Data Science für die Humangeographie: Ein pragmatischer Einstieg mit R

Projektseminar Konzeption quantitativer Forschung

true

Winter- und Sommersemester 2020/21

Inhaltsverzeichnis

Terminüberblick **4**

Online-Ressourcen **5**
R Tutorials und eBooks 5
Inspiration für Visualisierungen 6
Spezialthemen 6

1 Vorbesprechung **6**
1.1 Überblick 6
1.2 Seminarformat 7
1.3 Leistungsnachweise 8
1.4 Lehrphilosophie 12

2 Erste Schritte **13**
2.1 Vorbereitung 13
2.2 Lernziele für diese Sitzung 13
2.3 Operatoren 13
2.4 Variablen 14
2.5 Konstanten 14
2.6 Funktionen 15
2.7 Strings 16
2.8 Datentypen 16
2.9 Aufgaben 17

3 Text: Anderson 2008 **18**
3.1 Lesetext 18
3.2 Fragen an den Text 19

4 Datenstrukturen **19**

4.1 Lernziele dieser Sitzung . . . 19
4.2 Vektoren . . . 19
4.3 Matritzen . . . 21
4.4 Listen . . . 21
4.5 Data Frames . . . 22
4.6 Tibbles . . . 22
4.7 Aufgaben . . . 23

5 Visualisierungen **24**
5.1 Lernziele dieser Sitzung . . . 24
5.2 Voraussetzungen . . . 24
5.3 Überblick . . . 25
5.4 Visualisierung mit dem Standardpaket . . . 26
5.5 Visualisierung mit `ggplot()` . . . 28
5.6 Aufgaben . . . 33

6 Text: Shelton et al. 2014 **35**
6.1 Lesetext . . . 35
6.2 Fragen an den Text . . . 35

7 Geodaten **35**
7.1 Lernziele dieser Sitzung . . . 35
7.2 Voraussetzungen . . . 35
7.3 Exkurs: Pipes . . . 36
7.4 Daten importieren . . . 36
7.5 Überblick verschaffen . . . 37
7.6 Visualisieren . . . 37
7.7 Aufgaben . . . 40

8 Choroplethen **42**
8.1 Lernziele . . . 42
8.2 Vorbereitung . . . 42
8.3 Ziel . . . 42
8.4 Grundkarte . . . 42
8.5 OSM-Daten . . . 43
8.6 Koordinatenreferenzsysteme . . . 44
8.7 Verschneiden . . . 46
8.8 Aufgaben . . . 49

9 Text: Chandra 2014 **50**
9.1 Lesetext . . . 50
9.2 Fragen an den Text . . . 50
9.3 Themenfindung . . . 51

10 HTML-Tabellen **51**
10.1 Lernziele dieser Sitzung . . . 51
10.2 Vorbereitung . . . 51

10.3 Datenbeschaffung . . . 51
10.4 Datenformatierung . . . 53
10.5 Datenaufbereitung . . . 54
10.6 Datenvisualisierung . . . 55
10.7 Aufgaben . . . 56

11 Web scraping **58**
11.1 Lernziele dieser Sitzung . . . 58
11.2 Vorbereitung . . . 58
11.3 Exkurs: HTML . . . 58
11.4 Seite laden . . . 59
11.5 Elemente suchen . . . 60
11.6 Elemente reinigen . . . 61
11.7 Aufgaben . . . 62

12 Text: Straube 2021 **66**
12.1 Lesetexte . . . 66
12.2 Fragen an den Text . . . 66

13 Präsentationen **66**

14 APIs **66**
14.1 Vorbereitung . . . 67
14.2 SWAPI . . . 67
14.3 Exkurs: Funktionen schreiben . . . 68
14.4 Abfragefunktion . . . 69

15 Serialisierung **70**
15.1 Vorbereitung . . . 70
15.2 Zielsetzung . . . 70
15.3 URLs der Anzeigen auslesen . . . 70
15.4 Informationen der Anzeigen auslesen . . . 72
15.5 Aufbereiten . . . 75

16 Text: Breuer 2005 **76**

17 String manipulation **77**
17.1 Vorbereitung . . . 77
17.2 Aufgabe . . . 77
17.3 Tabellen aus Wikipedia laden . . . 77
17.4 Tabellen kombinieren . . . 77
17.5 Tabellen säubern . . . 80
17.6 Visualisierung . . . 83

18 Join und group **84**
18.1 Vorbereitung . . . 84
18.2 Aufgabe . . . 84

18.3 Daten einlesen . . . 84
18.4 Überblick verschaffen . . . 85
18.5 Zusammenfassen . . . 86
18.6 Verschneiden . . . 88
18.7 Kartieren . . . 89
18.8 Choroplethen . . . 91
18.9 Räumliches Verschneiden . . . 92

19 Rmarkdown und Kollaboration 93
19.1 Kollaboration . . . 93
19.2 Rmarkdown . . . 95

20 Text: Bowker und Star 1999 97

21 Interaktive Visualisierungen 98
21.1 Einleitende Bemerkungen . . . 98
21.2 Interaktive Karten mit `tmap` . . . 98
21.3 Interaktive Plots mit `plotly` . . . 102
21.4 Web apps mit `shiny` . . . 103

22 Clusteranalyse 103
22.1 Voraussetzungen . . . 103
22.2 Datensatz und Überblick . . . 103
22.3 Clusteranalyse . . . 105
22.4 Visualisierung . . . 108

23 Statistische Analyseverfahren 109

24 Text: Beer 2016 109

Terminüberblick

Alle Sitzungen finden von 13 bis 16h c.t. statt

Datum	Sitzung	Inhalt
2. November 2020	1	Vorbesprechung
9. November 2020	2	Erste Schritte
16. November 2020	3	Text: Anderson 2008
23. November 2020	4	Datenstrukturen
30. November 2020	5	Visualisierungen
7. Dezember 2020	6	Text: Shelton et al. 2014
14. Dezember 2020	7	Geodaten
11. Januar 2021	8	Choroplethen
18. Januar 2021	9	Text: Chandra 2014
25. Januar 2021	10	HTML-Tabellen
1. Februar 2021	11	Web scraping

Datum	Sitzung	Inhalt
8. Februar 2021	12	Text: Straube 2021
15. Februar 2021	13	Präsentationen
31. März 2021		*Abgabe Exposé*
12. April 2021	14	APIs
19. April 2021	15	Serialisierung
26. April 2021	16	Text: Breuer 2005
3. Mai 2021	17	String manipulation
10. Mai 2021	18	Join und group
17. Mai 2021	19	Rmarkdown und Kollaboration
24. Mai 2021		*Entfällt (Pfingstmontag)*
31. Mai 2021	20	Text: Bowker und Star 1999
7. Juni 2021	21	Interaktive Visualisierungen
14. Juni 2021	22	Clusteranalyse
21. Juni 2021	23	Statistische Analyseverfahren
28. Juni 2021	24	[Text: Beer 2017]
5. Juli 2021	25	Präsentationen
12. Juli 2021	26	Präsentationen

Online-Ressourcen

R Tutorials und eBooks

- **R for Data Science**
  https://r4ds.had.co.nz/
  Ausführliches Handbuch, Fokus auf Data Science
- **RStudio Cloud Primers**
  https://rstudio.cloud/learn/primers/1
- **Swirl**
  https://swirlstats.com/students.html
  Interaktives Tutorial als R-Paket, mit verschiedenen Lektionen
- **Quick-R**
  https://www.statmethods.net/r-tutorial/index.html
  Überblickartiges Tutorial, kurz und bündig
- **RStudio Cheat Sheets**
  https://www.rstudio.com/resources/cheatsheets/
  Einseitige Cheat Sheets zu verschiedenen Themen
- **Google's R Style Guide**
  https://google.github.io/styleguide/Rguide.xml
  Regeln für leserlichen R Code

Inspiration für Visualisierungen

- **R Graph Gallery**
  https://www.r-graph-gallery.com/
  Viele Beispiele für verschiedenste Visualisierungen
- **DDJ Katalog**
  http://katalog.datenjournalismus.net/#/
  Portfolio Datenjournalismus, leider etwas veraltet
- **Subreddits**
  https://www.reddit.com/r/dataisbeautiful
  https://www.reddit.com/r/DataArt/
  https://www.reddit.com/r/MapPorn/
- **Infographics**
  https://www.listendata.com/2019/06/create-infographics-with-r.html
- **HTML Widgets für R** http://gallery.htmlwidgets.org/

Spezialthemen

- **HTML-Überblick**
  https://www.tutorialspoint.com/de/html/
- **Tutorial Reguläre Ausdrücke**
  https://danielfett.de/en/tutorials/tutorial-regulare-ausdrucke/
  Deutschsprachige Einführung zu regulären Ausdrücken
- **Reguläre Ausdrücke testen**
  https://www.regexpal.com/
  Online-Sandkasten um reguläre Ausdrücke auszuproblieren
- **Übersicht CSS-Selektoren**
  https://www.w3schools.com/cssref/css_selectors.asp

1 Vorbesprechung

1.1 Überblick

1.1.1 Seminar im Curriculum

- Dieses Seminar ist Bestandteil des Moduls BA3.
- Das Projektseminar besteht aus zwei Teilen über zwei Semester:
  - Konzeption quantitativer Forschung (Wintersemester)
  - Analyse quantitativer Daten (Sommersemester)
- Im Winter gibt es 12 inhaltliche Termine (davon 4x Textarbeit).
- Im Sommer wird das Seminar mit den selben Teilnehmer*innen fortgeführt.

1.1.2 Lernziele für das Wintersemester

Sie können…

- einfache Skripte in R eigenständig erstellen.
- Datensätze in vielfältigen Formaten visualisieren.
- Online-Ressourcen gezielt einsetzen.
- Möglichkeiten der Datenbeschaffung identifizieren.
- epistemologische Verschiebungen durch Data Science wiedergeben.

