Datenanalyse mit R

SoSe 2020

Christina Bogner

Version vom 25. Mai 2020

Contents

1 Vorwort **5**
- 1.1 Organisatorisches . . . 5
- 1.2 Sinn und Unsinn dieses Skripts . . . 6

2 Der Kurs **7**
- 2.1 Zuordnung zum Modul und Leistungsnachweis . . . 7
- 2.2 Lernziele des Kurses . . . 7
- 2.3 Was mir im Umgang miteinander wichtig ist . . . 7

3 Erste Schritte in **9**
- 3.1 Was ist R? . . . 9
- 3.2 Was ist RStudio? . . . 10
- 3.3 RStudio Cloud . . . 10
- 3.4 Inhalt der live Einführung . . . 10

4 Daten in R **13**
- 4.1 Datenstrukturen erzeugen . . . 13
- 4.2 Arten von Daten in R . . . 15
- 4.3 Objekt, sag mir wer du bist . . . 15
- 4.4 Datenlücken, Fehlschläge etc. . . . 17
- 4.5 Inhalt der live Einführung . . . 17

5 Daten visualisieren I: Einfache Grafiken **19**
- 5.1 Plotten mit R-Basisfunktionen . . . 19
- 5.2 Tuning mit `par` . . . 25
- 5.3 Inhalt der live Einführung . . . 28

6 Reproduzierbare Forschung **29**
- 6.1 Warum Reproduzierbarkeit in der Forschung wichtig ist . . . 29
- 6.2 *Literate Programming* Idee von Donald Knuth . . . 29
- 6.3 Reproduzierbare Berichte mit R Markdown . . . 30
- 6.4 Wichtigste Regeln für Reproduzierbarkeit . . . 30
- 6.5 Weiterführende Videos und Literatur . . . 30
- 6.6 Inhalt der live Einführung . . . 30

7 Funktionen **31**
7.1 Eigene Funktionen schreiben 31
7.2 Fallunterscheidungen 34
7.3 `for`-Schleifen (`for` *loops*) 35
7.4 Weiterführende Literatur 36
7.5 Inhalt der live Einführung 36

8 Tidyverse **37**
8.1 Grundpakete 37
8.2 Der Workflow 38
8.3 Weiterführende Literatur und Videos 46
8.4 Inhalt der live Einführung 46

9 Aufgabensammlung **47**
9.1 Erste Schritte 47
9.2 Daten in R 48
9.3 Daten visualisieren, Teil I: Fokus auf R 48
9.4 Reproduzierbare Berichte mit R Markdown 51
9.5 Eigene Funktionen schreiben 51
9.6 Tidyverse 52
9.7 Daten visualisieren, Teil II: Fokus auf Daten 52
9.8 Effizientes Programmieren 54

Chapter 1

Vorwort

> "And honey, we're gonna do it in style"
>
> — Fools Garden

1.1 Organisatorisches

Die Coronaviruspandemie verändert unser Leben und unser Lernen. Die UzK bittet Lehrende, zumindest zu Beginn des SoSe 2020 auf digitale Lernformen umzusteigen. Daher wird dieser Kurs als ein Onlinekurs beginnen. Abhängig von der (sehr dynamischen) Lage werden wir im weiteren Kursverlauf das Format anpassen. Bitte seien Sie nachsichtig, wenn nicht alles so klappt, wie in Präsenzveranstaltungen. Wir müssen aktuell alle sehr viel dazu lernen in Sachen digitale Lehre. Sie können sicher sein, dass das Geographische Institut bemüht ist, die Lehre so effizient wie möglich weiter laufen zu lassen, damit Sie in Ihrem Studium fortfahren können.

In dieser Veranstaltung werden wir folgende Werkzeuge verwenden:

1. **ILIAS**: die Online-Lernplattform der UzK. Entweder sind Sie bereits automatisch in dem Kurs registriert oder werden von mir per Hand angemeldet.
2. **Campuswire**: die Live-Chatplattform dient der allgemeinen Kommunikation und der Selbstorganisation des Lernens. Verwenden Sie diese, um Fragen mit Ihren Kommilitonen und mir zu diskutieren. Sie sollten eine Einladungsmail zu Campuswire erhalten haben.
3. **Zoom**: die Videokonferenz-Software werden wir für live Einführungen nutzen. Die Anmeldemodalitäten sind auf den Kursseiten in ILIAS erklärt.

1.2 Sinn und Unsinn dieses Skripts

Dieses Skript ist ein lebendiges Begleitdokument des Kurses. Es wird laufend angepasst und aktualisiert.

Ich nutze verschiedene Farbkästen, um wichtige Stellen hervorzuheben:

Infoblock

Achtung, wichtig!

Beispielblock

Lernziele

Zusammenfassung

Chapter 2

Der Kurs

2.1 Zuordnung zum Modul und Leistungsnachweis

Dieser Kurs gehört zum Modul *Fachmethodik I* oder *Fachmethodik II* und ist aus 4 SWS Praktikum und 2 SWS Seminar aufgebaut. Das wichtigste Ziel besteht darin, Ihnen einen sicheren Umgang mit `R` beizubringen.

Den Leistungsnachweis bildet ein benoteter Praktikumsbericht.

2.2 Lernziele des Kurses

- Daten für Analysen vorbereiten
- eigene wiederverwendbare Skripte schreiben
- eigene Funktionen schreiben
- einfache Datenanalysen durchführen
- Daten visualisieren
- Ergebnisse reproduzierbar im Praktikumsbericht darstellen

2.3 Was mir im Umgang miteinander wichtig ist

- Pünktlichkeit bei live und Präsenzsitzungen
- Gute Vorbereitung durch erledigen der blenden learning Einheiten und Hausaufgaben
- Respektieren anderer Meinungen
- Offenheit gegenüber neuen Sichtweisen, Themen und Methoden
- Geduld mit sich selbst und den anderen

Chapter 3

Erste Schritte in

- Layout und Bedeutung einzelner Fenster in RStudio kennen
- Anweisungen aus dem Skript an die Konsole schicken
- R als Taschenrechner benutzen
- erste Funktionen aufrufen
- Objekte mit eckigen Klammern [ ] ansprechen
- R-Hilfeseiten aufrufen

3.1 Was ist R?

R ist eine Programmiersprache für Datenanalyse und statistische Modellierung. Es ist frei verfügbar (*open source software*) und neben Python einer der am meisten benutzten Programmiersprachen zur Datenanalyse und -visualisierung. R wurde von Ross Ihaka und Robert Gentleman 1996 veröffentlicht Ihaka and Gentleman (1996). Es gibt für R eine Vielzahl von Zusatzpaketen, die die Funktionalität und die Einsatzmöglichkeiten enorm erweitern.

Sie können R für Ihren Computer auf der offiziellen R-Seite https://www.r-project.org/ herunter laden und installieren. Auch die Pakete finden Sie dort unter CRAN (*The Comprehensive R Archive Network*). Auf den CRAN-Seiten finden Sie sogen. CRAN Task Views, eine Übersicht über Pakete in verschiedenen Themenbereichen. Für den Umweltbereich sind folgende Paketsammlungen besonders relevant:

- Environmetrics: Analyse von Umweltdaten
- Multivariate: Multivariate Statistik
- Spatial: Analyse von räumlichen Daten
- TimeSeries: Zeitreihenanalyse

Zu Beginn des Kurses, werden wir jedoch nicht auf Ihren lokalen Rechnern arbeiten, sondern in einer Cloud (s.u.). Das ermöglicht einen schnelleren Einstieg in R und bietet eine live Unterstützung durch den Dozenten beim Pro-

grammieren. Daher biete ich zu diesem frühen Zeitpunkt im Kurs keine Unterstützung bei der Installation. Für die ganz Ungeduldigen, gibt es hier eine kurze *Einleitung zur Installation*

3.2 Was ist RStudio?

RStudio Desktop ist eine Entwicklungsumgebung für R. Sie können die *open source* Version kostenlos für Ihren Rechner hier herunterladen.

Es gibt eine live Einführung in RStudio im Kurs. Zusätzlich können Sie hier ein Video dazu ansehen.

3.3 RStudio Cloud

Zu Beginn des Kurses werden wir in der RStudio Cloud arbeiten. Sie sollten eine Einladungsmail zu unserem Kurs in der Cloud bekommen haben. Ich werde in der Cloud Projekte für Sie anlegen (*assignment*), die Skripte, Arbeitsanweisungen etc. beinhalten. Wenn Sie auf so ein Assignment klicken, wird für Sie automatische ein Kopie des Projekts erstellt, in der Sie dann arbeiten können.

Der große Vorteil der Cloud ist, dass ich direkt in Ihre Projekte eingreifen kann, wenn es mal zu Fehlern kommt. Während ich in Ihrem Projekt arbeite, werden Sie kurz aus der R-Sitzung ausgeloggt, da die Cloud kein gleichzeitiges Arbeiten unterstützt. Nehmen Sie sich etwas Zeit, um die Cloud und die darin enthaltenen Tutorials kennen zu lernen.

Sowohl in der RStudio Cloud als auch in einer lokalen Installation, ist Ihr RStudio so aufgebaut wie in Abbildung 3.1.

3.4 Inhalt der live Einführung

- Überblick über RStudio
- R als Taschenrechner
- einfache Funktionen aufrufen
- Zuordnungen (*assignments*)
- Notation mit eckigen Klammern [ ] (*array*-Notation)
- Hilfeseiten aufrufen

Funktionen, die wir in der Session nutzen werden:

Funktion	Bedeutung	Beispielaufruf
`pi`	Zahl pi	`pi`
`sin`	Sinus	`sin(2)`
`cos`	Cosinus	`cos(2)`
`sqrt`	Quadratwurzel	`sqrt(2)`

Funktion	Bedeutung	Beispielaufruf
`c`	(*concatenate*) Fügt Daten zu einem Vektor zusammen	`c(1,2,3,4)`
`help.start`	Öffnet ein Browser-Fenster mit diversen Handbüchern	`help.start()`
`help.search`	Sucht nach einem Begriff in Hilfe-Dateien	`help.search('time')`
`??`	alias `help.search`	`??time`
`help`	Sucht nach einer Funktion	`?mean`
`?`	alias `help()`	`?mean`
`mean`	Mittelwert	`mean(c(1,2,3,4))`
`var`	Varianz	`var(c(1,2,3,4))`
`sd`	Standardabweichung	`sd(c(1,2,3,4))`
`sum`	Summe	`sum(c(1,2,3,4))`
`vector`	Generiert einen Vektor	`vector(length=3, mode='numeric')`

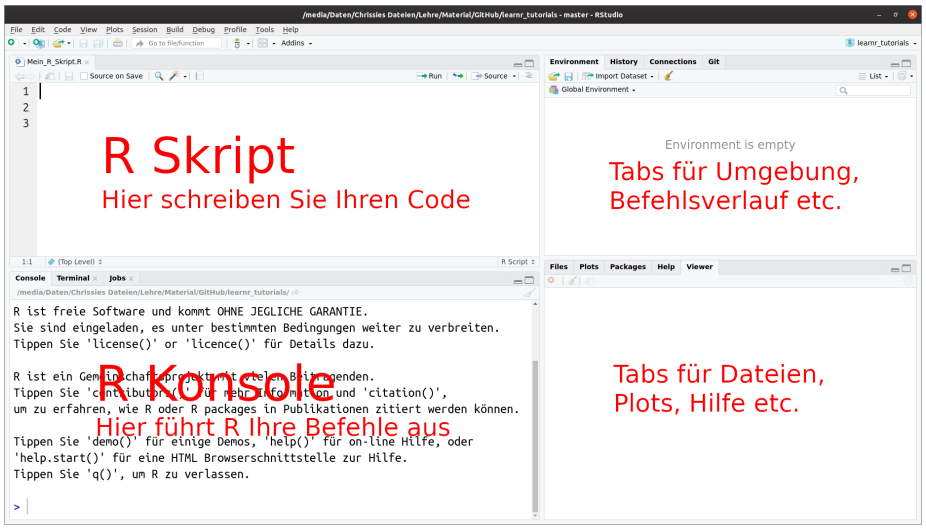


Figure 3.1: Aufbau von RStudio

Chapter 4

Daten in R

- Daten einlesen mit `read.table`
- Datenstrukturen erstellen
- Typen von Daten in R abfragen
- Daten speichern mit `write.table`

4.1 Datenstrukturen erzeugen

In R gibt es unterschiedliche Datenobjekte. Es ist wichtig, sich über die Struktur (oder Typ) des Datenobjekts Gedanken zu machen. Denn diese bestimmt, was mit einem Objekt gemacht werden kann und ob Funktionen damit richtig umgehen können. Schließlich ist es nicht egal, ob es sich bei einem Objekt um ein numerisches Objekt oder einfach Text (*character*) handelt.

