(0:1, 30,T))
```

```
##  [1] 1 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 0 1 0 1 0
## [26] 0 0 1 0 1
```

```
sexFac <- factor(sexNum, labels=c("male", "female"))
sexFac
```

```
##  [1] female male   female male   female male   male
##  [8] male   male   male   male   female female female
```

```
## [15] male   female female female female female male
## [22] female male   female male   male   male   female
## [29] male   female
## Levels: male female
```

```
## ausgeben, wie häufig welche Stufe vorkommt
summary(sexFac)
```

```
##   male female
##     15     15
```

```
## die Struktur des Faktors ausgeben
str(sexFac)
```

```
##  Factor w/ 2 levels "male","female": 2 1 2 1 2 1 1 1 1 1 ...
```

Achtung: Die Struktur des Faktors ist immer numerisch von 1 aufsteigend (auch wenn es ursprünglich "0" und "1" waren, s.o.).

6.2.3 ändern von Faktorstufen

Das `tidyverse` bietet einen netten Wrapper um Faktoren umzuwandeln.

Zuerst wandeln wir die Stufen unseres `sexFac` in andere labels um:

```
library(tidyverse)

recode_factor(sexFac,
              male = 'männlich',
              female = 'weiblich')
```

```
##  [1] weiblich männlich weiblich männlich weiblich
##  [6] männlich männlich männlich männlich männlich
## [11] männlich weiblich weiblich weiblich männlich
## [16] weiblich weiblich weiblich weiblich weiblich
## [21] männlich weiblich männlich weiblich männlich
## [26] männlich männlich weiblich männlich weiblich
## Levels: männlich weiblich
```

Was aber im Zweifel noch praktischer sein kann, ist unnötige Stufen zusammenzufassen:

```
numericfactor
```

```
## [1] 1 1 3 3 4 4
## Levels: 1 2 3 4 5
```

```
recode_factor(numericfactor,
              '1' = 'one',
              '2' = 'two',
              '3' = 'three',
              .default = 'rest')
```

```
## [1] one   one   three three rest  rest
## Levels: one two three rest
```

6.2.4 Quantitative in kategoriale Variablen umwandeln

Für Median-Splits und ähnliches ist es sehr praktisch, direkt numerische Variablen in nach Grenzen eingeteilte Faktoren umzuwandeln. Dabei ist Variante 1 die schon bekannte `ifelse`-Funktion:

```
a_numeric_variable <- sample(1:100, 100, T)


factor(ifelse(a_numeric_variable > median(a_numeric_variable),
              'high',
              'low'))
```

```
##   [1] low  high low  high low  low  low  low  low
##  [10] high low  high low  high low  low  low  low
##  [19] low  high high high low  high high high low
##  [28] high high high low  low  low  high low  low
##  [37] low  low  low  low  high high high low  high
##  [46] high low  low  high low  high high high low
##  [55] high high high high high low  high high high
##  [64] high low  high high low  high high low  high
##  [73] high low  low  low  high low  low  low  high
##  [82] low  high low  low  high low  high low  high
##  [91] high high high low  low  high low  low  high
## [100] low
## Levels: high low
```

Wenn mehr als 2 Gruppen gewünscht sind, hilft die `cut`-Funktion IQ-Werte in 3 Klassen einteilen:

```
IQ <- sample(80:120, 100, T)
## Intervalle: [0;85], (85;115], (115;inf]
IQfac <- cut(IQ, breaks=c(0, 85, 115, Inf),
             labels=c("lo", "mid", "hi"))
summary(IQfac)
```

```
##  lo mid  hi
##  19  65  16
```

`cut` kann man außerdem ganz einfach zusammen mit der `quantile`-Funktion benutzen um beliebige Perzentil-Splits durchzuführen:

```
quantSplit <- cut(IQ,
                  breaks=c(-Inf,
                           quantile(IQ,
                                    probs=c(0.25,
                                            0.5,
                                            0.75)),
                           Inf))
summary(quantSplit)
```

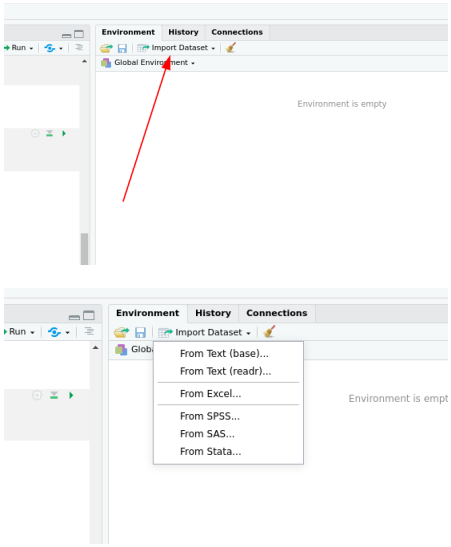
```
## (-Inf,90.8] (90.8,99.5]  (99.5,112]  (112, Inf]
##          25          25          25          25
```

6.3 Daten einlesen I

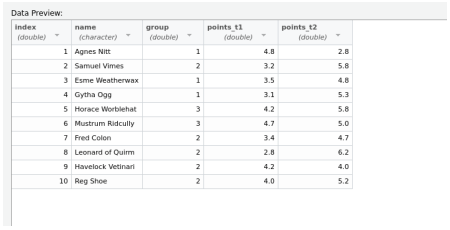
6.3.1 `csv`

Die nächste Veranstaltung wird sich nochmal umfassen mit dem Einlesen und Abspeichern von Datensätzen beschäftigen, um ein bisschen flexibler bei den Übungsaufgaben zu sein, führen wir hier aber schon mal eine Funktion aus dem `readr`-Paket (natürlich Teil des `tidyverse`) ein; die `read_csv`-Funktion. `.csv`-files sind eine weit verbreitete Art und Weise, tabellarische Daten abzuspeichern.

Der einfachste Weg, solche files einzulesen, führt über die RStudio-GUI:



Hier im Beispiel hab ich jetzt die Datei `test.csv` ausgewählt, die ein Beispiel aus der letzten Vorlesung enthält:


Data Preview:

index (double)	name (character)	group (double)	points_t1 (double)	points_t2 (double)
1	Agnes Nitt	1	4.8	2.8
2	Samuel Vimes	2	3.2	5.8
3	Esme Weatherwax	1	3.5	4.8
4	Gytha Ogg	1	3.1	5.3
5	Horace Worblehat	3	4.2	5.8
6	Mustrum Ridcully	3	4.7	5.0
7	Fred Colon	2	3.4	4.7
8	Leonard of Quirm	2	2.8	6.2
9	Havelock Vetinari	2	4.2	4.0
10	Reg Shoe	2	4.0	5.2

Was mir den den folgenden Code-Schnipsel liefert, den ich dann in mein Skript kopieren kann:


Code Preview:

```
library(readr)
test <- read_csv("data/test.csv")
View(test)
```

```
test <- read_csv("data/test.csv")
```

Für's Erste soll uns das an Einlese-Strategien reichen.

6.4 deskriptive Kennwerte

6.4.1 Einfache deskriptiv-statistische Kennwerte

Wir hatten in der zweiten Veranstaltung ja schon ein paar deskriptive Kennwerte, die wollen wir jetzt auf Datensätze anwenden.

Wir könnten uns beispielsweise fragen, was der Mittelwert der beiden Punkte pro Testzeitpunkt war.

Dafür können wir die `pull`-Funktion nutzen, um uns einfache Spalten des Datensatzes als Vektor ausgeben zu lassen:

```
test %>%
  pull(points_t1) %>%
  mean()

test %>%
  pull(points_t2) %>%
  mean()
```

```
## [1] 3.79
## [1] 4.96
```

Das funktioniert zwar, wird aber umständlicher, je mehr Spalten und Kennwerte wir berechnen wollen.

Natürlich gibt es `tidyverse` auch dafür Funktionen, die uns die Arbeit leichter machen. Mit der `summarise`-Funktion können wir ähnlich wie mit der `mutate`-Funktion Variablen definieren, die dann aber Zusammenfassungen über angegebene Funktionen sind:

```
test %>%
  summarise(m_t1 = mean(points_t1),
            sd_t1 = sd(points_t1),
            m_t2 = mean(points_t2),
            sd_t2 = sd(points_t2),)
```

```
## # A tibble: 1 x 4
##    m_t1 sd_t1  m_t2 sd_t2
##   <dbl> <dbl> <dbl> <dbl>
## 1  3.79 0.689  4.96 0.989
```

Das ist ja schon ganz nett, aber diese Infos sind selten hilfreich. In unserem Datensatz sind Experimentalgruppen eingeteilt, eigentlich wollen wir unsere Mittelwerte pro Gruppe ausrechnen. Mit der `group_by`-Funktion ist das auch ganz einfach möglich.

In unsere `pipe` von eben bauen wir dazu einfach einen kurzen `group_by`-Aufruf ein und schon sind wir beim erwünschten Ergebnis:

```
test %>%
  group_by(group) %>%
  summarise(m_t1 = mean(points_t1),
            sd_t1 = sd(points_t1),
            m_t2 = mean(points_t2),
            sd_t2 = sd(points_t2))
```

```
## # A tibble: 3 x 5
##   group  m_t1 sd_t1  m_t2 sd_t2
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1  3.8  0.889  4.3  1.32
## 2     2  3.52 0.576  5.18 0.873
## 3     3  4.45 0.354  5.4  0.566
```

Und diese Gruppierungen können wir jetzt einfach mit dem vorhin gesehenen `cut` zur Gruppierung kombinieren.

So könnten wir uns zum Beispiel fragen, wie die Mittelwerte und Streuungen in den Quartilen aussehen. Dafür können wir einfach ein `mutate` vorschalten, indem wir die Verbesserung zum zweiten Termin und einen Quartilsplit einführen, den wir dann direkt zum Gruppieren benutzen:

```
test <- test %>%
  mutate(improvement = points_t2 - points_t1,
         quart_split = cut(improvement,
                           breaks = c(-Inf,
                                      quantile(improvement,
                                               probs = c(.25,.5,.75)),
                                      Inf),
                           right = T,
                           labels = c('q1', 'q2', 'q3', 'q4')
                           )
         )

test %>%
  group_by(quart_split) %>%
  summarise(m_t1 = mean(points_t1),
            sd_t1 = sd(points_t1),
            m_t2 = mean(points_t2),
            sd_t2 = sd(points_t2))
```

```
## # A tibble: 4 x 5
##   quart_split  m_t1 sd_t1  m_t2 sd_t2
##   <fct>       <dbl> <dbl> <dbl> <dbl>
## 1 q1           4.57 0.321  3.93 1.10
## 2 q2           3.75 0.354  5    0.283
## 3 q3           3.8  0.566  5.25 0.778
## 4 q4           3.03 0.208  5.77 0.451
```

6.4.2 Häufigkeitsauszählungen

Diese `pipe` können wir auch verwenden, um uns Häufigkeiten von Bedingungskombinationen anzugucken. Dafür tauschen wir einfach die `summarise`- durch die `count`-Funktion aus und schon ist der Output eine Tabelle mit den absoluten Häufigkeiten:

```
test %>%
  group_by(quart_split) %>%
  count()
```

```
## # A tibble: 4 x 2
## # Groups:   quart_split [4]
##   quart_split     n
##   <fct>       <int>
## 1 q1              3
## 2 q2              2
## 3 q3              2
## 4 q4              3
```

Die `group_by`-Funktion kann dabei auch mehrere Argumente verstehen, wir können also auch nach Gruppe und Quantilen auszählen:

```
test %>%
  group_by(quart_split,group) %>%
  count()
```

```
## # A tibble: 9 x 3
## # Groups:   quart_split, group [9]
##   quart_split group     n
##   <fct>       <dbl> <int>
## 1 q1              1     1
## 2 q1              2     1
## 3 q1              3     1
## 4 q2              1     1
## 5 q2              2     1
## 6 q3              2     1
## 7 q3              3     1
## 8 q4              1     1
## 9 q4              2     2
```

Chapter 7

Aggregation und Filemanagement

7.1 Organisatorisches

7.1.1 Semesterplan

Einheit	Vorlesung	Übungswoche	Thema
1	2.11.20	keine Übung	Grundlagen und Begriffe
2	16.11.20	KW 48	Vektoren und Indizierung
			Datenformate erstellen und transform
3	30.11.20	KW 50	Pakete installieren und benutzen
			Datensätze erstellen und ergänzen kö
			Datensätze sortieren und indizieren k
4	14.12.20	KW 1	Faktoren
			deskriptive Kennwerte
			Aggregation I
5	11.01.21	KW 3	Aggregation II
			In- und Export von Datensätzen
6	25.01.21	KW 5	Grafische Darstellungen I
7	08.02.21	KW 7	Grafische Darstellungen II
8	22.02.21	keine Übung	Puffer
			Probeklausur

7.2 Aggregation

7.2.1 Aggregation über mehrere Spalten

Beispiel aus der letzten Sitzung:

```
test <- read_csv("data/test.csv")
test %>%
  group_by(group) %>%
  summarise(m_t1 = mean(points_t1),
            sd_t1 = sd(points_t1),
            m_t2 = mean(points_t2),
            sd_t2 = sd(points_t2))
```

```
## # A tibble: 3 x 5
##   group  m_t1 sd_t1  m_t2 sd_t2
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1  3.8  0.889  4.3  1.32
## 2     2  3.52 0.576  5.18 0.873
## 3     3  4.45 0.354  5.4  0.566
```

7.2.2 across-Funktion

Statt die Spalten auf denen wir Operationen durchführen wollen alle einzeln auszuwählen, können wir mit der `across`-Funktion auch eine Stapelweise Operation anstoßen.

Dazu benutzen wir die `select`-helper, die wir schon kennen:

```
test %>%
  select(contains('points'))
```

```
## # A tibble: 10 x 2
##    points_t1 points_t2
##        <dbl>     <dbl>
##  1       4.8       2.8
##  2       3.2       5.8
##  3       3.5       4.8
##  4       3.1       5.3
##  5       4.2       5.8
##  6       4.7       5
##  7       3.4       4.7
##  8       2.8       6.2
##  9       4.2       4
## 10       4         5.2
```

Diese praktischen Tools können wir jetzt mit der `across`-Funktion in unsere `summarise`-pipeline integrieren:

```
test %>%
  group_by(group) %>%
  summarise(across(contains('points'),
                   .fns = mean,
```

```
                .names = 'm_{.col}'),
          across(contains('points'),
                .fns = sd,
                .names = 'sd_{.col}'))
```

```
## # A tibble: 3 x 7
##   group m_points_t1 m_points_t2 sd_points_t1
##   <dbl>       <dbl>       <dbl>        <dbl>
## 1     1        3.8         4.3         0.889
## 2     2        3.52        5.18        0.576
## 3     3        4.45        5.4         0.354
##   sd_points_t2 sd_m_points_t1 sd_m_points_t2
##          <dbl>          <dbl>          <dbl>
## 1        1.32              NA             NA
## 2        0.873             NA             NA
## 3        0.566             NA             NA
```

7.2.3 Frage