1.1.3 Technische Anforderungen

- Es sind keine Vorkenntnisse in R erforderlich.
- Sie brauchen einen Laptop, mit dem Sie gut arbeiten können.
- Wir benutzen die RStudio Cloud als Plattform.
- Sie brauchen einen ruhigen Arbeitsplatz.

1.1.4 Unterstützung im Corona-Semester

- Die Uni bietet einen "Semesterlaptop" an.
- Bei Bedarf kann ich gerne versuchen, Arbeitsplätze im Seminarraum (PEG) anzubieten.
- Bitte kontaktieren Sie mich per E-Mail, falls Sie einen Arbeitsplatz regelmäßig in Anspruch nehmen wollen würden.

1.2 Seminarformat

- Das Seminar findet jede Woche Montags, 13–16h c.t. statt.
- Der Zoom-Link, den Sie per E-Mail erhalten haben, bleibt gleich.
- Wir machen um ca. 14:25h eine zehnminütige Pause.
- Für Textbesprechungen wird die Gruppe zweigeteilt.
- Dieses Seminar findet in verschiedenen Modi statt:

1.2.1 Input und Plenum

- Ich rede oder moderiere (mit Folien oder ohne)
- Sie hören mir und Ihren Kommiliton*innen aufmerksam zu
- Sie "melden" sich für Redebeiträge oder Fragen (Zoom-Funktion)
- Die*der Chat-Verantwortliche unterbricht mich bei Klärungsbedarf

1.2.2 Think-pair-share

- Sie bearbeiten eine Fragestellung in zufälligen Zweier-Konstellationen (Breakout-Session)
- Nach einer vorgegebenen Zeitspanne kehren Sie ins Plenum zurück
- Ich fordere Sie ggf. auf, Ergebnisse und offene Fragen mit der Gruppe zu teilen

1.2.3 Follow the recipe

- Ich teile ein unvollständiges Beispielprojekt.
- Wir gehen die Teilschritte nach und nach durch.
- Ich "habe den Plan", stelle aber immer wieder Fragen ans Plenum.
- Sie vollziehen die Schritte an Ihrer eigenen Kopie des Projekts nach.
- Die*der Chat-Verantwortliche unterbricht mich bei Klärungsbedarf

1.2.4 Hands-on session

- Sie bearbeiten praktische Aufgabenstellungen alleine.
- Dabei sind sie in zufälligen Dreier-Konstellationen (Breakout-Session).
- Bei Fragen oder Problemen wenden Sie sich zunächst an Ihre Kleingruppe.
- Falls Sie nicht weiterkommen, fordern Sie Hilfe an (Zoom-Funktion).
- Ich reagiere auf Hilfegesuche oder schaue in zufälligen Gruppen vorbei.

1.2.5 Share your work

- Ich wähle eine Teilnehmer*in zufällig aus.
- Die Person teilt ihren Bildschirm und berichtet von ihrer Bearbeitung eines Problems.
- Alle anderen unterstützen solidarisch durch aktives Nachvollziehen, Nachfragen und Hinweise.

1.3 Leistungsnachweise

1.3.1 Exposé (WiSe)

- Zum Ende des Wintersemesters geben Sie ein Exposé für ein Untersuchungsvorhaben für das Sommersemester ab.
- Sie können sich mit bis zu vier Personen zusammenschließen.
- Die Projektgruppe besteht dann verbindlich für das Sommersemester.
- Damit steigen aber auch die Anforderungen an Umfang, Detail und technischen Anspruch.
- Umfang für das Exposé: max. 15k Zeichen inkl. Leerzeichen, exkl. Literaturverzeichnis
- Als Abgabetermin haben wir den 31. März vereinbart.

1.3.1.1 Inhalte

- Einführung ins Thema
- Forschungsstand / Literaturüberblick
- Herleitung einer klar abgegrenzten (vorläufigen) Forschungsfrage
- Konkrete Datenquellen
- Ideen für Verfahren und Visualisierungen

1.3.1.2 Bewertungskriterien Alle Kriterien werden mit einer (runden) Schulnote bewertet. Der gewichtete Schnitt ergibt die Gesamtnote.

Kriterium	Gewichtung	Erläuterung
Ziterweise und Formatierung	10%	Der Text erfüllt formale Anforderungen an Wissenschaftlichkeit.
Ausdruck und Rechtschreibung	10%	Der Text ist sprachlich gelungen; Abbildungen und Tabellen werden ggf. hilfreich eingesetzt.
Roter Faden	10%	Der Text ist übersichtlich strukturiert und die Einzelteile greifen gut ineinander.
Recherche	10%	Die zitierten Quellen sind für eine Einführung ins Thema geeignet und werden gut zusammengefasst.
Theorie	10%	Relevante wissenschaftliche Perspektiven werden anhand von geeigneter Fachliteratur aufgezeigt.
Fragestellung	10%	Die Forschungsfrage ist für das Vorhaben geeignet und wird überzeugend hergeleitet.
Datenquellen	20%	Die Datenquellen sind geeignet und detailliert beschrieben.
Design	20%	Das Untersuchungsvorhaben ist nachvollziehbar beschrieben, und der technische Anspruch ist dem Projektseminar angemessen.

1.3.2 Projektbericht (SoSe)

- Zum Ende des Sommersemesters geben Sie einen Projektbericht ab.
- Der Projektbericht darf (überarbeitete) Teile des Exposés enthalten.
- Die Gruppen, in denen Sie Ihr Exposé verfasst haben, bleiben verbindlich bestehen.
- Umfang für den Projektbericht:
  - max. 35k Zeichen
  - inkl. Leerzeichen
  - exkl. Literaturverzeichnis
  - exkl. Code
- Als Abgabetermin haben wir den 31. August vereinbart.

1.3.2.1 Format

- Der Projektbericht muss in Rmarkdown verfasst werden und (grundsätzlich) ausführbar sein.
  - Daten, die für die Ausführung benötigt werden, müssen mit abgegeben werden. (Bei sehr großen Datensätzen wenden Sie sich bitte frühzeitig an mich).
  - Dabei dürfen alle Pakete aus CRAN verwendet werden. Eigene Scripts müssen mit abgegeben werden.

  - Aufwändige oder prekäre Zwischenschritte (etwa Web Scraping, rechenintensive Grafiken) bitte als Zwischenergebnis speichern. (Code trotzdem darlegen!)

1.3.2.2 Inhalte

- Einführung ins Thema
- Forschungsstand / Literaturüberblick
- Herleitung einer klar abgegrenzten Forschungsfrage
- Besprechung der Datenquellen
- Darstellung der Methoden für
  - Datenbeschaffung
  - Datenaufbereitung (säubern, verschneiden)
  - Datenanalyse (Visualisierung, statische Verfahren)
- Präsentation der Ergebnisse
- Einordnung der Ergebnisse

1.3.2.3 Bewertungskriterien Alle Kriterien werden mit einer (runden) Schulnote bewertet. Der gewichtete Schnitt ergibt die Gesamtnote.

Kriterium	Gewichtung	Erläuterung
Ziterweise und Formatierung	10%	Der Text erfüllt formale Anforderungen an Wissenschaftlichkeit.
Ausdruck und Rechtschreibung	10%	Der Text ist sprachlich gelungen.
Roter Faden	10%	Der Text ist übersichtlich strukturiert und die Einzelteile greifen gut ineinander.
Literatur	5%	Die zitierten Quellen sind für eine Einführung ins Thema geeignet und werden gut zusammengefasst.
Theorie	10%	Relevante wissenschaftliche Perspektiven werden anhand von geeigneter Fachliteratur aufgezeigt.
Fragestellung	5%	Die Forschungsfrage ist für das Vorhaben geeignet und wird überzeugend hergeleitet.
Anspruch	20%	Der technische Anspruch ist dem Projektseminar angemessen.
Beschaffung	10%	Die Datenquellen sind geeignet und werden nachvollziehbar ausgelesen.
Aufbereitung	10%	Die Datenaufbereitung ist sauber durchgeführt und gut nachvollziehbar.
Analyse	10%	Die Datenanalyse ist geeignet, sauber durchgeführt und anschaulich beschrieben.

1.3.3 Anwesenheit

- Es besteht Anwesenheitspflicht.
- Für Ihre ersten zwei Fehltermine pro Semester brauche ich keine Entschuldigung (aber Sie sollten das ggf. mit ihrer Projektgruppe absprechen).
- Sie sind dann selbstständig für die Nacharbeit der behandelten Themen zuständig.
- Im Falle eines zusätzlichen Fehltermins brauche ich ein Attest und einen Nachweis über Nacharbeit.
- Zur Anwesenheit gehört…
  - uneingeschränkte Aufmerksamkeit über die komplette Veranstaltungsdauer,
  - aktive Mitarbeit an Beispielen,
  - Bearbeitung von Übungsaufgaben,
  - aktive Beteiligung an Diskussionen.
- Eine eingeschaltete Kamera macht das allen Beteiligten leichter!

1.4 Lehrphilosophie

- Die folgenden vier "Säulen" habe ich mal im Rahmen einer Fortbildung als meine "Lehrphilosophie" definiert.
- Sie spiegeln meinen eigenen Anspruch an meine Lehre wider und sind auch als Vorschlag für ein gutes Miteinander zu verstehen.
- Begreifen Sie die hier genannten Aspekte gerne auch als Ermunterung, sie von mir und Ihren Kommiliton*innen einzufordern, wenn sie in der Veranstaltung zu kurz kommen.

1.4.1 Transparenz

- Erforderliche Leistungen und Bewertungskriterien sind vorab bekannt.
- Termine und Regelungen werden in der Vorbereitungssitzung verbindlich vereinbart.
- Aktuelle Lehrmaterialien stehen online durchgängig zur Verfügung.