Die wichtigsten Datentypen sind

- **Vektoren**: hier gruppiert man gleichartige Elemente, z.B. Zahlen. Auch eine einzelne Zahl (ein Skalar) wird von R wie ein Vektor behandelt.
- **Matrizen**: zweidimensionale (Zeilen und Spalten) Datentabellen mit gleichartigen Elementen.
- **Listen**: können beliebige Elemente beliebiger Länge enthalten.
- **Dataframes**: zweidimensionale Datentabellen, die beliebige Elemente enthalten können. Die Spalten der Dataframes müssen allerdings gleichartige Elemente enthalten. Dataframes sind eine Unterart von Listen.

Neben diesen Hauptstrukturen gibt es

- **Factor**: ein besonderer Vektor für kategorielle Variablen

Um diese Datenstrukturen zu erzeugen, gibt es jeweils eine Funktion mit gleichlautendem Namen.

```r
# Vektor erzeugen
my_vect = vector(length = 3, mode = 'numeric')
my_vect
```

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

```r
# Matrix erzeugen
my_matrix = matrix(data = c(1:(3*4)), nrow = 3, ncol = 4)
my_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```r
# Dataframe erzeugen
my_dataframe = data.frame('Spalte_1' = rep('Text', 10),
                          'Spalte_2' = 1:10)
my_dataframe
```

```
##    Spalte_1 Spalte_2
## 1      Text        1
## 2      Text        2
## 3      Text        3
## 4      Text        4
## 5      Text        5
## 6      Text        6
## 7      Text        7
## 8      Text        8
## 9      Text        9
## 10     Text       10
```

```r
# Liste erzeugen
my_list = list('Schachtel_1' = 3, 'Schachtel_2' = my_dataframe,
               'Schachtel_3' = 'Noch mehr Text')
my_list
```

```
## $Schachtel_1
## [1] 3
##
## $Schachtel_2
##    Spalte_1 Spalte_2
## 1      Text        1
## 2      Text        2
```

```
## 3      Text        3
## 4      Text        4
## 5      Text        5
## 6      Text        6
## 7      Text        7
## 8      Text        8
## 9      Text        9
## 10     Text       10
##
## $Schachtel_3
## [1] "Noch mehr Text"
```

```
# Factor erzeugen
my_factor = factor(c('R', 'RStudio', 'Cloud', 'Cloud', 'R', 'R'))
my_factor
```

```
## [1] R       RStudio Cloud   Cloud   R       R
## Levels: Cloud R RStudio
```

4.2 Arten von Daten in R

Die Datenstrukturen `vector`, `data.frame` usw. können unterschiedliche Arten von Daten enthalten.

Name	Beispiele
raw	3A, FE
logical	TRUE, FALSE
integer	1, 42, -3
numeric/double	3, 2.81, 6.032e23
complex	1.2+2.2i
character	“foo”

4.3 Objekt, sag mir wer du bist

Um die Struktur und/oder Datenart abzufragen, verwendet man `class`, `typeof`, `mode` und `storage.mode`.

```
class(my_vect)
```

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

```
typeof(my_vect)
```

```
## [1] "double"
```

```
class(my_dataframe)
```

```
## [1] "data.frame"
```

```
typeof(my_dataframe)
```

```
## [1] "list"
```

Mit `str` kann man das Innenleben eines Objekts anzeigen. Das ist besonders wichtig nach dem Einlesen von Daten, um das Ergebnis des Einlesens zu kontrollieren. Dabei kontrolliert man, dass z.B. alle numerischen Spalten auch als Zahlen eingelesen wurden und nichts schief gegangen ist.

```
str(my_dataframe)
```

```
## 'data.frame':    10 obs. of  2 variables:
##  $ Spalte_1: Factor w/ 1 level "Text": 1 1 1 1 1 1 1 1 1 1
##  $ Spalte_2: int  1 2 3 4 5 6 7 8 9 10
```

Weiter Funktionen, die Auskunft über Objekte geben sind `length`, sinnvoll auf nur Vektoren und Listen, und `dim`, sinnvoll auf zweidimensionalen Datenobjekten. Wenn Sie versuchen, `dim` auf einem Vektor aufzurufen, gibt es `NULL` (s.u.), weil Vektoren keine Dimensionen haben. Wenn Sie `length` auf einem `data.frame` aufrufen, bekommen Sie die Anzahl der Dimensionen, nämlich 2. Das sind keine besonders spannenden Informationen .

```
length(my_vect)
```

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

```
dim(my_vect)
```

```
## NULL
```

```
length(my_dataframe)
```

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

```
dim(my_dataframe)
```

```
## [1] 10  2
```

4.4 Datenlücken, Fehlschläge etc.

Datenlücken werden in R mit `NA` kodiert, Fehlschläge bei Berechnungen mit `NaN` (not a number) und Vektoren der Länge 0 mit `NULL`. Letzteres wird häufig beim Aufruf von Funktionen benutzt, wenn man bestimmte Parameter ausschalten möchte. Die Benutzung muss aber immer in der Hilfe zur jeweiligen Funktion nachgeschlagen werden.

4.5 Inhalt der live Einführung

- Daten einlesen und `data.frame` erstellen: Aufgabe 9.2.1

Funktionen, die wir in der Session nutzen werden:

Funktion	Bedeutung	Beispielaufruf
`read.table`	Liest Daten aus einer Datei ein.	`read.table(file= 'Daten.txt', header=TRUE)`
`ls`	Zeigt den Inhalt des Workspaces.	`ls`
`head`	Zeigt den ersten Teil eines Objekts.	`head(x)`
`tail`	Zeigt den letzten Teil eines Objekts.	`tail(x)`
`str`	Zeigt die Struktur (Innenleben) eines Objekts an	`str(my_dataframe)`
`length`	Gibt die Länge eines Objekts.	`length(x)`
`dim`	Gibt die Dimension eines Objekts (Reihenfolge: Zeilen, Spalten)	`dim(x)`
`seq`	Erstellt eine regelmäßige Reihe.	`seq(from=-2, to=4, by=0.1)`
`data.frame`	Erstellt eine Datentabelle.	`data.frame(x,y,z)`
`colnames`, `rownames`	Benennt Spalten bzw. Zeilen eines Datenobjekts.	`colnames(x)`
`rm`	Löscht Objekte aus dem Workspace.	`rm(x)`
`summary`	Fasst ein Objekt zusammen.	`summary(x)`
`table`	Erstellt eine Häufigkeitstabelle.	`table(x)`

Funktion	Bedeutung	Beispielaufruf
`which`	Gibt die TRUE-Indices eines logischen Objekts.	`which(LETTERS == 'R')`
`history`	Zeigt die Liste mit ausgeführten Befehlen der Session.	`history`
`write.table`	Speichert Datenobjekte als Tabelle ab.	`write.table(x, file='Tabelle.txt')`
`save.image`	Speichert den Workspace.	`save.image(file= 'RSession.Rdata')`
`savehistory`	Speichert die History.	`savehistory(file= 'Myhistory.Rhistory')`

Chapter 5

Daten visualisieren I: Einfache Grafiken

- Einfache Grafiken erstellen
- Grafiken beschriften und speichern
- Die Arbeitsweise der Funktion `par` beschreiben
- Die grafischen Parameter für Randgröße, Farbe, Schrift- und Symbolgröße einstellen
- Unterschiede zwischen *high-level* und *low-level* Grafikfunktionen erklären
- Grafiken mit mehreren Plots erstellen

5.1 Plotten mit R-Basisfunktionen

Für Grafikverliebte und Neugierige empfehle ich die Kapitel 2 und 3 in Murrell (2006).

5.1.1 *High-level* Grafikfunktion `plot` und *low-level* Grafikfunktion `lines`

Ein Streudiagramm stellt zwei numerische Variablen gegeneinander dar. Wir betrachten Klimadaten der Station Köln-Bonn, die man beim Deutschen Wetterdienst herunterladen kann (https://www.dwd.de/DE/leistungen/klimadatendeutschland/klimadatendeutschland.html).

Sie können den Code aus den Chunks leicht herauskopieren und in RStudio laufen lassen (rechts oben in den Chunks auf das Symbol *copy to clipboard* klicken).

Wir lesen die Daten ein und sehen uns deren Struktur an.

```
meteo <- read.table('produkt_klima_monat_20181001_20200430_02667.txt',
header = T, sep = ';')
str(meteo)
```

```
## 'data.frame':    19 obs. of  17 variables:
##  $ STATIONS_ID      : int  2667 2667 2667 2667 2667 2667 2667 2667 2667 2667 ...
##  $ MESS_DATUM_BEGINN: int  20181001 20181101 20181201 20190101 20190201 20190301 201
##  $ MESS_DATUM_ENDE  : int  20181031 20181130 20181231 20190131 20190228 20190331 201
##  $ QN_4             : int  3 3 3 3 3 3 3 3 3 3 ...
##  $ MO_N             : num  4.76 5.51 6.49 6.66 4.79 5.62 4.56 5.37 3.85 4.95 ...
##  $ MO_TT            : num  12.01 7.31 5.64 2.41 6.4 ...
##  $ MO_TX            : num  17.58 10.95 8.47 4.74 12 ...
##  $ MO_TN            : num  6.41 3.51 2.61 -0.26 1.12 ...
##  $ MO_FK            : num  2.42 2.57 2.68 2.84 2.54 3.06 2.53 2.32 2.4 2.23 ...
##  $ MX_TX            : num  26.7 19.2 15 8.8 21 20.4 25.9 24.3 36.2 40.3 ...
##  $ MX_FX            : num  15.3 16.5 18.9 22.8 18.7 28.8 19.5 16.5 26.1 14.6 ...
##  $ MX_TN            : num  0.4 -4.3 -4.9 -10.7 -3.5 -2.8 -2.3 -1.8 7.3 4.4 ...
##  $ MO_SD_S          : num  145.4 74.2 33.8 38.2 127.3 ...
##  $ QN_6             : int  9 9 9 3 3 3 3 3 3 3 ...
##  $ MO_RR            : num  26.5 25 101.9 101.8 30 ...
##  $ MX_RS            : num  6.9 9.7 15.7 22.3 12.3 18.2 21.4 10.6 8.8 11.9 ...
##  $ eor              : Factor w/ 1 level "eor": 1 1 1 1 1 1 1 1 1 1 ...
```

Uns interessieren hier nur die Spalten MO_TT, MO_TN, MO_TX und MESS_DATUM_BEGINN. Das sind jeweils die Monatsmittel der Lufttemperatur in 2 m Höhe, Monatsmittel des Minimums der Lufttemperatur, Monatsmittel des Maximums der Lufttemperatur und der Beginn der jeweiligen Messperiode (d.h. des Kalendermonats). Um die Daten als Zeitreihen darstellen zu können, wandeln wir die Spalte MESS_DATUM_BEGINN in ein richtiges Zeitobjekt (d.h. ein Objekt der Klasse *Date*). Das geht mit der Funktion `as.Date`. Der Parameter `format` beschreibt den Aufbau des Datums im Objekt `meteo`: erst steht das Jahr mit 4 Zeichen (z.B. 2018), dann folgt der Monat mit 2 Zeichen (z.B. 01) und dann der Tag mit 2 Zeichen (z.B. 01). Näheres zu Datumsformaten finden Sie mit `?strptime`.