```
## # A tibble: 3 x 7
##   group m_points_t1 m_points_t2 sd_points_t1
##   <dbl>       <dbl>       <dbl>        <dbl>
## 1     1        3.8         4.3         0.889
## 2     2        3.52        5.18        0.576
## 3     3        4.45        5.4         0.354
##   sd_points_t2 sd_m_points_t1 sd_m_points_t2
##          <dbl>          <dbl>          <dbl>
## 1        1.32              NA             NA
## 2        0.873             NA             NA
## 3        0.566             NA             NA
```

Warum sind da zwei leere Spalten?

1. Weil wir was falsch gemacht haben.
2. Weil auch die Streuungen der Mittelwerte berechnet wurde, also von jeweils einem Wert, was nicht geht.
3. Das ist das Ergebnis der Standardisierung der Streuungen am N, da das N hier 0 ist, wird durch 0 geteilt und es kommt nix raus.

7.2.4 lazy evaluation

Die `tidyverse`-Funktionen folgen dem Prinzip der 'lazy evaluation', das heißt ein Argument wird nach dem anderen ausgeführt.

Wenn wir also im ersten Argument Spalten erstellen, die zusätzlich zum ursprünglichen Datensatz von `contains` betroffen werden, werden die folgenden Funktionen auch auf diese angewandt. Um das zu verhindern, können wir das `.fn`-Argument erweitern und im `.names`- Argument einen weiteren Platzhalter verwenden:

```
test %>%
  group_by(group) %>%
```

```
  summarise(across(contains('points'),
                   .fns = list(mean = mean,
                               sd = sd),
                   .names = '{.fn}_{.col}'))
```

```
## # A tibble: 3 x 5
##   group mean_points_t1 sd_points_t1 mean_points_t2
##   <dbl>          <dbl>        <dbl>          <dbl>
## 1     1           3.8         0.889           4.3
## 2     2           3.52        0.576           5.18
## 3     3           4.45        0.354           5.4
##   sd_points_t2
##          <dbl>
## 1        1.32
## 2        0.873
## 3        0.566
```

Die lazy evaluation ist ein feature, das wir zum Beispiel dazu benutzen können, Zwischenergebnisse zu benutzen. Da die `summarise`-Argumente Schritt für Schritt ausgeführt werden, funktioniert der folgende Code:

```
test %>%
  group_by(group) %>%
  summarise(n = n(),
            sd = sd(points_t1),
            sem = sd / sqrt(n))
```

```
## # A tibble: 3 x 4
##   group     n    sd   sem
##   <dbl> <int> <dbl> <dbl>
## 1     1     3 0.889 0.513
## 2     2     5 0.576 0.258
## 3     3     2 0.354 0.25
```

7.2.5 Mit aggregierten Daten weiterarbeiten

Um die Ergebnisse einer nach einer `group`ierten `summarise`-Funktion für weitere Manipulationen zu verwenden, müssen wir noch ein `ungroup` in die pipe einfügen, um R mitzuteilen dass die folgenden Operationen wieder Datensatz-übergreifend gedacht sind. So können wir zum Beispiel aus absoluten relative Häufigkeiten errechnen.

```
test %>%
  group_by(group) %>%
  count() %>%
  ungroup() %>%
  mutate(rel_n = n/sum(n))
```

```
## # A tibble: 3 x 3
##   group     n rel_n
##   <dbl> <int> <dbl>
## 1     1     3   0.3
```

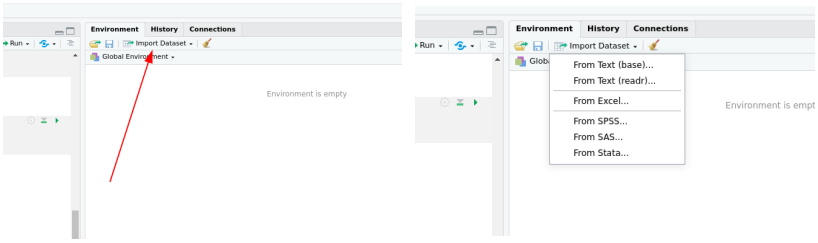
```
## 2     2     5   0.5
## 3     3     2   0.2
```

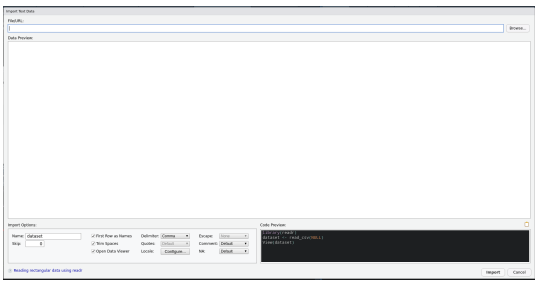
7.3 Daten einlesen II und zusammenfügen

7.3.1 Daten aus Textfiles

Daten aus Textfiles lassen sich am Einfachsten (wie letzte Sitzung auch schon kurz gezeigt) über die RStudio-IDE einlesen.

Dazu einfach auf das 'Import Dataset'- Menü über dem Environment klicken und 'From Text (readr)...' auswählen:





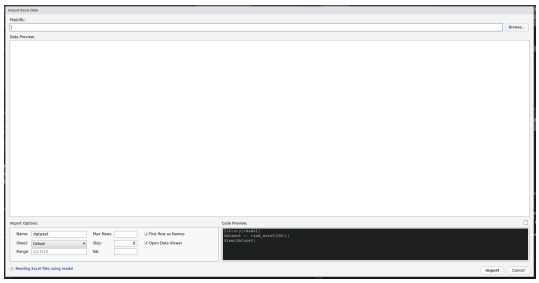
Der mit der grafischen Oberfläche erstellte Code lässt sich dann einfach in's R-Skript kopieren:

```
test <- read_csv("data/test.csv")

test2 <- read_delim("data/test2.txt",
                    "\t",
                    escape_double = FALSE,
                    trim_ws = TRUE)

stadtteile <- read_delim("data/test3.csv",
                    ";",
                    escape_double = FALSE,
                    locale = locale(decimal_mark = ",",
                                    encoding = "ISO-8859-2",
                                    asciify = TRUE),
                    trim_ws = TRUE)
glimpse(stadtteile)
```

```
## Rows: 30
## Columns: 6
## $ Land      <chr> "de-sh", "de-sh", "de-sh", "de-...
## $ Stadt     <chr> "Kiel", "Kiel", "Kiel", "Kiel",...
## $ Kategorie <chr> "geo", "geo", "geo", "geo", "ge...
## $ Merkmal   <chr> "Flächen in Hektar", "Flächen i...
## $ Stadtteil <chr> "Altstadt", "Vorstadt", "Exerzi...
## $ Hektar    <dbl> 35.0983, 45.8515, 42.0120, 45.1...
```

7.3.2 Daten aus Excel-Dateien

Daten aus Excel-Dateien lassen sich ganz ähnlich einlesen, nur dass wir statt 'From Text (readr)...' nun 'From Excel...' auswählen.



```
library(readxl)
kiel_haushalte <- read_excel("data/kiel.xlsx", sheet = "Haushalte")

glimpse(kiel_haushalte)
```

```
## Rows: 30
## Columns: 7
## $ Stadtteile                     <chr> "Altstadt"...
## $ Einpersonen                    <dbl> 384, 657, ...
## $ `Paar ohne Kind`               <dbl> 305, 303, ...
## $ `Paar mit Kindern`             <dbl> 165, 167, ...
## $ `Paar mit Nachkommen`          <dbl> 29, 13, 56...
## $ Alleinerziehende               <dbl> 28, 34, 21...
## $ `Sonst. Mehrpersonenhaushalte` <dbl> 118, 110, ...
```

7.3.3 Kombinieren von Datensätzen

Das `tidyverse` bietet eine sehr praktische Familie von Funktionen, um Datensätze zusammenzufügen, die `join`-Funktionen.

Wir werden hier erstmal nur eine davon benutzen, die `left_join`-Funktion. Sie nimmt zwei Datensätze als Argumente, gerne auch den ersten als Ergebnis einer pipeline, und fügt diese anhand einer im `by`-Argument angegebenen Spalte zusammen.

Wir wollen mal die Datensätze zu den Haushalten und der Fläche der Kieler Stadtteile zusammenfügen:

```
df <- stadtteile %>%
  left_join(kiel_haushalte,by = c('Stadtteil' = 'Stadtteile'))

glimpse(df)
```

```
## Rows: 30
## Columns: 12
## $ Land                              <chr> "de-sh", "...
## $ Stadt                             <chr> "Kiel", "K...
## $ Kategorie                         <chr> "geo", "ge...
## $ Merkmal                           <chr> "Flächen i...
## $ Stadtteil                         <chr> "Altstadt"...
## $ Hektar                            <dbl> 35.0983, 4...
## $ Einpersonen                       <dbl> 384, 657, ...
## $ `Paar ohne Kind`                  <dbl> 305, 303, ...
## $ `Paar mit Kindern`                <dbl> 165, 167, ...
## $ `Paar mit Nachkommen`             <dbl> 29, 13, 56...
## $ Alleinerziehende                  <dbl> 28, 34, 21...
## $ `Sonst. Mehrpersonenhaushalte`    <dbl> 118, 110, ...
```

Das sieht ja sehr nett aus, wir können den zusammengefügten Datensatz nun benutzen, um eine Korrelationsmatrix für alle numerischen Spalten zu erstellen:

```
df %>%
  select(where(is.numeric)) %>%
  cor()
```

	Hektar	Einpersonen	Paar ohne Kind	Paar mit Kindern	Paar mit Na
Hektar	1.000	-0.029	0.364	0.314	
Einpersonen	-0.029	1.000	0.766	0.565	
Paar ohne Kind	0.364	0.766	1.000	0.822	
Paar mit Kindern	0.314	0.565	0.822	1.000	
Paar mit Nachkommen	0.445	0.426	0.814	0.950	
Alleinerziehende	0.199	0.617	0.739	0.932	
Sonst. Mehrpersonenhaushalte	0.204	0.750	0.853	0.933	

7.3.4 Exportieren von Datensätzen in csv-Format

Um diesen Datensatz jetzt wieder zu exportieren, können wir uns eins von den schon beim Einlesen kennengelernten Formaten aussuchen. Hier können wir dann aber leider nicht so einfach die GUI benutzen, wenn wir nicht ein `.RData`-File erstellen wollen (was wir dieses Semester nicht tun).

Um in `csv`-Dateien zu exportieren, können wir einfach die `write_csv`-Funktion aus dem `tidyverse` verwenden. Sie nimmt wieder einen Datensatz als erstes Argument (pipeline!) und dazu einen Pfad, in den wir unser Ergebnis abspeichern wollen. Wir versuchen mal unsere Korrelationstabelle von eben abzuspeichern. Dabei müssen wir aber darauf achten, dass wir aus der Tabelle ein `tibble` und am Besten noch die Zeilennamen zu einer Spalte machen, damit diese erhalten bleiben:

```
correlations <-
  df %>%
  select(where(is.numeric)) %>%
  cor() %>%
  as_tibble(rownames = 'Masseinheit')

correlations %>%
  write_csv('data/correlations.csv')
```

7.3.5 Exportieren von Datensätzen in Excel-Format

Manchmal ist es aber praktischer (vor allem wenn man einen Bericht in MS Office erstellt), die Ergebnisse als `.xlsx` zu exportieren.

Das ist leider nicht mit dem `readxl`-Paket möglich. Dafür benötigen wir das `openxlsx`-Paket, das natürlich zuerst mit `install.packages` installiert werden muss.

Mit den Funktionen `createWorkbook` und `addWorksheet` können wir ein Excel-Dokument und Tabellen in diesem anlegen, die wir dann mit `writeData` füllen und mit `saveWorkbook` abspeichern können.

```
library(openxlsx)

excel_output <- createWorkbook()

addWorksheet(excel_output, 'correlations')
writeData(excel_output, 'correlations', correlations)

addWorksheet(excel_output, 'original_data')
writeData(excel_output, 'original_data', df)

saveWorkbook(excel_output,'data/output.xlsx',overwrite = T)
```

Bibliography

Grolemund, G. and Wickham, H. (2017). *R for Data Science*.

Wollschläger, D. (2016). *R kompakt: Der schnelle Einstieg in die Datenanalyse*. Springer-Lehrbuch. Springer Spektrum, second edition.