1.4.2 Praktische Übungen

- Eigenständige Anwendung steht im Vordergrund.
- Verfahren und Techniken werden mit Beispielen und Übungen erarbeitet.
- Die perfekte Aufgabe ist immer ein bisschen "zu schwer".
- Toleranz für Frustration ist eine wichtige Fähigkeit und lässt sich trainieren.

1.4.3 Geschützte Räume

- Alle können sich im Plenum respektiert und sicher fühlen. Verletzendes Verhalten wird benannt.
- Es gibt einen vertrauensvollen Rahmen für ehrlichen Austausch.

- Frustrationen und Momente des Scheiterns werden ernst genommen und konstruktiv bearbeitet.

1.4.4 Kritische Reflexion

- Auch Teilnehmende, die kein weiterführendes Interesse an der Anwendung quantitativer Verfahren haben, sind im Seminar gut aufgehoben.
- Verfahren werden kontextualisiert, ihre Limitationen werden aufgezeigt.
- Kritische Forschung zu quantitativen Praktiken wird besprochen.

2 Erste Schritte

2.1 Vorbereitung

- Machen Sie sich einen kostenlosen Account auf https://rstudio.cloud
- Treten Sie dem Seminar-Workspace bei. (Sie erhalten eine Einladung per E-Mail.)
- Optional/alternativ: installieren Sie R und RStudio auf Ihrem Computer.

2.2 Lernziele für diese Sitzung

Sie können…

- Rechenoperatoren einsetzen.
- Variablen zuweisen.
- Funktionen aufrufen.
- Hilfe zu Funktionen anzeigen.
- die wichtigsten Variablentypen bestimmen.
- zwischen Variablentypen konvertieren.

2.3 Operatoren

Zunächst stellen wir fest, dass man die R-Konsole ganz banal als Taschenrechner benutzen kann:

```
1 + 4
## [1] 5
8 / 3
## [1] 2.666667
(2.45 + 3.5) * 7
## [1] 41.65
```

Die Zeichen `+`, `-`, `*` usw. heißen in der Informatik Operatoren oder Infixe (weil sie immer zwischen zwei Werten stehen).

2.4 Variablen

Variablen funktionieren so, dass man einem *Wert* einen Namen gibt. Die Zuweisung folgt dabei dem Schema `NAME <- WERT`:

```
x <- 5
```

Nach einer erfolgreichen Variablenzuweisung gibt die Konsole *keine* Rückmeldung, sondern nur bei Fehlern.

`x` steht jetzt für die Zahl fünf. Mit dieser Variable können wir jetzt genauso rechnen wie mit einer Zahl:

```
x + 3
## [1] 8
```

Auch die Zuweisung von Variablen kann Rechenoperationen und andere Variablen enthalten:

```
y <- (x * 2) - 1
print(y)
## [1] 9
```

Der Befehl `print(y)` ist dabei ganz einfach die Anweisung an die Konsole, den Wert für `y` auszugeben. Das passiert zwar auch, wenn man nur `y` eingibt, aber `print(y)` (oder `print(x)`, `print(1 + 1)`, usw.) ist die formal korrekte Schreibweise.

Der Wert einer Variable kann auch verändert werden. Dafür weisen wir ihr einfach einen neuen Wert zu:

```
x <- 20
print(x)
## [1] 20
```

Eine Besonderheit ist, dass der alte Wert der Variable auch innerhalb der Zuweisung eines neuen Werts benutzt werden darf. Das kann in einem Script sehr praktisch sein. Wenn wir `x` also um 0,5 erhöhen wollen, sieht das so aus:

```
x <- x + 0.5
print(x)
## [1] 20.5
```

Dabei wird als Dezimaltrennzeichen ausschließlich der Punkt verwendet.

2.5 Konstanten

Manche benannten Werte sind schon in R eingebaut:

```
print(pi)
## [1] 3
```

Diese Werte heißen üblicherweise "Konstanten" – allerdings lassen sie sich in R auch überschreiben!

```
pi <- 3
print(pi)
## [1] 3
```

2.6 Funktionen

Mit `print()` haben wir schon unsere erste *Funktion* kennengelernt. R stellt uns eine Vielzahl von verschiedenen Funktionen zur Verfügung, und sie werden immer nach dem gleichen Schema benutzt: `FUNKTIONSNAME(PARAMETER)`.

Parameter (auf Englisch auch "arguments") sind die Werte, die als Input an die Funktion übergeben werden. Je nach Funktion können das auch mehrere Werte sein, die dann durch Kommas getrennt werden. So nimmt die Funktion `max()`, die den Maximalwert bestimmt, beliebig viele Zahlen als Parameter:

```
max(1, 2, 2, 5, 4, 3)
## [1] 5
```

Die Funktion `round()` hat als optionalen Parameter die Anzahl der Nachkommastellen, auf die gerundet werden soll. Wenn er nicht angegeben wird, nimmmt dieser Parameter immer den Wert 0 an:

```
round(4.567)
## [1] 5
```

Aber er lässt sich auch spezifizieren:

```
round(4.567, digits = 2)
## [1] 4.57
```

Dabei sind die folgenden Ausdrücke identisch:

```
round(4.567, digits = 2)
## [1] 4.57
round(4.567, 2)
## [1] 4.57
round(digits = 2, 4.567)
## [1] 4.57
```

Was Funktionen genau machen und welche Parameter sie dabei nehmen, ist in der R-Dokumentation sehr ausführlich (und auf den ersten Blick recht kompliziert) beschrieben. Ganz am Ende der Hilfeseite finden sich oft Beipsiele. Die Hilfe zu einer Funktion kann mit folgendem Befehl aufgerufen werden:

```
?max
```

Notiz am Rande: Auch die Infix-Operatoren `+`, `-`, `*`, usw. sind eigentlich nur verkürzte Schreibweisen von Funktionen. Mit "backticks" (‘) lassen sie sich in

vollwertige Funktionen zurückverwandeln:

```
`+`(2, 2)
## [1] 4
```

2.7 Strings

R kann nicht nur mit Zahlen umgehen, sondern auch mit Text. Ein *String* ist eine Aneinanderreihung von Buchstaben, und wird mit einfachen oder doppelten Anführungszeichen umschlossen:

```
print("Hello, World!")
## [1] "Hello, World!"
```

Auch Variablen können Strings als Wert haben:

```
name <- "Hase"
```

Es gibt auch Funktionen, die Strings als Parameter nehmen. `paste` fügt Strings aneinander:

```
paste("Mein Name ist", name)
## [1] "Mein Name ist Hase"
```

2.8 Datentypen

Den *Typ* einer Variable oder eines Wertes bestimmen wir durch den Befehl `str()`:

```
str(name)
##  chr "Hase"
str(10)
##  num 10
```

Dabei steht `chr` („character“) für Strings und `num` („numeric“) für Zahlen.

Ein weiterer Variablentyp ist `logi` („logical“), der prinzipiell nur die Werte `TRUE` oder `FALSE` annehmen kann. Dieser Typ heißt auch Boolsche Variabel:

```
str(FALSE)
##  logi FALSE
```

Soweit es ein eindeutiges Ergebnis gibt, kann R mit den entsprechenden Befehlen Werte vom einen in den anderen Typ umwandeln:

```
as.numeric("1000")
## [1] 1000
as.character(x)
## [1] "20.5"
as.logical(0)
## [1] FALSE
```

Kann R einen Wert nicht umwandeln, dann kommt dabei `NA` raus (mit einer Warnung):

```
as.numeric("Hallo!")
## [1] NA
```

`NA` („not available/assigned“) ist dabei ein besonderer Wert, den jeder Variablentyp annehmen kann.

2.9 Aufgaben

2.9.1 Rechnen

Lösen Sie folgende Rechenaufgaben mit Hilfe von R:

- 4 plus 10
- 8 mal 12
- 4 minus 7
- 3 hoch 18
- 4,5 geteilt durch die Summe von 5 und 8
- Quadratwurzel aus 101
- Kubikwurzel aus 12

2.9.2 Variablen

Weisen Sie den Variablen a bis g folgende Werte zu:

a) TRUE
b) 2
c) Ihren Namen
d) Die Quadratwurzel aus b
e) 8 $\frac{1}{4}$
f) Das vierfache von e
g) Die aktuelle Uhrzeit mit Datum und Zeitzone (automatisch generiert)

2.9.3 Datentypen

Bestimmen Sie die Typen der Variablen a bis g.

Finden Sie je zwei Beispiele für die Umwandlung…

- von `numeric` zu `character`
- von `numeric` zu `logical`
- von `character` zu `logical`
- von `character` zu `numeric`
- von `logical` zu `character`
- von `logical` zu `numeric`
- von `character` zu `Date`
- von `Date` zu `numeric`

(`Date` ist kein eigentlicher Datentyp, aber erfüllt an dieser Stelle denselben Zweck.)

2.9.4 Swirl

Folgen Sie den Anleitungen, um Swirl zu installieren: https://swirlstats.com/students.html

Absolvieren Sie Lektion 1 („Basic Building Blocks“).

2.9.5 Recherche

Recherchieren Sie:

- Welche Funktion gibt den absoluten Wert einer Zahl aus? (z.B. -4 ergibt 4, 8 ergibt 8)
- Welche Konstanten sind in R „eingebaut“?
- Wie bestimmt man den „Rest“ einer Division? (z.B. 40 geteilt durch 7 hat den Rest 5)
- In der Statistik wird zwischen stetigen und diskreten Variablen unterschieden. Welche äquivalente Unterscheidung nimmt R vor?