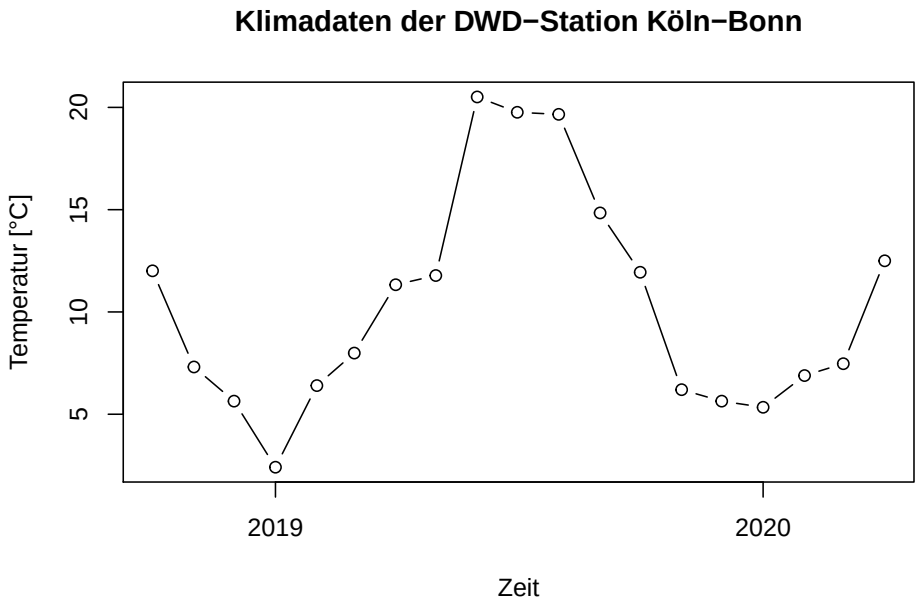
```
my_date <- as.Date(as.character(meteo$MESS_DATUM_BEGINN), format = '%Y%m%d')
my_date
```

```
##  [1] "2018-10-01" "2018-11-01" "2018-12-01" "2019-01-01" "2019-02-01" "2019-03-01"
##  [7] "2019-04-01" "2019-05-01" "2019-06-01" "2019-07-01" "2019-08-01" "2019-09-01"
## [13] "2019-10-01" "2019-11-01" "2019-12-01" "2020-01-01" "2020-02-01" "2020-03-01"
## [19] "2020-04-01"
```

Es sind Daten von Oktober 2018 bis April 2020. Wir erstellen ein Streudiagramm mit der Funktion `plot`. Mit den Parametern `xlab` und `ylab` lassen sich

die beiden Achsen beschriften und `main` fügt einen Titel dazu. Der Parameter `type` bestimmt die Wahl der Symbole; hier benutzen wir `type = b` für *both*, also sowohl Punkte als auch Linien.

```
plot(my_date, meteo$MO_TT, type = 'b', xlab = 'Zeit', ylab = 'Temperatur [°C]', main = 'Klimadate
```



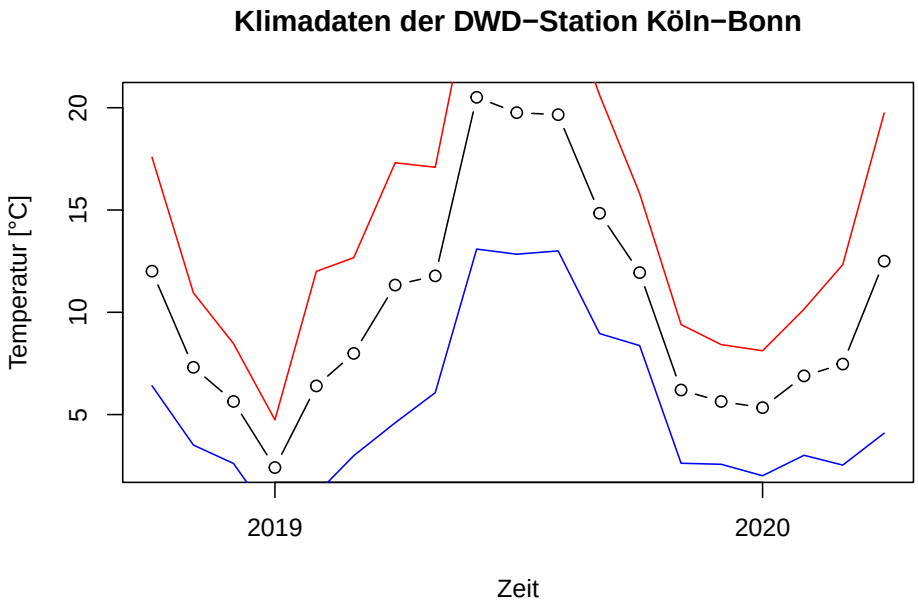
Die Funktion `plot` ist eine sogen. *high-level* Grafikfunktion. Das bedeutet, dass sie alle Schritte des Plottens übernimmt: sie öffnet ein neues Grafikfenster (ein Device), berechnet die Größe der Plotfläche und der Ränder (s. unten), berechnet die Ausdehnung der Achsen und die beste Achseneinteilung und plottet Ihre Daten.

Daneben gibt es *low-level* Grafikfunktionen, die nur in ein bestehendes Device plotten können. Wir wollen zu unserer Grafik nun die Minimum- und die Maximumtemperatur dazu plotten.

```
plot(my_date, meteo$MO_TT, type = 'b', xlab = 'Zeit', ylab = 'Temperatur [°C]', main = 'Klimadate

# Minimumtemperatur in blau
lines(my_date, meteo$MO_TN, col = 'blue')

# Maximumtemperatur in rot
lines(my_date, meteo$MO_TX, col = 'red')
```

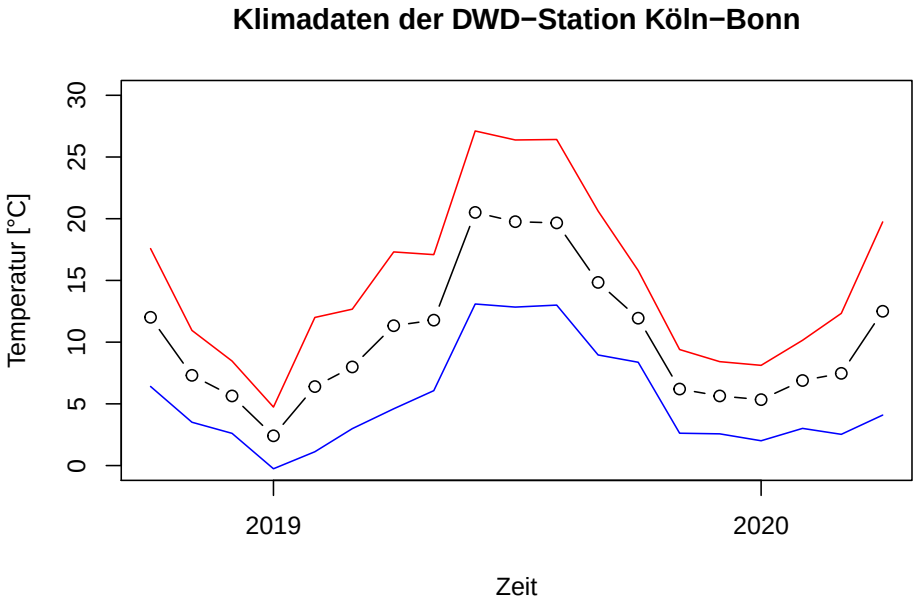


Dass `lines` nur eine *low-level* Grafikfunktion ist, erkennen Sie daran, dass sie nicht in der Lage ist, den Bereich auf der y-Achse zu vergrößern, um alle Daten sichtbar zu machen. Das kann nur `plot`. Daher muss der Bereich bereits in `plot` richtig festgelegt werden. Das macht der Parameter `ylim`.

```
plot(my_date, meteo$MO_TT, type = 'b', xlab = 'Zeit', ylab = 'Temperatur [°C]', main =

# Minimumtemperatur in rot
lines(my_date, meteo$MO_TN, col = 'blue')

# Maximumtemperatur in blau
lines(my_date, meteo$MO_TX, col = 'red')
```



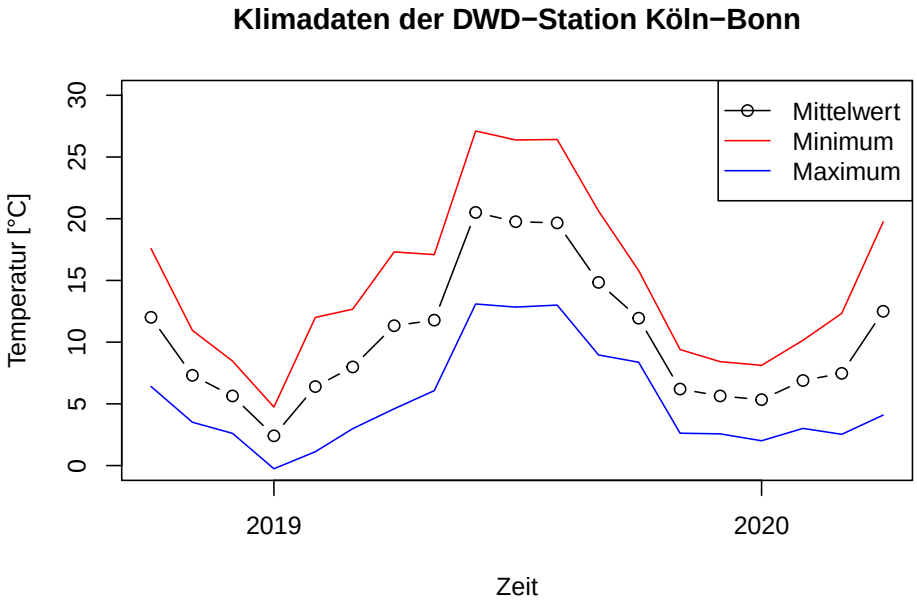
Wenn in einer Grafik mehrere Elemente dargestellt werden, benötigt man eine Legende. Das erledigt die * low-level* Grafikfunktion `legend`.

```
plot(my_date, meteo$MO_TT, type = 'b', xlab = 'Zeit', ylab = 'Temperatur [°C]', main = 'Klimadate

# Minimumtemperatur in rot
lines(my_date, meteo$MO_TN, col = 'blue')

# Maximumtemperatur in blau
lines(my_date, meteo$MO_TX, col = 'red')

legend('topright', legend = c('Mittelwert', 'Minimum', 'Maximum'),
       col = c('black', 'red', 'blue'),
       pch = c(1, NA, NA),
       lty = 1)
```



Der Parameter `lty` steht für *line type* und die 1 bedeutet durchgezogene Linie. Mit `pch` legend wir die Art des Symbols fest; hier steht die 1 für das Standardsymbol "offener Kreis". Die Funktion `legend` hat viele Möglichkeiten und es lohnt sich, in die Hilfe zu sehen `?legend`.

5.1.2 Überblick über die wichtigsten *high-level* und *low-level* Grafikfunktionen

Die wichtigsten *high-level* Grafikfunktionen nach Ligges (2008), verändert:

Funktion	Beschreibung
`plot`	kontextabhängig – generische Funktion mit vielen Methoden
`barplot`	Säulendiagramm
`boxplot`	Boxplot
`contour`	Höhenlinien-Plot
`coplot`	Conditioning-Plots: Plots zweier Variablen aufgeteilt nach Werten einer dritten
`curve`	Funktionen zeichnen
`dotchart`	Dotplots (nach Cleveland)
`hist`	Histogramm
`image`	Bilder (3. Dimension als Farbe)
`mosaicplot`	Mosaikplots (kategorielle Daten)
`pairs`	Streudiagramm-Matrix
`persp`	perspektivische Flächen
`qqnorm` und `qqplot`	QQ–Plot

Die wichtigsten *low-level* Grafikfunktionen nach Ligges (2008), verändert:

Funktion	Beschreibung
`abline`	Fügt eine Linie hinzu; diese kann horizontal, vertikal oder über Steigung und Achsenabschnitt definiert werden
`arrows`	Pfeile
`axis`	Achsen
`grid`	Gitternetz
`legend`	Legende
`lines`	Linien (schrittweise)
`mtext`	Text in den Rändern
`plot.new`	Grafik initialisieren
`plot.window`	Koordinatensystem initialisieren
`points`	Punkte
`polygon`	(ausgefüllte) Polygone
`pretty`	berechnet "hübsche" Einteilung der Achsen
`segments`	Linien (vektorwertig)
`text`	Text
`title`	Beschriftung

5.2 Tuning mit par

Zur Vertiefung dieses Kapitels, empfehle ich Ligges (2008), Kapitel 8.1.3.

Die Grafikebene in R ist aufgeteilt in drei Regionen (Abbildung 5.1) und hat innere und äußere Ränder. Die Ränder werden von unten im Gegenuhrzeigersinn durchnummeriert.

Mit der Funktion `par` lassen sich sehr viele Einstellung der Grafik verändern. Viele Einstellungen übergibt die Funktion `plot` selbständig an `par`, zu.B. `log` (Logarithmieren der Achsen), `cex` (Größe eines Punkts) oder `col` (Farbe). Andere können aber nur durch Aufrufen der Funktion `par` verändert werden. Dazu gehören die inneren Ränder `mar` und die äußeren Ränder `oma`, die Aufteilung der Grafikebene mit `mfrow` oder `mfcol`.

Richtige Benutzung von `par`:

- Parameter setzen: `op <- par( ... )`
- plotten
- Parameter auf Standard zurück setzen: `par(op)`

Die Zuweisung `op <- par( ... )` speichert die Standardeinstellungen im Objekt `par`, bevor Sie sie ändern. Der Aufruf `par(op)` setzt Ihre Änderungen zurück. Das ist sehr praktisch, wenn Sie z.B. die Aufteilung der Grafikebene nicht mehr benötigen. Wenn Sie die Parameter nicht zurücksetzen, bleiben diese bestehen, bis das Grafikfenster geschlossen wird (z.B. mit `dev.off()`).

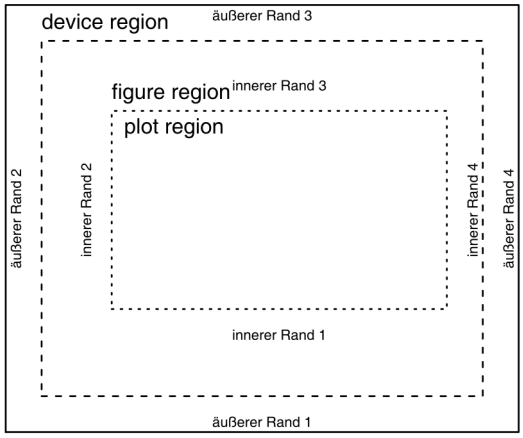


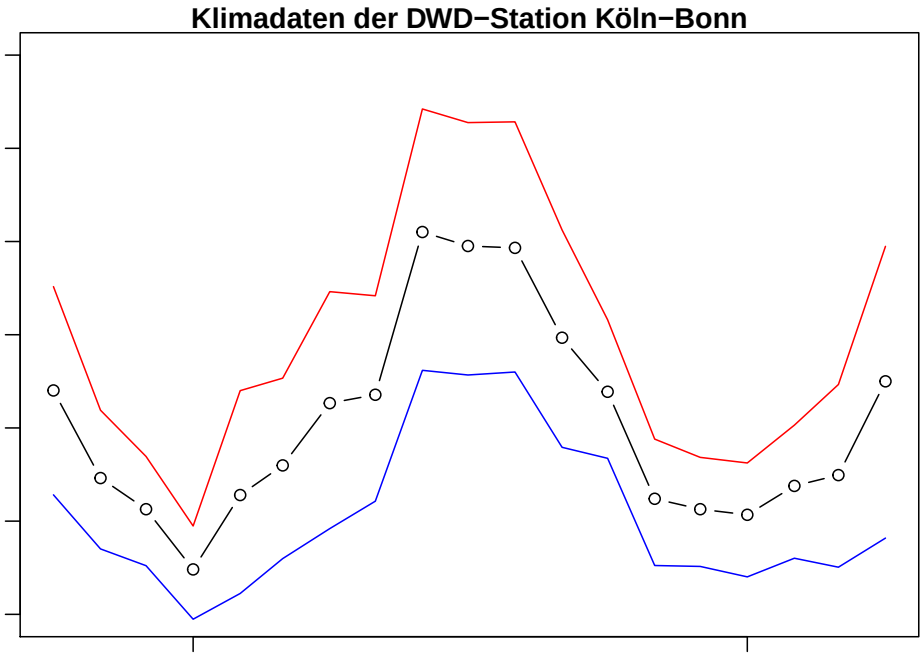
Figure 5.1: Aufteilung der Grafikfläche [@Ligges2008].

Um die Ränder zu verändern, rufen wir `par` auf und beschneiden die Ränder, damit Sie den Unterschied erkennen können.

```
op <- par(mar = c(1, 1, 1, 1))
plot(my_date, meteo$MO_TT, type = 'b', xlab = 'Zeit', ylab = 'Temperatur [°C]', main =

# Minimumtemperatur in rot
lines(my_date, meteo$MO_TN, col = 'blue')

# Maximumtemperatur in blau
lines(my_date, meteo$MO_TX, col = 'red')
```



Die Achsenbeschriftungen und die Zahlen haben jetzt nicht mehr genug Platz und verschwinden. Die Größe der Ränder wird in Zeilen angegeben, ist also relativ zur Gesamtgröße. Die Standardeinstellung ist `c(5, 4, 4, 2) + 0.1`.

Einige häufig genutzte Argumente in Grafikfunktionen und in `par` (nach Ligges, 2008, verändert). Schlagen Sie die Erklärungen dazu immer in `?par` oder `?plot` nach.

Funktion	Beschreibung
`axes`	Achsen sollen (nicht) eingezeichnet werden
`bg`	Hintergrundfarbe
`cex`	Größe eines Punktes bzw. Buchstaben
`col`	Farben
`las`	Ausrichtung der Achsenbeschriftung
`log`	Logarithmierte Darstellung
`lty`, `lwd`	Linientyp (gestrichelt, …) und Linienbreite
`main`	Überschrift
`mar`	Größe der inneren Ränder für Achsenbeschriftung etc.
`mfcol`, `mfrow`	mehrere Grafiken in einem Bild
`pch`	Symbol für einen Punkt
`type`	Typ (l für Linie, p für Punkt, b für beides, n für nichts)
`usr`	Ausmaße der Achsen auslesen
`xlab`, `ylab`	x-/y-Achsenbeschriftung
`xlim`, `ylim`	zu plottender Bereich in x-/y- Richtung

Funktion	Beschreibung
`xpd`	in die Ränder hinein zeichnen

5.3 Inhalt der live Einführung

- `plot`, `barplot`, `mfrow`
- Aufgaben 9.3.1, 9.3.2 und 9.3.3.
- Speichern als pdf

Chapter 6

Reproduzierbare Forschung

- Wichtigkeit der Reproduzierbarkeit erklären
- Begriff *literate programming* definieren
- Aufbau einer RMarkdown-Datei erklären
- Einen einfachen ersten reproduzierbaren Bericht schreiben

6.1 Warum Reproduzierbarkeit in der Forschung wichtig ist

6.2 *Literate Programming* Idee von Donald Knuth

Die Idee, dass man den Code und die dazugehörige Interpretation (Text, Bericht etc.) nicht von einander trennen sollte, geht auf Knuth (1984) zurück. Mit *Literate Programming* meinte Knuth, dass Programme auch nichts anderes wie literatirsche Werke sind. Er setzte den Fokus darauf, mit Programmen menschlichen Benutzern zu erklären, was man den Computer machen lassen möchte. Also weg vom computer- hin zum menschzentrierten Zugang. So wird Programmieren und in unserem Fall die Datenanalyse verständlich und vor allem reproduzierbar.

Leider ist es in unserer modernen Forschungslandschaft immer noch nicht Standard. Das Trennen von Analyseergebnissen und Berichten (Forschungsartikeln) sorgt für viele (unendeckte und unnötige) Fehler und Frust.

6.3 Reproduzierbare Berichte mit R Markdown

R hat sein eigenes System von reproduzierbaren Berichten, genannt R Markdown (Xie et al., 2018). Es ist benutzerfreundlich und ermöglicht unterschiedliche Formate von Berichten, wie HTML-Dokumente, PDF-Dateien, Präsentationsfolien usw.

Es wird Sie vielleicht überraschen, aber das Skript, das Sie gerade lesen ist nichts anderes als ein "literarisch" programmierter Bericht in R Bookdown (Xie, 2016), einem R-Paket speziell für lange R Markdown-Dokumente.

Wir werden vor allem mit R Notebooks arbeiten, die eine gute Interaktion zwischen dem geschriebenen Text und dem R-Code ermöglichen. Das Notebook kann sowohl in ein HTML-Dokument als auch in PDF oder Word als endgültiges Berichtdokument umgewandlet werden. Diesen Prozess nennt man *knit* (der Knopf in RStudio mit dem Wollknäuel).

6.4 Wichtigste Regeln für Reproduzierbarkeit

6.5 Weiterführende Videos und Literatur

Die Playlist zu *Reporducible Research* finden Sie hier.

Report Writing for Data Science in R (Peng, 2019) (auf ILIAS)

6.6 Inhalt der live Einführung

- Erstellen eines einfachen R Notebooks
- R-Code Chunks
- Einfache Layoutelemente: Überschriften, Listen, fett und kursiv

Chapter 7

Funktionen

- Den Aufbau von Funktionen in R beschreiben
- Den Aufruf von Funktionen in R erklären
- Einfache Funktionen selbst schreiben
- Fallunterscheidungen
- `for` Schleifen

Was genau Funktionen sind und wie man sie in R aufruft, lesen Sie bitte bei Ligges (2008) in Kapitel 4.1 nach. In diesem Kapitel des Skripts geht es um das Schreiben der eigenen Funktionen, Fallunterscheidungen mit `if-else` und `for` Schleifen.

7.1 Eigene Funktionen schreiben

Funktionen sind eine großartige Möglichkeit, sich das Leben einfacher zu machen. Sie können repetitive Aufgaben erledigen, ohne dass wir ständig mit Copy und Paste hantieren müssen, machen unsere Notebooks übersichtlich und helfen, Fehler und Inkonsistenzen zu vermeiden. Als Faustregel gilt: wenn Sie ein Stück Code mehr als 2 Mal kopieren und abändern, wird es Zeit für eine Funktion .

Schauen wir uns ein Beispiel an. Als erstes würfeln wir ein paar Daten aus der Gleichverteilung mit unterschiedlichen Minima und Maxima und sehen uns die Zusammenfassung und die Standardabweichungen an.

```
my_data <- data.frame(col1 = runif(20, min = 3, max = 10),
                      col2 = runif(20, min = 7, max = 12),
                      col3 = runif(20, min = 13, max = 100))
summary(my_data)
```

```
##       col1            col2             col3
##  Min.   :3.523   Min.   : 7.164   Min.   :14.13
```

```
##  1st Qu.:4.853   1st Qu.: 8.237   1st Qu.:38.80
##  Median :6.065   Median : 9.818   Median :59.96
##  Mean   :6.099   Mean   : 9.545   Mean   :58.46
##  3rd Qu.:7.083   3rd Qu.:10.345   3rd Qu.:72.71
##  Max.   :9.195   Max.   :11.742   Max.   :99.62
```

```
sapply(my_data, sd)
```

```
##      col1      col2      col3
##  1.661756  1.269660 25.740582
```

Ich möchte die Daten so transformieren, dass alle Variablen im Datensatz (d.h. Spalten) einen Mittelwert von 0 und eine Standardabweichung von 1 haben. Dazu ziehe ich den Mittelwert ab und teile durch die Standardabweichung. Das ist eine klassische Transformation (manchmal z-Transformation genannt), die manche Analysemethoden (z.B. Hauptkomponentenanalyse) als Vorbehandlung der Daten verlangen.