2.9.6 Kniffliges

Lösen Sie die folgenden Probleme:

- Durch welchen Ausdruck lässt sich eine Zahl auf die nächste *gerade* Zahl runden? (z.B. 18,9 auf 18,0 oder 21,2 auf 22,0)
- Durch welchen Ausdruck lässt sich eine Zahl auf die nächste *halbe* Zahl abrunden? (z.B. 18,9 auf 18,5 oder 21,2 auf 21,0)
- Absolvieren Sie in die Lektion 8 („Logic“).
- Machen Sie sich mit der Funktion `xor()` vertraut. Finden Sie einen Ausdruck, der `xor()` simuliert, aber nur aus Infix-Operatoren besteht.
- Was bedeutet „strong“ bzw „weak typing“? Wie ist R hier einzuordnen?
- Was sind funktionale Programmiersprachen? Welche Eigenschaften von R sind funktional, welche nicht?
- Starten Sie den R Track in Excercism
- Richten Sie sich ein IDE *außer* RStudio für einen R Workflow ein.

3 Text: Anderson 2008

3.1 Lesetext

Anderson, Chris. 2008. *The End of Theory: The Data Deluge Makes the Scientific Method Obsolete.* URL: https://www.wired.com/2008/06/pb-theory/ (zugegriffen: 11. Juli 2017).

3.2 Fragen an den Text

1. Um welche Art von Text handelt es sich? Wer ist der Autor, und an wen wendet er sich?
2. Was ist das zentrale Anliegen des Texts? Welche Entwicklungen werden beschrieben?
3. Mit welchen Begriffen würden wir diese Phänomene heute beschreiben?
4. Aus heutiger Perspektive: Hatte der Autor recht? Warum / warum nicht?
5. In welchen Punkten stimmen Sie dem Autor zu? Wie würden Sie den Text problematisieren?

4 Datenstrukturen

4.1 Lernziele dieser Sitzung

Sie können…

- die verschiedenen Strukturen für Datensätze in R benennen.
- Vektoren generieren.
- einfache Befehle mit Vektoren durchführen.
- Beispieldatensätze aufrufen und beschreiben.

4.2 Vektoren

Vektoren (engl. *vectors*) sind eindimensionale Reihen von Werten gleichen Typs. Sie bilden einen wichtigen Baustein von R und von den hier im Seminar besprochenen Inhalten.

Sie können manuell mit der Funktion `c(...)` erstellt werden und wie Variablen benannt werden:

```
alter <- c(39, 49, 63, 44, 40)
alter
## [1] 39 49 63 44 40
```

Es gibt darüber hinaus aber auch Möglichkeiten, Vektoren automatisch zu generieren:

```
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
seq(100, 10, by=-10)
##  [1] 100  90  80  70  60  50  40  30  20  10
```

Buchstaben sind als Vektor in R eingebaut:

```
letters
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

Manche Funktionen sind speziell für Vektoren gedacht:

```
rev(1:10)
##  [1] 10  9  8  7  6  5  4  3  2  1
```

Andere Funktionen, die für einzelne Werte gedacht sind, werden für jeden Wert einzeln ausgeführt:

```
toupper("hallo")
## [1] "HALLO"
toupper(c("ein", "paar", "strings"))
## [1] "EIN"     "PAAR"    "STRINGS"
```

Elemente von Vektoren können mit eckigen Klammern einzeln oder selektiv angesprochen bzw entfernt werden:

```
letters[2]
## [1] "b"
letters[2:3]
## [1] "b" "c"
letters[-2]
##  [1] "a" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
## [20] "u" "v" "w" "x" "y" "z"
```

Vektoren können wie Variablen benutzt werden:

```
2018 - alter
## [1] 1979 1969 1955 1974 1978
paste(alter, "ist ein gutes Alter")
## [1] "39 ist ein gutes Alter" "49 ist ein gutes Alter" "63 ist ein gutes Alter"
## [4] "44 ist ein gutes Alter" "40 ist ein gutes Alter"
```

`length(x)` gibt die Anzahl der Elemente in einem Vektor `x` aus:

```
length(alter)
## [1] 5
```

Von Verteilungen, die als Vektoren vorliegen, lassen sich statistische Parameter einfach errechnen:

```
mean(alter)
## [1] 47
median(alter)
## [1] 44
sd(alter)
## [1] 9.77241
IQR(alter)
```

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

(Aber `IQR()` berechnet anders als in der Vorlesung besprochen!)

Wir können den Mittelwert auch mit Hilfe der `sum()` und `length()` Funktionen selbst berechnen:

```
sum(alter) / length(alter)
## [1] 47
```

4.3 Matritzen

Matritzen (engl. *matrix*) sind zweidimensionale Reihen von Werten gleichen Typs.

```
matrix(1:15, nrow=3)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
```

Sie spielen in diesem Seminar aber keine große Rolle.

4.4 Listen

Listen sind eindimensionale Reihen von Werten, wobei der Typ egal ist:

```
list("Hallo", 10, F)
## [[1]]
## [1] "Hallo"
##
## [[2]]
## [1] 10
##
## [[3]]
## [1] FALSE
```

Dabei können die Werte benannt sein, und Listen können Unterlisten enthalten:

```
profil <- list(name="Till", plz=60326, x=list(TRUE, TRUE, FALSE))
str(profil)
## List of 3
##  $ name: chr "Till"
##  $ plz : num 60326
##  $ x   :List of 3
##   ..$ : logi TRUE
##   ..$ : logi TRUE
##   ..$ : logi FALSE
```

4.5 Data Frames

Data frames sind tabellarische Daten. Die Werte in jeder Spalte haben dabei denselben Typ.

Viele Beispieldatensätze sind in Form von data frames in R eingebaut.

`head(x)` gibt nur die ersten sechs Zeilen aus:

```
head(mtcars)
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

4.6 Tibbles

Tibbles können alles, was data frames können, und haben darüber hinaus noch Funktionen, die wir später kennenlernen werden.

Sie sind teil der Paketsammlung `tidyverse`, die einmalig installiert werden muss und dann geladen werden kann:

```
library(tidyverse)
```

Ein Beispieldatensatz ist `diamonds`:

```
diamonds
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

Einzelne Spalten lassen sich mit $ ansprechen und verhalten sich dann wie Vektoren:

```
str(diamonds$carat)
##  num [1:53940] 0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
mean(diamonds$depth)
## [1] 61.7494
```

4.7 Aufgaben

4.7.1 Vektoren

- Generieren Sie die folgenden Vektoren (und seien Sie dabei möglichst faul).

```
## [1]  TRUE FALSE FALSE
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
## [13]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##  [1] "Z" "Y" "X" "W" "V" "U" "T" "S" "R" "Q" "P" "O" "N" "M" "L" "K" "J" "I" "H"
## [20] "G" "F" "E" "D" "C" "B" "A"
##  [1] "aA" "bB" "cC" "dD" "eE" "fF" "gG" "hH" "iI" "jJ" "kK" "lL" "mM" "nN" "oO"
## [16] "pP" "qQ" "rR" "sS" "tT" "uU" "vV" "wW" "xX" "yY" "zZ"
```

- Wandeln Sie die Typen der ersten drei obigen Vektoren um:

```
## [1] 1 0 0
##  [1] 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
##  [1] "2"  "4"  "6"  "8"  "10" "12" "14" "16" "18" "20" "22" "24" "26" "28" "30"
## [16] "32" "34"
```

4.7.2 Tibbles

- Schauen Sie sich den Beispieldatensatz `faithful` an.
- Wandeln Sie den Datensatz `faithful` in einen tibble um.
- Wenden Sie `str()` auf den Datensatz an. und Interpretieren Sie das Ergebnis.
- Erstellen Sie einen eigenen tibble mit Vornamen, Nachnamen und Alter von (ausgedachten?) Menschen.
- Lassen Sie sich nur die zweite Zeile des tibbles `diamonds` anzeigen
- Lassen Sie sich nur jede zweite Zeile des tibbles `diamonds` anzeigen

4.7.3 Statistik

- Berechnen Sie die durchschnittliche Eruptionszeit im Datensatz `faithful` (als tibble).
- Berechnen Sie Varianz und Standardabweichung der Karatzahl im Beispieldatensatz `diamonds`
- Was sagen die einzelnen Kennzahlen des Befehls `summary(x)` aus?

4.7.4 Swirl

Absolvieren Sie die folgenden Swirl-Lektionen (Anleitung zu Swirl s. letzte Lektion):

- 3: Sequences of Numbers
- 4: Vectors
- 5: Missing Values
- 6: Subsetting Vectors

4.7.5 Recherche

- Nach welcher Methode berechnet R den Quartilsabstand einer Verteilung (im Unterschied zur Vorlesung)?
- Finden Sie fünf Befehle, die mit tibbles funktionieren, aber nicht mit data frames.
- Welche Pakete sind Teil des `tidyverse`? Wofür sind sie gedacht?
- Lesen Sie die Hilfe zu `tibble::tibble`. Recherchieren Sie eigenständig unklare Begriffe.

4.7.6 Kniffliges

- Kehren Sie auf möglichst elegante und allgemeingültige Weise die Reihenfolge eines Vektors um, ohne die Funktion `rev()` zu benutzen.

5 Visualisierungen

5.1 Lernziele dieser Sitzung

Sie können…

- einfache Befehle zur Visualisierung in Base R anwenden.
- die Grammatik von `ggplot2` für Visualisierungen in Grundzügen wiedergeben und anwenden.
- eigene Ideen für Visualisierungen entwickeln und umsetzen.

5.2 Voraussetzungen

Für diese Lektion benötigen wir das Paket `tidyverse`:

```
library(tidyverse)
```

Und einen Datensatz, der in Form eines tibble vorliegt. Der Beispieldatensatz `diamonds` wird mitgeliefert:

```
diamonds
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
```

```
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

Wenn wir mögen, können wir ihn mit der Funktion `data()` explizit in unser Environment laden:

```
data(diamonds)
```