Erst einmal das Naheliegende: Transformation per Copy und Paste.

```
my_data_trans <- data.frame(col1_trans = (my_data$col1 - mean(my_data$col1))/sd(my_data
                            col2_trans = (my_data$col2 - mean(my_data$col2))/sd(my_data
                            col3_trans = (my_data$col3 - mean(my_data$col3))/sd(my_data
```

Haben Sie den Fehler bemerkt? Ich habe einmal vergessen eine 1 durch eine 2 zu ersetzten. Das merke ich aber nur, wenn ich mir das Ergebnis ansehen. Die Standardabweichung in `col2_trans` ist nicht gleich 1.

```
summary(my_data_trans)
```

```
##    col1_trans         col2_trans        col3_trans
##  Min.   :-1.55056   Min.   :-1.4328   Min.   :-1.72228
##  1st Qu.:-0.75016   1st Qu.:-0.7871   1st Qu.:-0.76381
##  Median :-0.02031   Median : 0.1645   Median : 0.05847
##  Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.00000
##  3rd Qu.: 0.59195   3rd Qu.: 0.4816   3rd Qu.: 0.55383
##  Max.   : 1.86303   Max.   : 1.3218   Max.   : 1.59914
```

```
sapply(my_data_trans, sd)
```

```
## col1_trans col2_trans col3_trans
##  1.0000000  0.7640471  1.0000000
```

So etwas passiert sehr schnell und wird durch das Schreiben einer Funktion vermieden.

Jede Funktionsdefinition beginnt mit der Wahl des Namens. Es ist eine gute

Idee, sich einen konsistenten und sauberen Stil gleich am Anfang anzugewöhnen. Seien Sie nett zu Ihrem zukünftigen Ich und anderen Menschen, die Ihren Code lesen werden . Es ist ein guter Stil, Verben als Funktionsnamen zu nutzen, die beschreiben, was eine Funktion macht. In jedem Fall, **wählen Sie keine Namen, die schon für Funktionen oder Variablen vergeben sind!** Das würde die ursprünglichen Funktionen oder Variablen einfach überschreiben.

Wir nennen unsere Funktion `z_transform`, weil sie Daten z-transformiert. Bei zusammengesetzten Namen sollten Sie den Unterstrich verwenden. Das macht den Namen einfacher zu lesen.

```
z_transofrm <- function(x) {
  (x - mean(x))/sd(x)
}
```

Bei jeder Funktionsdefinition arbeiten Sie drei Schritte ab:

1. Namen finden und das Skelett aus `Name_der_Funktion <- function() {}` hin schreiben.
2. Überlegen, welche Parameter die Funktion braucht und ob diese Standardwerte benötigen. Parameter zwischen die runden Klammern schreiben.
3. Funktionskörper (*body*), also die eigentliche Aufgabe der Funktion, hinschreiben.

Wir brauchen für unsere Funktion nur einen Parameter. Zwar sind die Namen der Parameter nicht ganz so wichtig wie Namen der Funktionen. Trotzdem sollte man hier konsistent sein: bei einfachen Vektoren sind `x`, `y`, `z` etc. völlig in Ordnung. Ansonsten sind Substantive ein guter Namensstil für Parameter.

Wir wenden die Funktion jetzt auf unseren Datensatz an.

```
my_data_trans_2 <- data.frame(col1_trans = z_transofrm(my_data$col1),
                              col2_trans = z_transofrm(my_data$col2),
                              col2_trans = z_transofrm(my_data$col3))
summary(my_data_trans_2)
```

```
##    col1_trans         col2_trans       col2_trans.1
##  Min.   :-1.55056   Min.   :-1.8753   Min.   :-1.72228
##  1st Qu.:-0.75016   1st Qu.:-1.0301   1st Qu.:-0.76381
##  Median :-0.02031   Median : 0.2153   Median : 0.05847
##  Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.00000
##  3rd Qu.: 0.59195   3rd Qu.: 0.6303   3rd Qu.: 0.55383
##  Max.   : 1.86303   Max.   : 1.7300   Max.   : 1.59914
```

```
sapply(my_data_trans_2, sd)
```

```
##   col1_trans   col2_trans col2_trans.1
```

```
##             1             1             1
```

Der Code ist viel aufgeräumter und übersichtlicher. Es ist viel klarer, was berechnet wird.

7.2 Fallunterscheidungen

Manchmal möchte man unterschiedliche Berechnung innerhalb einer Funktion durchführen, je nach aufgestellter Bedingung. Daszu brauchen wir eine Fallunterscheidung, die durch einen `if-else` Ausdruck definiert wird. Jede Fallunterscheidung hat die folgende Form, wobei die zweite Bedingung nicht zwingend notwendig ist.

```
if (Bedingung1) {
  # Code, der ausgeführt wird, wenn die erste Bedingung1 stimmt
} else if (Bedingung2) {
  # Code, der ausgeführt wird, wenn zweite Bedingung2 stimmt
} else {
  # Code, der ausgeführt wird, wenn keine der Bedingungung stimmt
}
```

Achten Sie genau darauf, wie `else if` und `else` positioniert werden, nämlich zwischen den beiden geschweiften Klammern. Eine Fallunterscheidung kann auch mehr als zwei Bedingungen haben, aber man sollte nicht übertreiben. Wenn die Anzahl der Bedingungen zu hoch ist, sollte man nachdenken, ob das Problem nicht anders als mit `if-else` gelöst werden kann.

Wir schreiben eine Begrüßungsfunktion, die ja nach Tageszeit die richtige Begrüßung ausgibt.

```
say_hello <- function(my_time) {
  if (my_time == 'Morgen') {
    'Guten Morgen'
  } else if (my_time == 'Mittag') {
    'Guten Tag'
  } else if (my_time == 'Abend'){
    'Guten Abend'
  } else {
    'In meiner Welt gibt es das nicht.'
  }
}
```

Wir rufen die Funktion auf mit einmal mit Abend und einmal mit abend.

```
say_hello('Abend')
```

```
## [1] "Guten Abend"
```

```
say_hello('abend')
```

```
## [1] "In meiner Welt gibt es das nicht."
```

7.3 for-Schleifen (for *loops*)

Wenn wir unsere say_hello Funktion auf einen Vektor von Tageszeiten anwenden wollen, gibt es eine Warnung.

```
say_hello(c('Morgen', 'Mittag', 'Abend'))
```

```
## Warning in if (my_time == "Morgen") {: Bedingung hat Länge > 1 und nur das erste
## Element wird benutzt
```

```
## [1] "Guten Morgen"
```

Das liegt daran, dass die Auswertung der Bedingung eine Antwort der Länge 1 liefern muss: entweder TRUE, wenn die Bedingung stimmt, oder FALSE, wenn sie nicht stimmt. Bei einem Vektor ist die Antwort aber länger als 1 und die Funktion benutzt nur die erste Stelle der Antwort. Man sagt auch, dass die Funktion say_hello *nicht vektorisiert* sei, sie kann also nicht von sich aus auf einem Vektor arbeiten.

Damit man eine eine nicht vektorisierte Funktion auf einen Vektor anwenden kann, gibt es mehrere Möglichkeiten. Eine davon ist die sogen. for Schleife. Eine Schleife ist eine wiederholte Ausführung von Code, man sagt auch *Iteration*.

```
begruessung <- c('Morgen', 'Mittag', 'Abend')

# Vektor für Ergebnisse erstellen
ergebnis <- vector(mode = 'character', length = 3)

# For Schleife über die Zahlenfolge 1:3 mit Hilfe der Dummy-Variablen i
for (i in 1:3) {
  ergebnis[i] <- say_hello(begruessung[i])
}

# Ergebnis ansehen
ergebnis
```

```
## [1] "Guten Morgen" "Guten Tag"    "Guten Abend"
```

Eine for Schleife braucht drei Bestandteile:

1. Einen Datencontainer, in unserem Fall den Vektor ergebnis, in dem

die Ergebnisse der Schleifendurchläufe gespeichert werden. Dieser Vektor muss vorher erstellt werden.
2. Eine Dummy-Variable (Hilfsvariable), die in jedem Schleifendurchlauf einen anderen Wert annehmen wird. Dadurch entsteht erst die Schleife. Die Dummy-Variable bei uns heißt `i` und nimmt Werte zwischen 1 und 3 (also 1, 2 und 3) an. Es gibt also drei Schleifendurchgänge.
3. Den Schleifen-Körper, in unserem Fall die Funktion `say_hello`. Das ist der Code, der wiederholt ausgeführt werden soll.

Später werden wir effiziente Funktionen kennen lernen, die ohne Schleifen Funktionen wiederholt ausführen (iterieren) können.

7.4 Weiterführende Literatur

Ligges (2008), Kapitel 4.1 für technische Beschreibung des Aufrufs von Funktionen

Dieses Kapitel orientiert sich stark an Wickham and Grolemund (2017), Kapitel 19

7.5 Inhalt der live Einführung

- Funktionsaufruf
- Funktionen selbst schreiben
- Aufgabe 9.5.2: `if` - `else` Bedingungen und `for` Schleifen

Chapter 8

Tidyverse

- Kernpakete aus `tidyverse` benennen
- ein einfaches Workflow (Daten einlesen, zusammenfassen, darstellen) mit `tidyverse` durchführen
- Funktionen des Pakets `dplyr` für Datentransformation anwenden

`tidyverse` ist eine Sammlung von R-Pakete, die explizit für Datenanalyse entwickelt wurden (https://www.tidyverse.org/). `tidyverse` versucht durch gemeinsame Philosophie in Design, Grammatik und Datenstruktur die Datenanalyse zu erleichtern (https://design.tidyverse.org/). Auch wenn `tidyverse` auf den ersten Blick etwas fremd erscheint, es ist ein Teil von R, kein eigenes Universum. Es ist also völlig in Ordnung, R-Basisfunktionen mit Funktionen aus `tidyverse` zu mischen.

Das wichtigste Einführungsbuch zu `tidyverse` ist sicherlich **R4DS**: "R for Data Science" (Wickham and Grolemund, 2017), das Sie kostenlos online lesen können (https://r4ds.had.co.nz/).

8.1 Grundpakete

`tidyverse` enthält folgende Grundpakete, die alle installiert werden, wenn Sie `install.packages('tidyverse')` eingeben.

Paketname	Kurzbeschreibung
`ggplot2`	Visualisierung
`dplyr`	Datentransformation
`tidyr`	Datenbereinigung
`readr`	Daten einlesen
`purrr`	Funktionale Programmierung (Funktionen auf Objekte anwenden)
`tibble`	Erweiterung von `data.frame`

Paketname	Kurzbeschreibung
`stringr`	Funktionen für Strings, d.h. Textvariablen
`forcats`	Funktionen für `factor`

Jedes dieser Pakete hat ein Cheat Sheet, eine übersichtliche Zusammenstellung der Funktionen des Pakets. Sie bekommen die Cheet Sheats über die `tidyverse`-Seite (https://www.tidyverse.org/packages/), indem Sie auf das jeweilige Paket klicken und zum Abschnitt 'Cheatsheet' scrollen.

8.2 Der Workflow

8.2.1 Daten einlesen

Sie kennen bereits die sehr umfangreiche Funktion `read.table()` zum Einlesen der Daten. Die Funktion `read_delim()` ist die allgemeinste Funktion der `read_*` Familie aus `readr` in `tidyverse`; `read_csv()` und `read_csv2()` sind jeweils für komma- und strichpunkt-getrennte Datensätze gedacht. Man könnte berechtigterweise fragen, warum eine neue Funktion für etwas erfinden, was es schon gibt. Die Autoren von `tidyverse` versprechen Konsistenz und Geschwindigkeit. Ersteres war schon immer ein Problem von R, da es nicht von Computerspezialisten, sondern von Anwendern erfunden wurde. Daher ist eine Vereinheitlichung durch `tidyverse` mehr als willkommen. Und Geschwindigkeit ist spätestens bei größeren Datensätzen ein wichtiger Punkt.

Wir sehen uns erneut Daten des Deutschen Wetterdienstes an, wie im Kapitel 5. Nur diesmal sind es Stundenwerte für relative Luftfeuchte (%) und Lufttemperatur (°C). Die Daten von drei Wetterstationen, nämlich Hof, Frankfurt und Köln-Bonn, befinden sich in der Datei \texttt{Drei_Stationen.csv} Beim Einlesen zeigt Ihnen `read_delim()` bereits, welche Spalten und welche Datentypen es erkennt, mit `trim_ws = T` werden Leerzeichen aus Spalten entfernt.

```
library(tidyverse)

temp_humid <- read_delim('Drei_Stationen.csv', delim = ';',
                         trim_ws = T)
```

```
## Parsed with column specification:
## cols(
##   STATIONS_ID = col_double(),
##   MESS_DATUM = col_double(),
##   QN_9 = col_double(),
##   TT_TU = col_double(),
##   RF_TU = col_double(),
##   eor = col_character()
```

```
## )
```

Eine weitere Kontrolle bietet die Funktion `print()`, die das eingelesene Ergebnis übersichtlich (und im Notebook interaktiv) darstellt. Sie müssen hier nicht mehr `head()` verwenden, da grundsätzlich nur die ersten 10 Zeilen dargestellt werden.

```
print(temp_humid)
```

```
## # A tibble: 39,600 x 6
##    STATIONS_ID MESS_DATUM  QN_9 TT_TU RF_TU eor
##          <dbl>      <dbl> <dbl> <dbl> <dbl> <chr>
##  1        2261 2018111900     3  -2.8    99 eor
##  2        2261 2018111901     3  -2.5   100 eor
##  3        2261 2018111902     3  -2.3   100 eor
##  4        2261 2018111903     3  -2     100 eor
##  5        2261 2018111904     3  -1.9    99 eor
##  6        2261 2018111905     3  -2.1    99 eor
##  7        2261 2018111906     3  -1.8    99 eor
##  8        2261 2018111907     3  -1.5    99 eor
##  9        2261 2018111908     3  -1.1    99 eor
## 10        2261 2018111909     3  -0.6    97 eor
## # ... with 39,590 more rows
```

Das gleiche Ergebnis bekommen Sie auch ohne `print()`, wenn Sie wie gewohnt den Namen des Objekts tippen.

```
temp_humid
```

```
## # A tibble: 39,600 x 6
##    STATIONS_ID MESS_DATUM  QN_9 TT_TU RF_TU eor
##          <dbl>      <dbl> <dbl> <dbl> <dbl> <chr>
##  1        2261 2018111900     3  -2.8    99 eor
##  2        2261 2018111901     3  -2.5   100 eor
##  3        2261 2018111902     3  -2.3   100 eor
##  4        2261 2018111903     3  -2     100 eor
##  5        2261 2018111904     3  -1.9    99 eor
##  6        2261 2018111905     3  -2.1    99 eor
##  7        2261 2018111906     3  -1.8    99 eor
##  8        2261 2018111907     3  -1.5    99 eor
##  9        2261 2018111908     3  -1.1    99 eor
## 10        2261 2018111909     3  -0.6    97 eor
## # ... with 39,590 more rows
```

In diesem Datensatz sind folgende Parameter (Spalten) enthalten (s. Datensatzbeschreibung des DWDs)

Parameter	Beschreibung
STATIONS_ID	Stationsidentifikationsnummer
MESS_DATUM	Zeitstempel im Format yyyymmddhh
QN_9	Qualitätsniveau der nachfolgenden Spalten
TT_TU	Lufttemperatur in 2m Höhe °C
RF_TU	relative Feuchte %
eor	Ende data record

```
class(temp_humid)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

Das Objekt `temp_humid` ist ein sogen. Tibble, ein `data.frame` mit "modernem" Verhalten. Z.B. gibt die Funktion `print()` nur die ersten 10 Zeilen aus, die Datentypen in den Spalten werden in hellgrau zwischen '<>' mit angegeben etc. Mehr zu Tibbles finden Sie in Kapitel 10 "Tibbles" in R4DS.

Ein weiteres Paket, dass zwar nicht zum Kern von `tidyverse` gehört, jedoch trotzdem extrem nützlich ist, heißt `lubridate`. Es hilft, Text in richtige Datums-Objekte zu transformieren (ohne sich die kryptischen Datumsformate von R merken zu müssen). Wir transformieren die Spalte `temp_humid$MESS_DATUM` in ein richtiges Datum mit Uhrzeit.

```
library(lubridate)

temp_humid$MESS_DATUM <- ymd_h(temp_humid$MESS_DATUM)

print(temp_humid)
```

```
## # A tibble: 39,600 x 6
##    STATIONS_ID MESS_DATUM           QN_9 TT_TU RF_TU eor
##          <dbl> <dttm>              <dbl> <dbl> <dbl> <chr>
##  1        2261 2018-11-19 00:00:00     3  -2.8    99 eor
##  2        2261 2018-11-19 01:00:00     3  -2.5   100 eor
##  3        2261 2018-11-19 02:00:00     3  -2.3   100 eor
##  4        2261 2018-11-19 03:00:00     3  -2     100 eor
##  5        2261 2018-11-19 04:00:00     3  -1.9    99 eor
##  6        2261 2018-11-19 05:00:00     3  -2.1    99 eor
##  7        2261 2018-11-19 06:00:00     3  -1.8    99 eor
##  8        2261 2018-11-19 07:00:00     3  -1.5    99 eor
##  9        2261 2018-11-19 08:00:00     3  -1.1    99 eor
## 10        2261 2018-11-19 09:00:00     3  -0.6    97 eor
## # ... with 39,590 more rows
```

8.2.2 Daten zusammenfassen

Die drei Wetterstationen haben folgende IDs:

```
station_ids <-  tibble('Hof' = '2261', 'Frankfurt' = '1420', 'Koeln' = '2667')
```

Ich möchte wissen, wie viele Messpunkte es pro Station gibt:

```
temp_humid %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1 39600
```

Die Zeichenkombination `%>%` heißt Pipe-Operator (*pipe*) und wird als 'und dann' gelesen (*then*). Der Ausdruck `temp_humid %>% count()` heißt also: nimm `temp_humid` und dann zähle die Einträge. Der Pipe-Operator ist die Kernphilosophie von `tidyverse` und wird Ihnen überall begegnen. Der Operator stammt aus dem Paket `magrittr` (https://magrittr.tidyverse.org/). Seine Hauptaufgabe ist es, den Code übersichtlicher und besser lesbar zu machen (vielleicht nicht gleich zu Beginn der Lernkurve aber schon sehr bald).

Die Funktion `count()` gehört zum Paket `dplyr`, das für Datentransformationen zuständig ist. Dieses Paket enthält 5 Grundfunktionen (alle nach Verben benannt):

Funktion	Bedeutung
`filter()`	Wähle Daten anhand ihrere Werte
`arrange()`	Sortiere Zeilen
`select()`	Wähle Variablen anhand ihrere Namen
`mutate()`	Erstelle neue Variablen als Funktionen vorhandener Variablen
`summarize()`	Fasse Daten zusammen

Ich möchte nun wissen, wie viele Messpunkte es für Köln-Bonn gibt. `tidyverse` ist konsistent: jeder Aufruf einer Funktion aus `dplyr` gibt ein `tiblle` zurück (und nicht mal `data.frame` mal Vektor).

```
temp_humid %>%
  filter(STATIONS_ID == station_ids$Koeln) %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
```

```
##   <int>
## 1 13200
```

8.2.3 Daten plotten

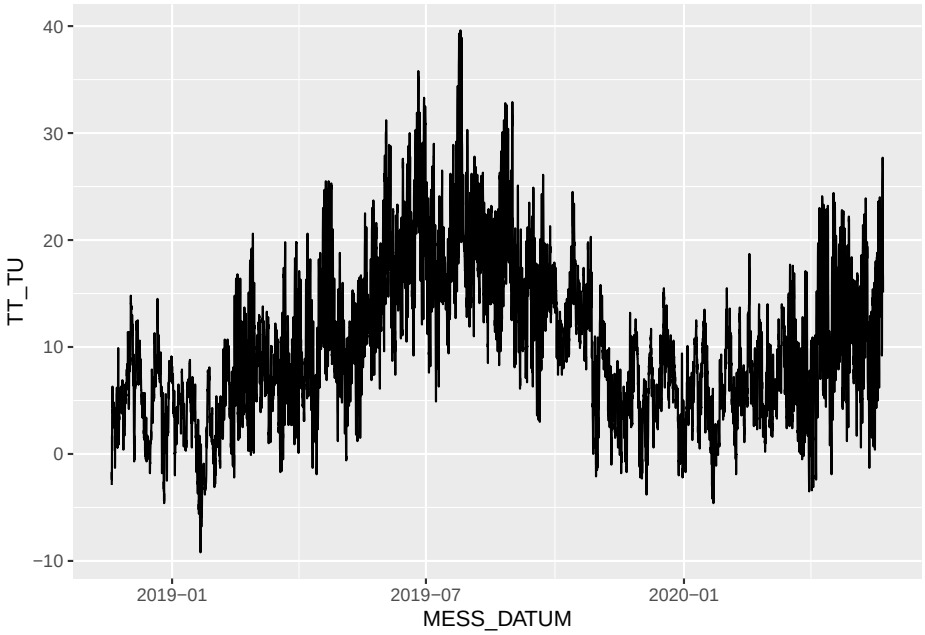
Wir wählen nur die Station Köln-Bonn und plotten die Stundenwerte der Temperatur mit dem Paket `ggplot2`.

```
# Köln-Bonn wählen
koeln <- temp_humid %>%
  filter(STATIONS_ID == station_ids$Koeln)

koeln
```

```
## # A tibble: 13,200 x 6
##    STATIONS_ID MESS_DATUM           QN_9 TT_TU RF_TU eor
##          <dbl> <dttm>              <dbl> <dbl> <dbl> <chr>
##  1        2667 2018-11-19 00:00:00     3  -2.9    69 eor
##  2        2667 2018-11-19 01:00:00     3  -1.7    78 eor
##  3        2667 2018-11-19 02:00:00     3  -2.4    82 eor
##  4        2667 2018-11-19 03:00:00     3   0.2    89 eor
##  5        2667 2018-11-19 04:00:00     3   1.7    84 eor
##  6        2667 2018-11-19 05:00:00     3   0.3    90 eor
##  7        2667 2018-11-19 06:00:00     3   3.4    84 eor
##  8        2667 2018-11-19 07:00:00     3   4.1    81 eor
##  9        2667 2018-11-19 08:00:00     3   3.9    80 eor
## 10        2667 2018-11-19 09:00:00     3   4.8    78 eor
## # ... with 13,190 more rows
```

```
# Plotten
ggplot(data = koeln, aes(x = MESS_DATUM, y = TT_TU)) +
  geom_line()
```



Sie werden sehr bald schon das Paket `ggplot2` nicht mehr missen wollen. Es erstellt mit wenig Aufwand sehr ansehnliche Grafiken. Natürlich müssen Sie, wie in Base-R auch, für professionelle Grafiken nacharbeiten.

`ggplot2` folgt einer sogen. Grafikgrammatik (*grammer of graphics*), indem es seine Grafiken mit '+' aufbaut. Erst sagt man in der Funktion `ggplot()` welchen Datensatz man plotten möchte, was auf die x- und y-Achse kommt und ob man z.B. Farbe für Gruppen in Daten möchte. All diese sichtbaren Elemente nennt man Ästhetiken (Parameter `aes`). Erst wenn diese 'Formalien' geklärt sind, sagt man, welche Form der Darstellung man möchte. Im oberen Beispiel ist es ein Linienplot. Alle Darstellungsarten in `ggplot2` beginnen mit `geom_*()` und sind daher konsistent benannt. Die Informationen zu Daten, x-, y-Achse etc. werden an die `geom_*()`-Funktionen von der Mutterfunktion `ggplot()` weiter vererbt. Daher sind die Klammern in `geom_line()` leer.

Wir wollen nun die Monatsmittelwerte für die Temperatur berechnen und diese darstellen. Als erstes erstellen wir zwei neue Spalten, die jeweils das Jahr und den Monat beinhalten. Die beiden neuen Spalten werden am Ende von `temp_humid` angehängt.