5.3 Überblick

Einen ersten Überblick kriegen wir zum Einen durch den Befehl `str()`, der uns die Typen in den Spalten anzeigt:

```
str(diamonds)
## tibble [53,940 x 10] (S3: tbl_df/tbl/data.frame)
##  $ carat  : num [1:53940] 0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num [1:53940] 61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num [1:53940] 55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int [1:53940] 326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num [1:53940] 3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num [1:53940] 3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num [1:53940] 2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

Zum Anderen gibt die Hilfefunktion Auskunft über den Datensatz und die einzelnen Variablen (Metadaten):

```
?diamonds
```

Einen Überblick über die wichtigsten statistischen Parameter erhalten wir mit:

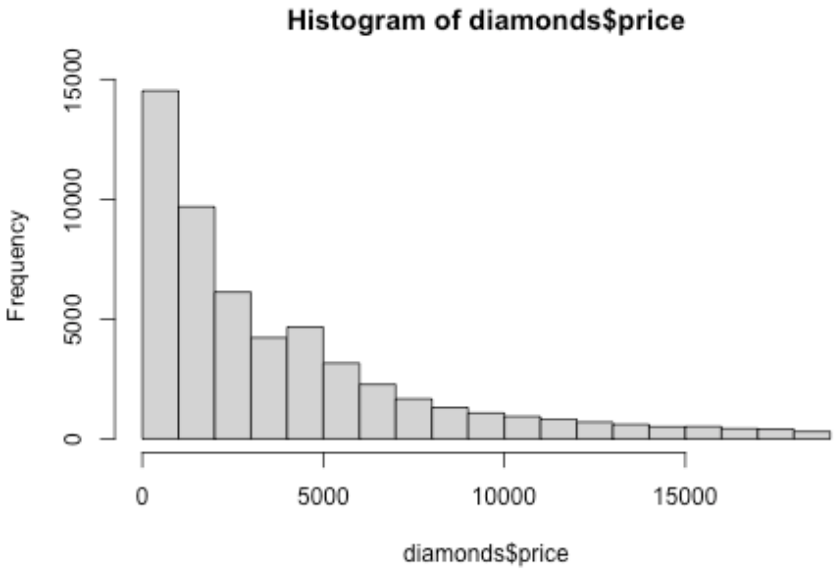
```
summary(diamonds)
##      carat               cut        color        clarity          depth      
##  Min.   :0.2000   Fair      : 1610   D: 6775   SI1    :13065   Min.   :43.00  
##  1st Qu.:0.4000   Good      : 4906   E: 9797   VS2    :12258   1st Qu.:61.00  
##  Median :0.7000   Very Good:12082   F: 9542   SI2    : 9194   Median :61.80  
##  Mean   :0.7979   Premium   :13791   G:11292   VS1    : 8171   Mean   :61.75  
```

```
##  3rd Qu.:1.0400   Ideal    :21551   H: 8304   VVS2   : 5066   3rd Qu.:62.50
##  Max.   :5.0100                    I: 5422   VVS1   : 3655   Max.   :79.00
##                                    J: 2808   (Other): 2531
##      table           price             x                y
##  Min.   :43.00   Min.   :  326   Min.   : 0.000   Min.   : 0.000
##  1st Qu.:56.00   1st Qu.:  950   1st Qu.: 4.710   1st Qu.: 4.720
##  Median :57.00   Median : 2401   Median : 5.700   Median : 5.710
##  Mean   :57.46   Mean   : 3933   Mean   : 5.731   Mean   : 5.735
##  3rd Qu.:59.00   3rd Qu.: 5324   3rd Qu.: 6.540   3rd Qu.: 6.540
##  Max.   :95.00   Max.   :18823   Max.   :10.740   Max.   :58.900
##
##        z
##  Min.   : 0.000
##  1st Qu.: 2.910
##  Median : 3.530
##  Mean   : 3.539
##  3rd Qu.: 4.040
##  Max.   :31.800
##
```

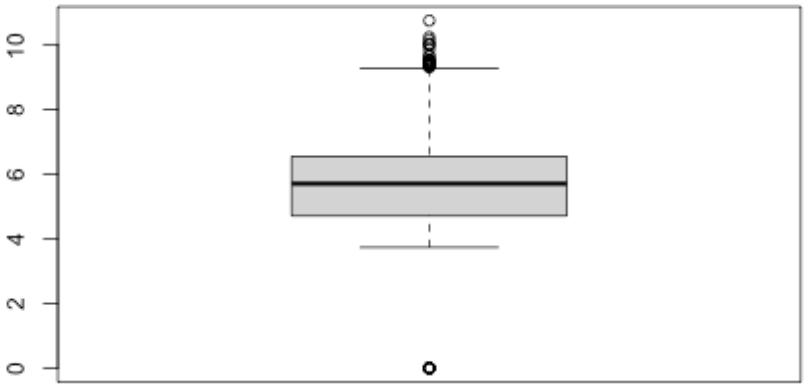
5.4 Visualisierung mit dem Standardpaket

Es gibt in R mehrere grundlegend verschiedene Möblichkeiten, Daten zu visualiseren. Für einen schnellen Überblick sind z.B. `hist()` und `boxplot()` hilfreich:

```
hist(diamonds$price)
```




```
boxplot(diamonds$x)
```



5.5 Visualisierung mit `ggplot()`

Das Paket `ggplot2` ist Teil vom `tidyverse`. Hiermit lassen sich sehr flexible Graphiken gestalten. Wir werden ausschließlich mit diesem System arbeiten.

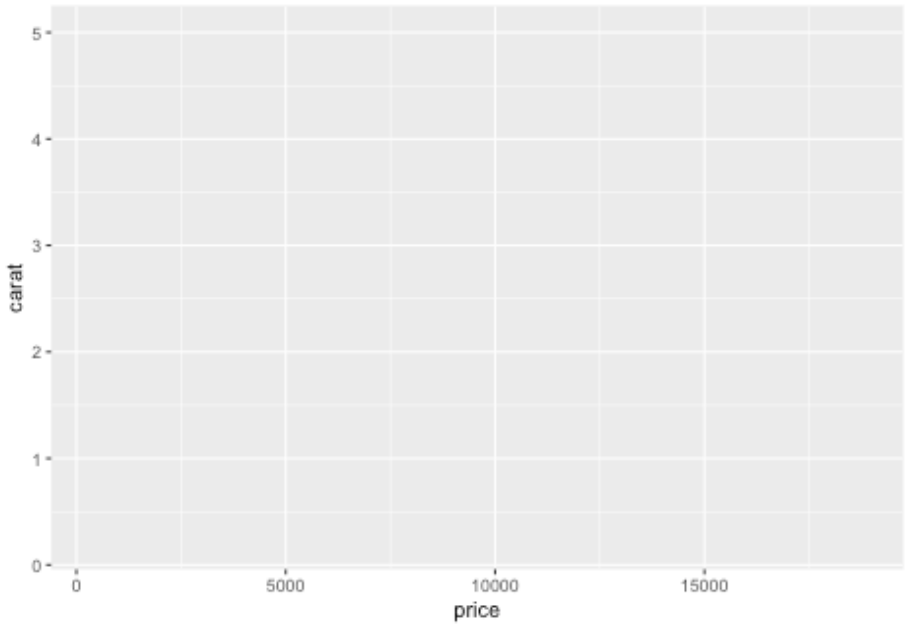
Die Syntax ist dabei auf den ersten Blick etwas komplexer.

Am Anfang steht der Befehl `ggplot(x)` mit dem Datensatz als Parameter

```
ggplot(data=diamonds)
```

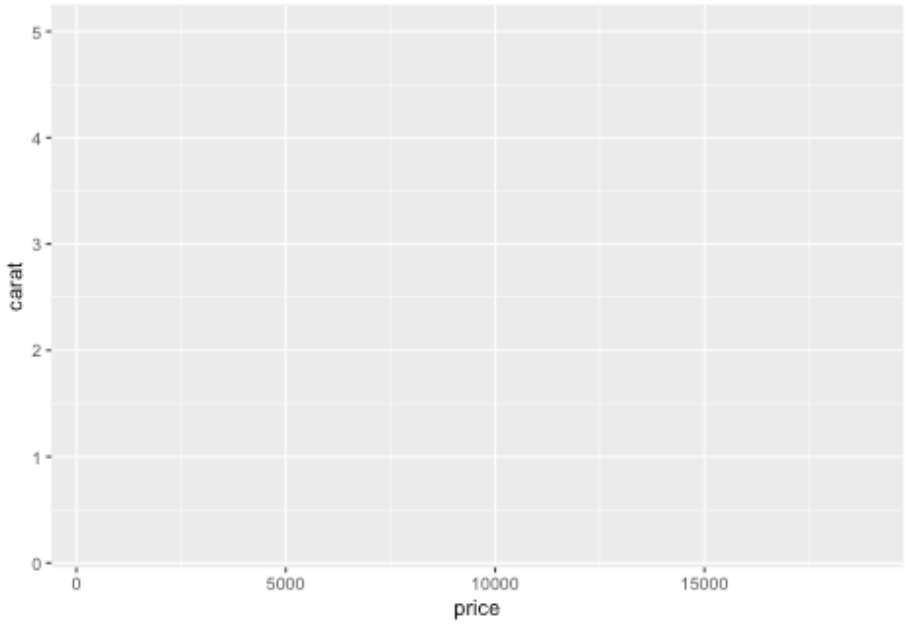

Mit einem Mapping-Parameter legen wir die Dimensionen fest:

```
ggplot(data=diamonds, mapping=aes(x=price, y=carat))
```



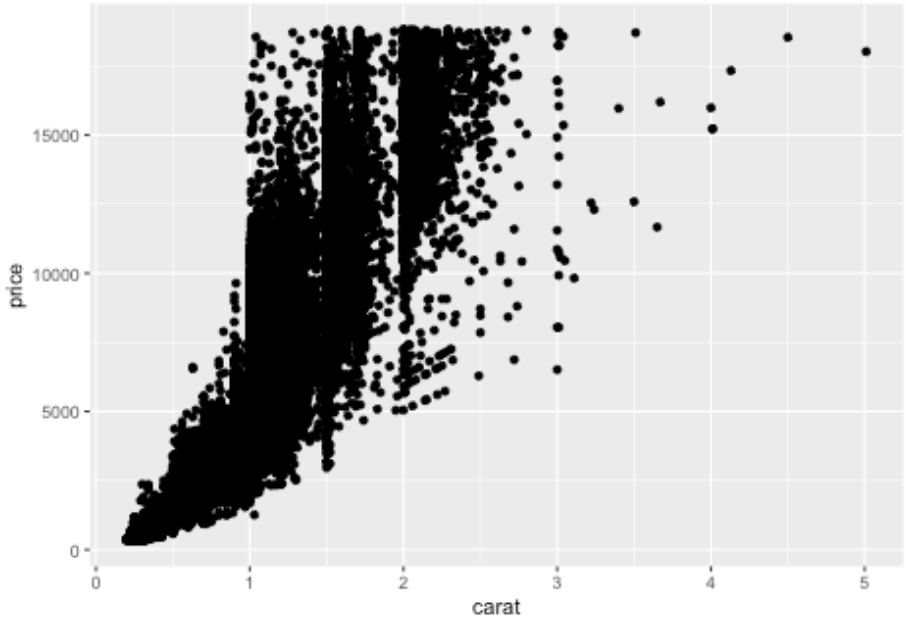
Das gleiche ohne Parameternamen:

```
ggplot(diamonds, aes(price, carat))
```



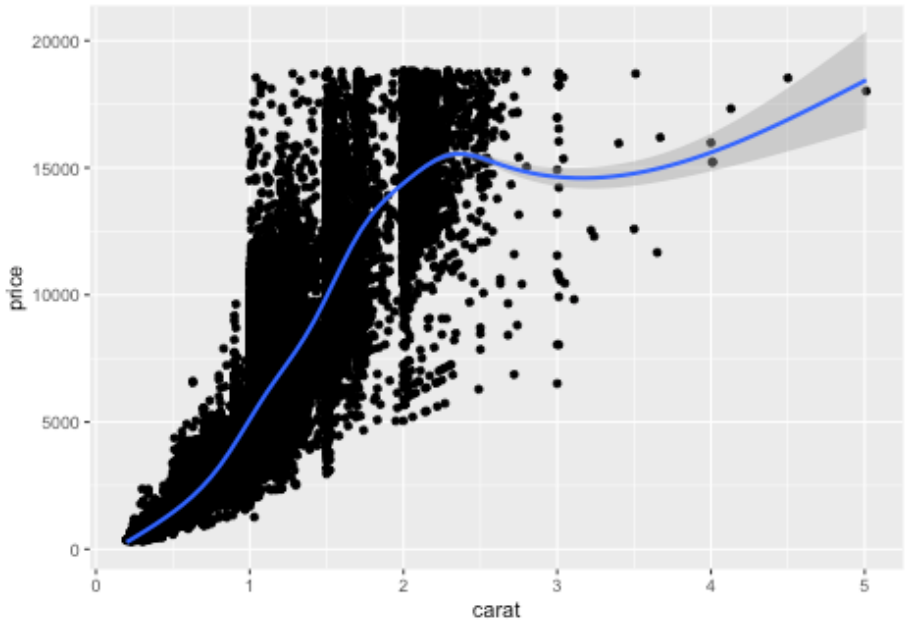
Nun kann mit dem +-Operator ein “geometrischer” Layer hinzugefügt werden:

```
ggplot(diamonds, aes(x=carat, y=price)) +
  geom_point()
```



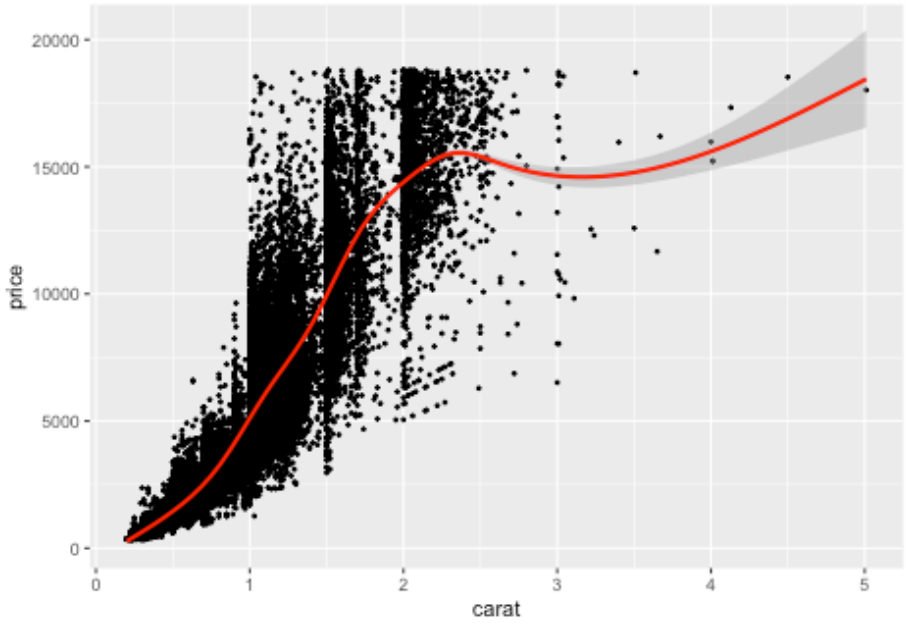
Weitere `geom`-Layer lassen sich mit dem `+`-Operator hinzufügen:

```
ggplot(diamonds, aes(x=carat, y=price)) +
  geom_point() +
  geom_smooth()
```



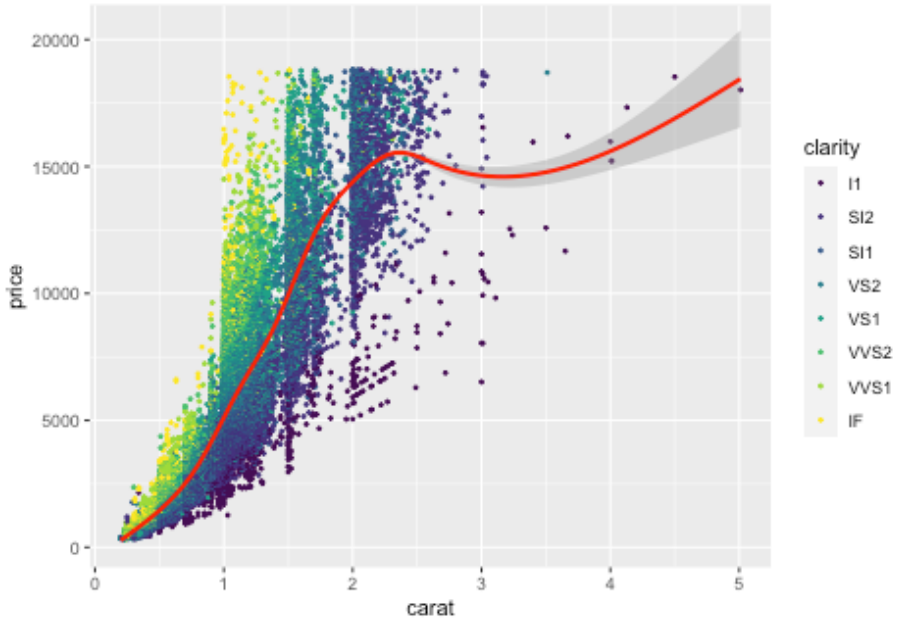
Die Layer-Funktionen können durch Parameter angepasst werden:

```
ggplot(diamonds, aes(x=carat, y=price)) +
  geom_point(size=0.5) +
  geom_smooth(color="red")
```



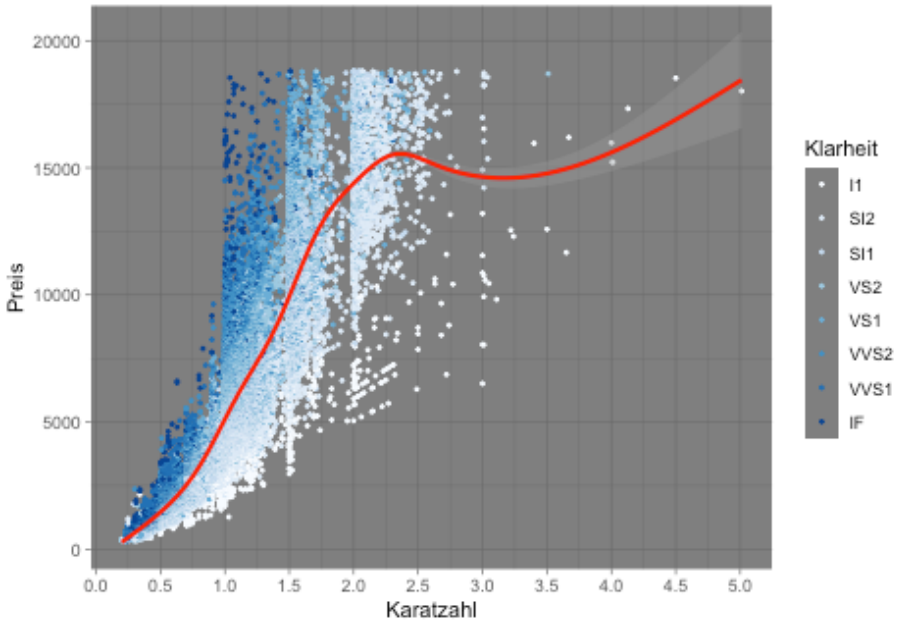
Dabei lassen sich in den einzelnen Layers mappings hinzufügen oder verändern:

```
ggplot(diamonds, aes(x=carat, y=price)) +
  geom_point(aes(color=clarity), size=0.5) +
  geom_smooth(color="red")
```



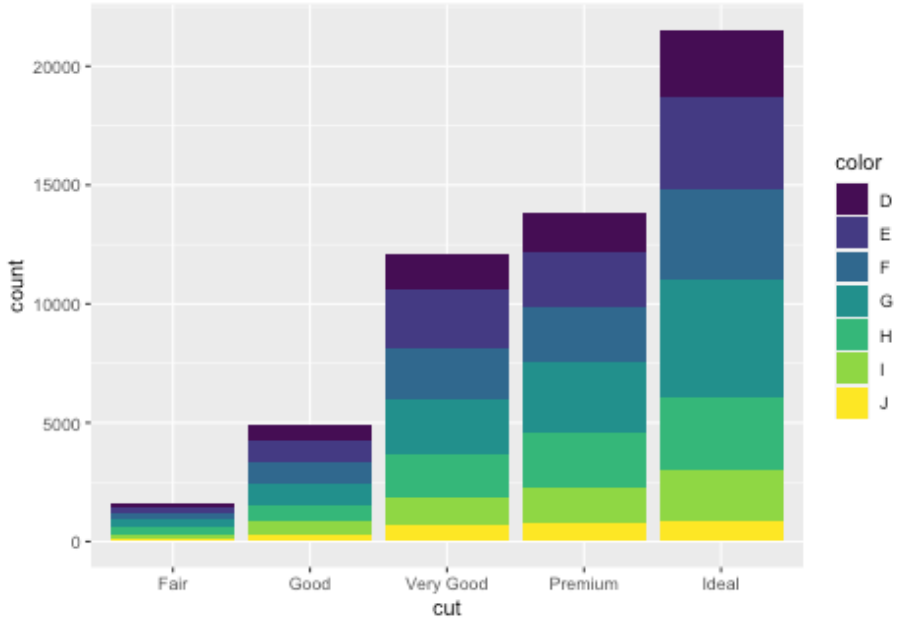
Schließlich lassen sich noch viele weitere optische Aspekte anpassen, z.B. Achsen, Farben, etc.:

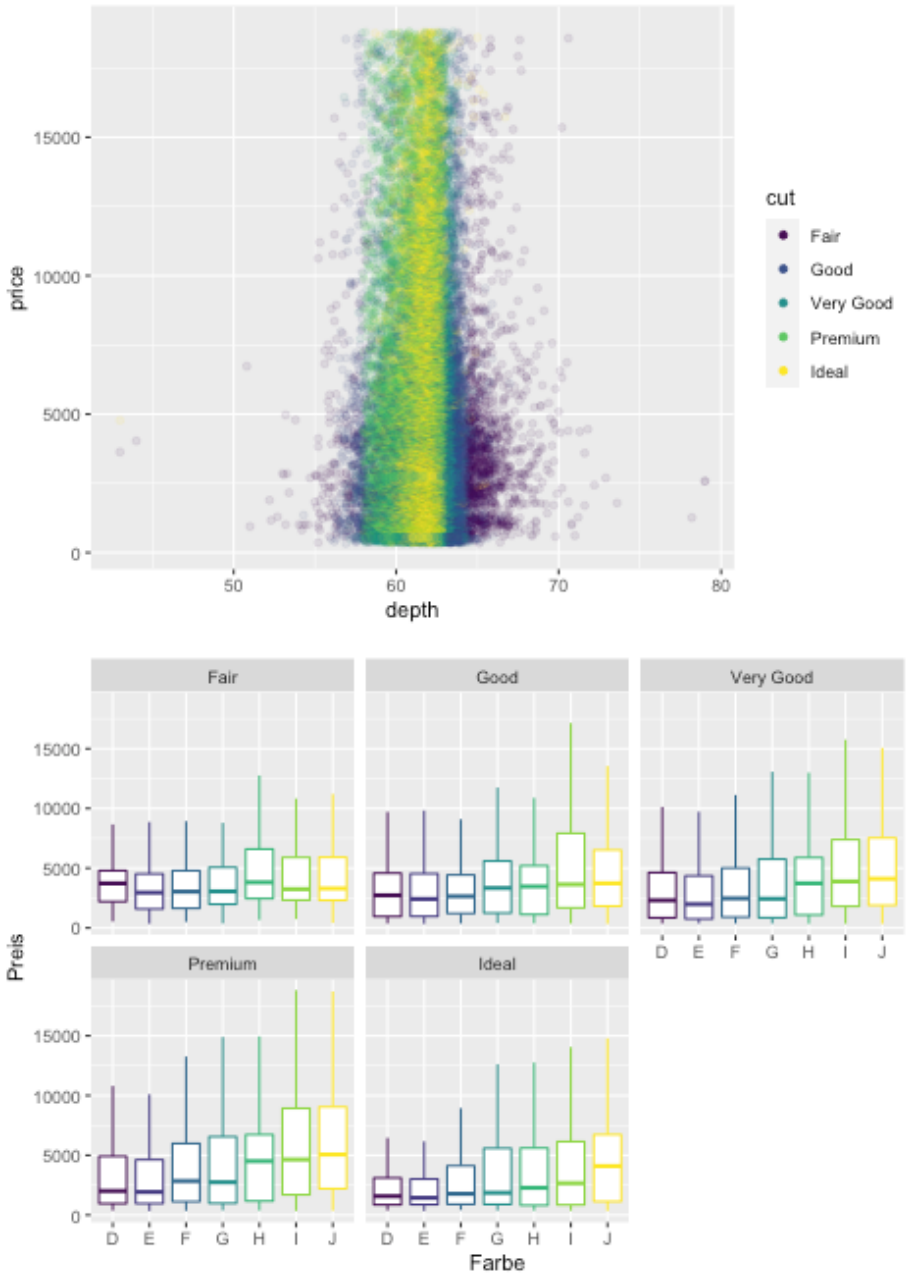
```
ggplot(diamonds, aes(x=carat, y=price)) +
  geom_point(aes(color=clarity), size=0.5) +
  geom_smooth(color="red") +
  scale_x_continuous("Karatzahl", breaks=seq(0,5,0.5)) +
  scale_y_continuous("Preis") +
  scale_color_brewer("Klarheit") +
  theme_dark()
```



5.6 Aufgaben

Versuchen Sie, folgende Visualisierungen des Datensatzes `diamonds` auszugeben:





5.6.1 R for Data Science

Schauen Sie sich die Publikation R for Data Science an.

Was ist das für ein Buch? Wer ist das Zielpublikum?

Lesen Sie das Kapitel “3: Data Visualization” und vollziehen Sie die Visualisierungen nach.

Bearbeiten Sie die Aufgaben.

Bearbeiten Sie die RStudio Primers zu Datenvisualisierung.

6 Text: Shelton et al. 2014

6.1 Lesetext

Shelton, Taylor, Ate Poorthuis, Mark Graham und Matthew Zook. 2014. Mapping the Data Shadows of Hurricane Sandy. Uncovering the Sociospatial Dimensions of ‘Big Data’. *Geoforum 52.* 167–79.

6.2 Fragen an den Text

1. Um welche Art von Text handelt es sich? Wer sind die Autoren, und an wen wenden sie sich?
2. Was war Hurricane Sandy, von dem der Text erzählt? Warum wird ausgerechnet dieser Hurricane herangezogen?
3. Welche Methoden wenden die Autoren an, und zu welchen Ergebnissen kommen sie?
4. Im Abstract versprechen die Autoren: “We also seek to fill a conceptual lacuna…” Was heißt das, und wie geht der Text das an?
5. In welchen Punkten finden Sie den Text überzeugend? Welche Kritik haben Sie am Text?

7 Geodaten

7.1 Lernziele dieser Sitzung

Sie können…

- Pipes benutzen
- einfache `dplyr`-Befehle ausführen
- Koordinaten visualisieren

7.2 Voraussetzungen

Wir laden erstmal `tidyverse`:

```
library(tidyverse)
```

7.3 Exkurs: Pipes

Teil vom `tidyverse` ist auch das Paket `magrittr`, das einen besonderen Operator enthält: `%>%`

Der Operator `%>%` heißt "Pipe" und setzt das Ergebnis der vorherigen Funktion als ersten Parameter in die nächste Funktion ein. Zur Veranschaulichung:

```
anzahl_buchstaben <- length(letters)
sqrt(anzahl_buchstaben)
```

...ist das gleiche wie...

```
sqrt(length(letters))
```

...ist das gleiche wie...

```
length(letters) %>%
  sqrt()
```

...ist das gleiche wie...