```
temp_humid <- mutate(temp_humid,
                     year = year(temp_humid$MESS_DATUM),
                     month = month(temp_humid$MESS_DATUM))

temp_humid
```

```
## # A tibble: 39,600 x 8
##    STATIONS_ID MESS_DATUM           QN_9 TT_TU RF_TU eor    year month
##          <dbl> <dttm>              <dbl> <dbl> <dbl> <chr> <dbl> <dbl>
##  1        2261 2018-11-19 00:00:00     3  -2.8    99 eor    2018    11
##  2        2261 2018-11-19 01:00:00     3  -2.5   100 eor    2018    11
##  3        2261 2018-11-19 02:00:00     3  -2.3   100 eor    2018    11
##  4        2261 2018-11-19 03:00:00     3  -2     100 eor    2018    11
##  5        2261 2018-11-19 04:00:00     3  -1.9    99 eor    2018    11
##  6        2261 2018-11-19 05:00:00     3  -2.1    99 eor    2018    11
##  7        2261 2018-11-19 06:00:00     3  -1.8    99 eor    2018    11
##  8        2261 2018-11-19 07:00:00     3  -1.5    99 eor    2018    11
##  9        2261 2018-11-19 08:00:00     3  -1.1    99 eor    2018    11
## 10        2261 2018-11-19 09:00:00     3  -0.6    97 eor    2018    11
## # ... with 39,590 more rows
```

Jetzt können wir einen neuen Datensatz mit den Mittelwerten erstellen. Dafür gruppieren wir erst einmal die Daten nach `STATIONS_ID`, `year` und `month`. Die Mittelwerte sollen ja je Station, Jahr und Monat berechnet werden.

```
by_month <- group_by(temp_humid, STATIONS_ID, year, month)
monthly_means <- summarize(by_month, mean_T = mean(TT_TU), mean_RH = mean(RF_TU))

monthly_means
```

```
## # A tibble: 57 x 5
## # Groups:   STATIONS_ID, year [9]
##    STATIONS_ID  year month mean_T mean_RH
##          <dbl> <dbl> <dbl>  <dbl>   <dbl>
##  1        1420  2018    11   4.00    79.7
##  2        1420  2018    12   4.73    83.7
##  3        1420  2019     1   2.12    79.3
##  4        1420  2019     2   4.48    74.1
##  5        1420  2019     3   8.28    68.5
##  6        1420  2019     4  11.7     61.0
##  7        1420  2019     5  12.7     67.5
##  8        1420  2019     6  21.4     60.6
##  9        1420  2019     7  21.6     55.6
## 10        1420  2019     8  20.7     65.6
## # ... with 47 more rows
```

Die Struktur von `monthly_means` zeigt uns, dass es sich um gruppierte Daten handelt.

```
str(monthly_means)
```

```
## Classes 'grouped_df', 'tbl_df', 'tbl' and 'data.frame':  57 obs. of  5 variables:
```

```
##  $ STATIONS_ID: num  1420 1420 1420 1420 1420 1420 1420 1420 1420 1420 ...
##  $ year       : num  2018 2018 2019 2019 2019 ...
##  $ month      : num  11 12 1 2 3 4 5 6 7 8 ...
##  $ mean_T     : num  4 4.73 2.12 4.48 8.28 ...
##  $ mean_RH    : num  79.7 83.7 79.3 74.1 68.5 ...
##  - attr(*, "groups")=Classes 'tbl_df', 'tbl' and 'data.frame':   9 obs. of  3 variables:
##   ..$ STATIONS_ID: num  1420 1420 1420 2261 2261 ...
##   ..$ year       : num  2018 2019 2020 2018 2019 ...
##   ..$ .rows      :List of 9
##   .. ..$ : int  1 2
##   .. ..$ : int  3 4 5 6 7 8 9 10 11 12 ...
##   .. ..$ : int  15 16 17 18 19
##   .. ..$ : int  20 21
##   .. ..$ : int  22 23 24 25 26 27 28 29 30 31 ...
##   .. ..$ : int  34 35 36 37 38
##   .. ..$ : int  39 40
##   .. ..$ : int  41 42 43 44 45 46 47 48 49 50 ...
##   .. ..$ : int  53 54 55 56 57
##   ..- attr(*, ".drop")= logi TRUE
```

Da wir aber mit den Daten weiter rechnen wollen, ist es besser, die Gruppierung wieder aufzugeben. Es könnte sonst später Fehlermeldungen geben.

```
monthly_means <- ungroup(monthly_means)
```

Um die Daten als Zeitreihen zu plotten, erstellen wir noch eine ordentliche Zeit-Spalte. Die Funktion `parse_date_time()` kann aus Character richtige Datums-Zeitobjekte erstellen. Sie ist allgemeienr als die oben verwendete `ymd()` Funktion, da man hier das Format explizit angeben kann. In unserem Fall ist das Format 'ym' für Jahr und Monat.

```
monthly_means <- monthly_means %>%
  mutate(my_date = parse_date_time(paste0(monthly_means$year,
                  monthly_means$month), 'ym', tz = 'CET'))
```

```
ggplot(data = monthly_means, aes(x = my_date, y = mean_T,
            col = factor(STATIONS_ID))) +
  geom_line() +
  labs(x = 'Time', y = 'Temperature (°C)', color = 'Station')
```

Wir werden in eine spätere Kapitel auf `ggplot2` genauer eingehen. Fürs Erste soll es als Appetithappen reichen .

8.3 Weiterführende Literatur und Videos

- R4DS: Kapitel 5 “Data transformation”, 11 “Data import”
- Eine live Analyse des Hauptautors von `tidyverse`, Hadley Wickham. Empfehlenswert, auch wenn er viel zu schnell tippt .

8.4 Inhalt der live Einführung

Verschiedene Funktionen aus `tydiverse` anhand der Aufgaben 9.6.1 und 9.6.2

Chapter 9

Aufgabensammlung

9.1 Erste Schritte

9.1.1 Ars Haushaltsbuch

Der angehende Datenanalyst Ar Stat möchte dem Rat seiner Mutter folgen und ein Haushaltsbuch anlegen. Als erstes möchte er sich einen Überblick über seine Ausgaben in der Uni-Mensa verschaffen und erstellt die folgende Tabelle:

1. Wie viel hat Ar insgesamt in der Woche ausgegeben?
2. Wie viel hat er im Schnitt pro Tag ausgegeben?
3. Wie stark schwanken seine Ausgaben?

Leider hat Ar sich beim übertragen der Daten vertippt. Er hat am Dienstag seine Freundin zum Essen eingeladen und 7,95 € statt 2,90 € ausgegeben.

4. Korrigieren Sie Ars Fehler.
5. Wie verändern sich die Ergebnisse aus den Teilaufgaben 1 bis 3 Warum?

Table 9.1: Ars Mensaausgaben

Wochentag	Ausgaben
Montag	2,57
Dienstag	2,90
Mittwoch	2,73
Donnerstag	3,23
Freitag	3,90

9.2 Daten in R

9.2.1 Bestandesaufnahme im Wald

Ar Stat arbeitet als HiWi in der AG Ökosystemforschung und soll im Nationalpark Eifel eine Bestandsaufnahme durchführen (d.h. Baumhöhen und -durchmesser vermessen). Er notiert den BHD (Brusthöhendurchmesser) und die Art der Bäume.

1. Lesen Sie den Datensatz `BHD.txt` ein und ordnen Sie ihn der Variable `BHD` zu.
2. Erstellen Sie einen Vektor `a` mit Baumnummern. Von welcher Art sind die Elemente des Vektors `a`?
3. Fügen Sie die Datensätze `BHD` und `a` zu einem `data.frame` zusammen und benennen Sie die Spalten sinnvoll.
4. Löschen Sie den Vektor `a`.
5. Lesen Sie den Datensatz `Art.txt` ein und ordnen Sie ihn der Variablen `art` zu.
6. Fügen Sie die Art in den `data.frame` ein.
7. Erstellen Sie eine Tabelle mit der Anzahl der jeweiligen Arten. Nutzen Sie die Funktion `table`.
8. Speichern Sie die Tabelle mit `write.table`.

9.3 Daten visualisieren, Teil I: Fokus auf R

9.3.1 Wahlbeteiligung bei der Bundestagswahl 2017

Bauen Sie die Grafiken aus der Einführung nach (Abbildung 9.1).

1. Lesen Sie den Datensatz `Wahlbeteiligung.csv` in R ein und ordnen Sie ihn dem Objekt `bet` zu. Der Datensatz hat einen *header* und haben einen Strichpunkt als Spaltentrenner.
2. Sehen Sie sich die Struktur und die ersten und letzten 6 Zeilen des Datensatzes an.
3. Stellen Sie die Wahlbeteiligung als Funktion der Zeit in einem Streudiagramm dar. Wählen Sie die passende Darstellungsform `type`.
4. Beschriften Sie die Grafik.
5. Speichern Sie die Grafik als pdf ab.

9.3.2 Zweitstimme bei der Bundestagswahl 2017

Bauen Sie die Grafiken aus der Einführung nach (Abbildung 9.2).

1. Lesen Sie den Datensatz `Zweitstimme.csv` in R ein und ordnen Sie ihn dem Objekt `zweit` zu. Der Datensatz hat einen *header* und haben einen Strichpunkt als Spaltentrenner.

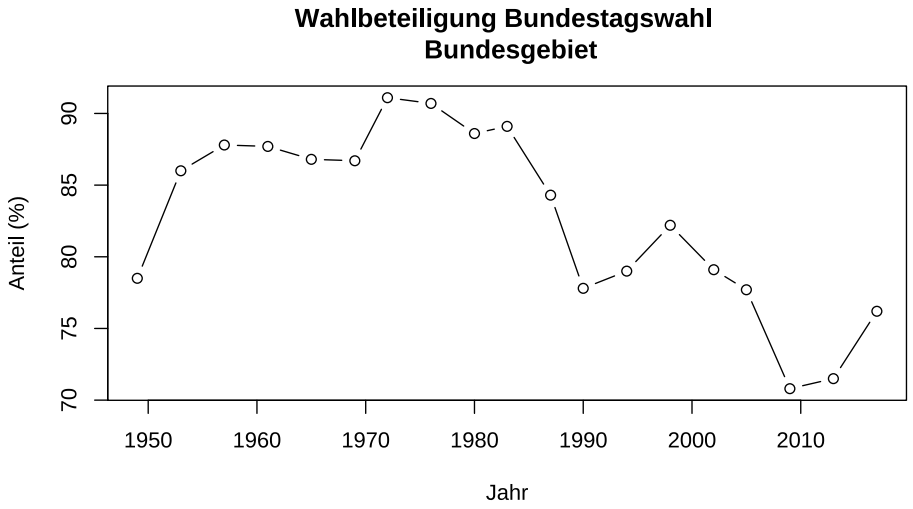


Figure 9.1: Wahlbeteiligung bei den Bundestagswahlen. Quelle: Der Bundeswahlleiter.

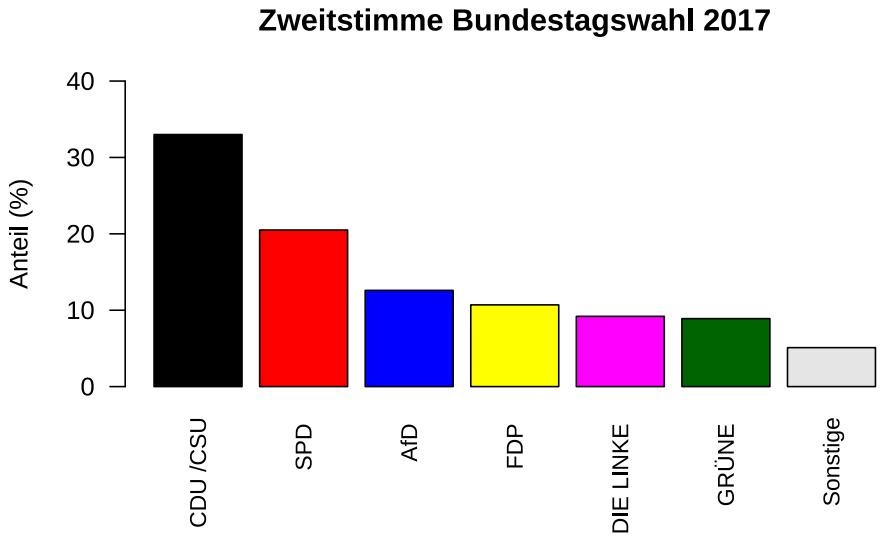


Figure 9.2: Zweitstimme bei der Bundestagswahl 2017. Quelle: Der Bundeswahlleiter.

2. Sehen Sie sich die Struktur und die ersten und letzten 6 Zeilen des Datensatzes an.
3. Stellen Sie die die Zweitstimmen pro Partei in einem Säulendiagramm dar. Sortieren Sie die Zweitstimmen in absteigender Reihenfolge.
4. Beschriften Sie die Grafik.
5. Speichern Sie die Grafik als pdf ab.

9.3.3 Ergebnisse der Bundestagswahl in einer Grafik

Stellen Sie beide Grafiken nebeneinander dar wie in Abbildung (9.3.3) gezeigt.

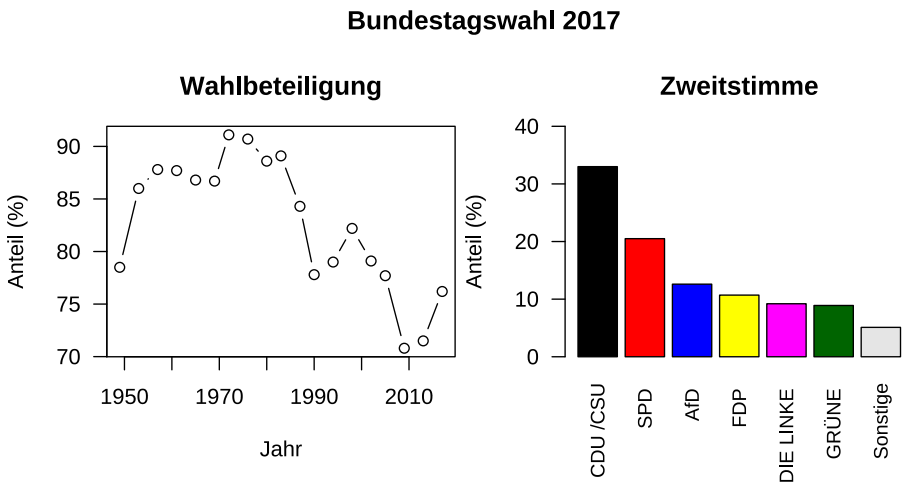


Figure 9.3: Ergebnisse der Bundestagswahl 2017. Quelle: Der Bundeswahlleiter.

9.3.4 Einen zu großen weißen Rand vermeiden

Bei Berichten haben Abbildungen meistens keine Überschrift, da alles in der Bildunterschrift erklärt wird. Wenn man die Überschrift beim plotten weglässt, die Standardeinstellungen für die Ränder aber beibehält, entsteht ein zu großer weißer Rand um die Grafik. Diesen wollen wir nun abschalten.

1. Kopieren Sie den Code zum Plotten der Temperaturen aus dem Kapitel 5.
2. Stellen Sie oben und rechts einen Rand von 0.1 Zeilen ein.
3. Speichern Sie die Grafik als pdf ab.

9.3.5 Spielen mit der Funktion `par`

Setzen Sie die Übung 9.3.4 fort. Denken Sie an den richtigen Aufruf mit der Zuweisung von `op <- par( ... )`!

1. Probieren Sie die Größeneinstellung `cex = 2` in `plot` aus. Testen Sie unterschiedliche Werte.
2. Probieren Sie die Einstellungen `cex.axis`, `cex.lab` und `cex.main` in `par` aus.
3. Probieren Sie die Einstellung `col` in `plot` aus.
4. Probieren Sie die Einstellungen `col.axis`, `col.lab` und `col.main` aus.
5. Probieren Sie die Schrifteinstellungen aus. Dazu stellen Sie den Parameter `family` in `par` auf "serif", "sans" oder "mono".
6. Probieren Sie die Parameter `font.lab = 2` und `font.axis = 2` direkt in `plot` aus. Zahlen 1 bis 5 stehen jeweils für normal, fett, kursiv, fett-kursiv und symbolisch.

9.4 Reproduzierbare Berichte mit R Markdown

9.4.1 Erster eigener Bericht

Erstellen Sie ein R Notebook aus den Notizen der ersten 3 R-Sessions.

9.5 Eigene Funktionen schreiben

9.5.1 R-Hausaufgaben

An dem Kurs "Einführung in R" nehmen 49 Studierende teil. Der Leistungsnachweis besteht aus Hausaufgaben, die insgesamt mit 100 Punkten bewertet werden. Ab 50 Punkten gilt der Kurs als bestanden.

1. Lesen Sie den Datensatz `R-HAs.txt`, der die Endpunkte enthält, ein.
2. Ermitteln Sie, wie viele Teilnehmer bestanden und wie viele nicht bestanden haben.

9.5.2 Fledermäuse, die Zweite

Wir beschäftigen uns erneut mit den Fledermäusen.

1. Lesen Sie den korrigierten(!) Datensatz \texttt{Fledermaus_cor.txt} ein.
2. Schreiben Sie eine Funktion, die den Entwicklungsstand der Tiere klassifiziert. Nutzen Sie dazu die ad hoc Regel: Individuum < 5 cm ist ein Jungtier, sonst erwachsen.
3. Erstellen Sie eine ordinal-skalierte Variable `alter` mit dem Entwicklungsstand der Tiere.
4. Wie viele Erwachsene und wie viele Jungtiere wurden vermessen?

9.6 Tidyverse

9.6.1 Fledermaus, die Dritte

1. Wiederholen Sie die Aufgabe 9.5.2 mit `tidyverse`
2. Berechnen Sie die Mittelwerte der Größe für weibliche und männliche Individuen.
3. Berechnen Sie die Mittelwerte der Größe für die Kategorien weiblich, männlich, Jungtier und erwachsen getrennt.

9.6.2 Unfaire Klausur?

Ar belegt im 4. Semester die Veranstaltung "Spaß mit R". Bei der Klausur gibt es 2 Aufgabengruppen mit jeweils 60 Punkten. Aufgabengruppe 1 wird an Studierende auf ungeraden Sitzplätzen und Aufgabengruppe 2 an Studierende auf geraden Sitzplätzen ausgegeben.

1. Lesen Sie den Datensatz `Klausurpunkte.txt` ein.
2. Überprüfen Sie Ars Vermutung, dass die Aufgabengruppe 1 im Schnitt leichter war als Aufgabengruppe 2 (d.h. in der Gruppe 1 im Schnitt mehr Punkte erzielt wurden).
   Berechnen Sie erst die Mittelwerte pro Gruppe.
   Testen Sie dann mit einem Permutationstest, ob die Mittelwerte signifikant verschieden sind. Adaptieren Sie den Code aus dem Tutorial Hypothesentests_1.Rmd.

9.7 Daten visualisieren, Teil II: Fokus auf Daten

9.7.1 Zeitreihen aus der Langen Bramke (Harz)

Im Harz wurden über eine längere Zeit Niederschlag, Abfluss und Temperatur gemessen.

1. Laden Sie den Datensatz `Data.dat`.
2. Stellen Sie die Temperatur in einem Streudiagramm dar. Welche Darstellungsart (Argument `type` in `plot`) erscheint Ihnen am sinnvollsten?
3. Beschriften Sie die Graphik und fügen Sie einen Titel hinzu.
4. Speichern Sie die Graphik als pdf ab.
5. Stellen Sie die Niederschläge in einem Diagramm dar. Wählen Sie einen geeigneten Darstellungstyp mit `type` (Tipp: geben Sie für die Hilfe `?plot` in die Konsole ein).

9.7.2 Temperatur-Datensatz

1. Laden Sie den Temperatur-Datensatz aus Zuur et al. (2009).
2. Berechnen Sie die Monatsmittelwerte für alle Stationen, sowie die Standardabweichungen.

3. Stellen Sie die Monatsmittel der Temperatur in einem Säulendiagramm dar.
4. Beschriften Sie die Graphik sinnvoll.
5. Fügen Sie die Standardabweichungen zu den einzelnen Balken hinzu.

9.7.3 Artenvielfalt in Grasländern

Sie erhalten Daten aus dem Grasland-Monitoring im Yellowstone Nationalpark und dem National Bison Range (USA). Das Ziel des Monitorings ist die Untersuchung möglicher Änderungen der Biodiversität und des Zusammenhang mit Umweltfaktoren. Biodiversität wurde durch die Anzahl unterschiedlicher Arten quantifiziert. Insgesamt haben die Forscher ca. 90 Arten in 8 Transekten kartiert. Die Aufnahmen wurden alle 4 bis 10 Jahre wiederholt. Insgesamt liegen 58 Beobachtungen vor. Die Daten sind in der Datei `Vegetation2.xls` gespeichert.

1. Laden Sie den Datensatz in R und sehen Sie sich das Ergebnis genau mit `str`, `head` und `tail` an. Diese Aufgabe dient dazu, das Einlesen von Excel-Dateien zu erarbeiten. Tipp: eine mögliche Bibliothek, die dabei helfen kann, wäre `xlsx`.

2. Berechnen Sie den Mittelwert und die Standardabweichung der Artenzahl (Variable `R`) pro Transekt.

3. Plotten Sie die Artenzahl gegen die Variable `BARESOIL` (Anteil von unbewachsenem Boden).

4. Benutzen Sie unterschiedliche Symbole pro Transekt, erstellen Sie eine Legende.

5. Beschriften Sie die Graphik sinnvoll und speichern Sie sie als pdf ab, ohne die Maus zu benutzen.

9.7.4 Tracerversuche

Im Waldstein wurden Tracerversuche mit dem Farbstoff Brilliant Blue durchgeführt und die gefärbten Bodenprofile *binärisiert* (d.h. in ein schwarz-weiß Bild umgewandelt). Schwarze Pixels stellen gefärbten Boden und weiße ungefärbten dar. Aus diesen Binärbildern wurde anschließend eine Reihe von Kenngrößen berechnet.

1. Lesen Sie die Datei \texttt{Waldstein2005_ind.txt} ein. Die Tiefe eines Profils ist 579 Pixel und es liegen 6 Profile untereinander in der Spalte d.
2. Berechnen Sie die 5%, 50% und 95% Quantile des Färbeanteils (Index d) der 6 Profile.
3. Stellen Sie den Median des Anteils der Färbung mit der Tiefe dar und fügen Sie die Quantile als transparente Fläche hinzu (Tipp: `polygon`).

9.8 Effizientes Programmieren

9.8.1 Lagerungsdichten

Auf 10 verschiedenen landwirtschaftlichen Feldern wurden im Oberboden je 25 Stechzylinder entnommen.

1. Lesen Sie den Datensatz `Bodendaten.txt` ein.
2. Bestimmen Sie die mittlere Lagerungsdichte pro Feld.

9.8.2 Temperatur-Datensatz, revisited

1. Laden Sie den Temperatur-Datensatz aus Zuur et al. (2009), Datei `Temperatur.csv`.
2. Berechnen Sie die Jahresmittelwerte je Station. (Tipp: Hilfe von `tapply` genau lesen!)

Bibliography

Ihaka, R. and Gentleman, R. (1996). R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314.

Knuth, D. E. (1984). Literate Programming. *The Computer Journal*, 27(2):97–111.

Ligges, U. (2008). *Programmieren mit R*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Murrell, P. (2006). *R Graphics*. Computer Science and Data Analysis Series. Chapman & Hall/CRC, Boca Raton, Fla. OCLC: 255097201.

Peng, R. D. (2019). *Report Writing for Data Science in R*.

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

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1138700109.

Xie, Y., Allaire, J., and Grolemund, G. (2018). *R Markdown: The Definitive Guide*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 9781138359338.

Zuur, A. F., Ieno, E., and Meesters, E. (2009). *A Beginner's Guide to R*. Springer.