```
letters %>%
  length %>%
  sqrt()
```

So können beliebig viele Funktionen aneinandergereiht werden. Und mit `->` kann eine Variable „in die andere Richtung" zugewiesen werden

```
letters %>%
  length() %>%
  sqrt() %>%
  round() %>%
  as.character() ->
  my_var
```

Gerade bei komplizierteren Zusammenhängen wird der Code so oft lesbarer, weil die Logik von links nach rechts, bzw. von oben nach unten gelesen werden kann.

7.4 Daten importieren

Beim Open-Data-Portal der Stadt Frankfurt steht ein Baumkataster zur Verfügung.

Die Datei im CSV-Format (comma separated values) kann entweder heruntergeladen und durch klicken importiert werden, oder direkt über den Befehl:

```
baumkataster <- read_csv2("http://offenedaten.frankfurt.de/dataset/73c5a6b3-c033-4dad-bb7d-8
```

7.5 Überblick verschaffen

Mit `summary()` lässt sich eine Zusammenfassung der Werte generieren:

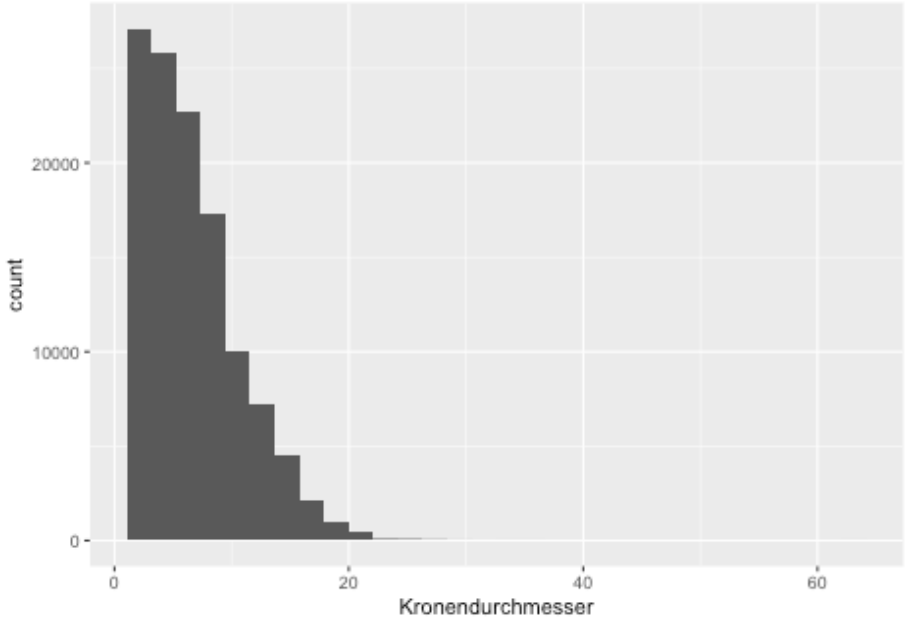
```
summary(baumkataster)
##  Gattung/Art/Deutscher Name   Baumnummer         Objekt             Pflanzjahr
##  Length:118403              Min.   :    1.0   Length:118403      Min.   :1645
##  Class :character           1st Qu.:   24.0   Class :character   1st Qu.:1970
##  Mode  :character           Median :   82.0   Mode  :character   Median :1982
##                             Mean   :  232.7                      Mean   :1979
##                             3rd Qu.:  270.0                      3rd Qu.:1995
##                             Max.   :20158.0                      Max.   :2017
##                             NA's   :1853
##  Kronendurchmesser    HOCHWERT         RECHTSWERT
##  Min.   : 2.000    Min.   :5545117   Min.   :463163
##  1st Qu.: 4.000    1st Qu.:5550428   1st Qu.:472715
##  Median : 6.000    Median :5552601   Median :475219
##  Mean   : 6.688    Mean   :5552953   Mean   :475244
##  3rd Qu.: 9.000    3rd Qu.:5555165   3rd Qu.:478201
##  Max.   :63.000    Max.   :5563639   Max.   :485361
##
```

Genauere Infos über diese Merkmale gibt es auf dem Datenportal.

7.6 Visualisieren

Wie in der letzten Lektion besprochen, lässt sich der Datensatz mit `ggplot()` visualisieren, z.B.:

```
ggplot(baumkataster, aes(x=Kronendurchmesser)) +
  geom_histogram()
```



Eine neue Messreihe lässt sich z.B. so errechnen:

```
alter <- 2020 - baumkataster$Pflanzjahr
head(alter)
## [1] 100 100 100 100 100 100
```

Der Befehl `mutate()` funktioniert sehr ähnlich, gibt aber den veränderten Datensatz zurück:

```
mutate(baumkataster, alter = 2020 - Pflanzjahr)
## # A tibble: 118,403 x 8
##    `Gattung/Art/Deutsch~ Baumnummer Objekt  Pflanzjahr Kronendurchmess~ HOCHWERT
##    <chr>                      <dbl> <chr>        <dbl>            <dbl>    <dbl>
##  1 Platanus x hispanica~          1 Ackerm~       1920                8 5549511.
##  2 Platanus x hispanica~          2 Ackerm~       1920                8 5549517.
##  3 Platanus x hispanica~          3 Ackerm~       1920                8 5549524.
##  4 Platanus x hispanica~          4 Ackerm~       1920                8 5549531
##  5 Platanus x hispanica~          5 Ackerm~       1920                8 5549538.
##  6 Platanus x hispanica~          6 Ackerm~       1920                8 5549544.
##  7 Platanus x hispanica~          7 Ackerm~       1920                8 5549551.
##  8 Platanus x hispanica~          8 Ackerm~       1920                8 5549557.
##  9 Platanus x hispanica~          9 Ackerm~       1920                8 5549564.
## 10 Platanus x hispanica~         10 Ackerm~       1920                8 5549571.
## # ... with 118,393 more rows, and 2 more variables: RECHTSWERT <dbl>,
## #   alter <dbl>
```

Derselbe Befehl mit dem Pipe-Operator:

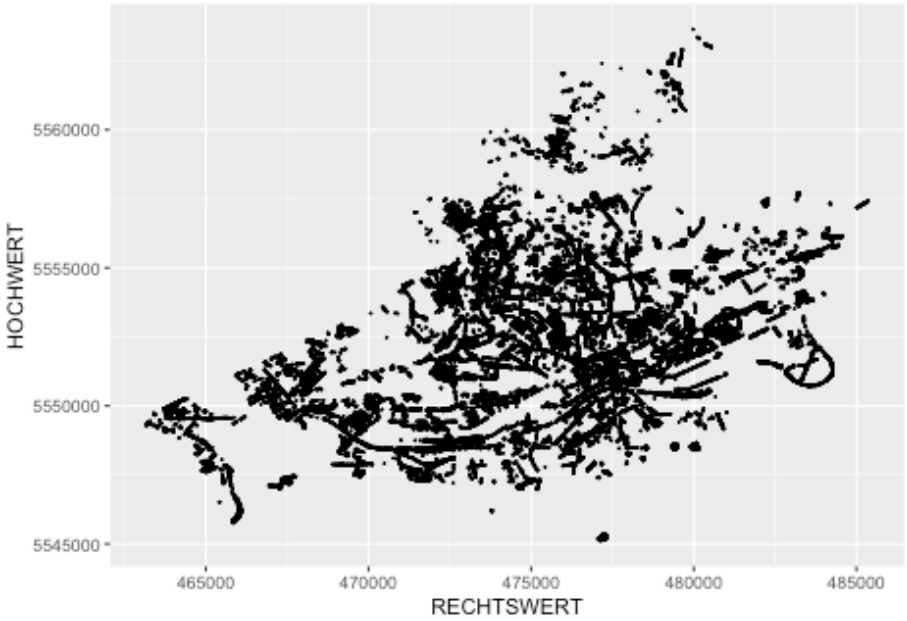
```
baumkataster %>%
  mutate(alter = 2020 - Pflanzjahr)
## # A tibble: 118,403 x 8
##    `Gattung/Art/Deutsch~ Baumnummer Objekt  Pflanzjahr Kronendurchmess~ HOCHWERT
##    <chr>                       <dbl> <chr>        <dbl>            <dbl>    <dbl>
##  1 Platanus x hispanica~           1 Ackerm~       1920                8 5549511.
##  2 Platanus x hispanica~           2 Ackerm~       1920                8 5549517.
##  3 Platanus x hispanica~           3 Ackerm~       1920                8 5549524.
##  4 Platanus x hispanica~           4 Ackerm~       1920                8 5549531
##  5 Platanus x hispanica~           5 Ackerm~       1920                8 5549538.
##  6 Platanus x hispanica~           6 Ackerm~       1920                8 5549544.
##  7 Platanus x hispanica~           7 Ackerm~       1920                8 5549551.
##  8 Platanus x hispanica~           8 Ackerm~       1920                8 5549557.
##  9 Platanus x hispanica~           9 Ackerm~       1920                8 5549564.
## 10 Platanus x hispanica~          10 Ackerm~       1920                8 5549571.
## # ... with 118,393 more rows, and 2 more variables: RECHTSWERT <dbl>,
## #   alter <dbl>
```

So lassen sich auch hier verschiedene Befehle verknüpfen. `filter()` beschränkt den Datensatz auf Merkmalsträger, die den Kriterien entsprechen:

```
baumkataster %>%
  mutate(alter = 2020 - Pflanzjahr) %>%
  filter(alter > 30) ->
  alte_baeume

summary(alte_baeume)
##  Gattung/Art/Deutscher Name   Baumnummer         Objekt            Pflanzjahr
##  Length:73859               Min.   :    1.0   Length:73859       Min.   :1645
##  Class :character           1st Qu.:   29.0   Class :character   1st Qu.:1960
##  Mode  :character           Median :   97.0   Mode  :character   Median :1974
##                             Mean   :  263.2                      Mean   :1966
##                             3rd Qu.:  314.0                      3rd Qu.:1980
##                             Max.   :10489.0                      Max.   :1989
##                             NA's   :684
##  Kronendurchmesser    HOCHWERT         RECHTSWERT         alter
##  Min.   : 2.000    Min.   :5545117   Min.   :463163   Min.   : 31.00
##  1st Qu.: 6.000    1st Qu.:5550415   1st Qu.:472667   1st Qu.: 40.00
##  Median : 8.000    Median :5552480   Median :475708   Median : 46.00
##  Mean   : 8.503    Mean   :5552593   Mean   :475402   Mean   : 53.54
##  3rd Qu.:10.000    3rd Qu.:5554589   3rd Qu.:478539   3rd Qu.: 60.00
##  Max.   :35.000    Max.   :5563639   Max.   :485360   Max.   :375.00
##
```

Schließlich ergibt das Streudiagramm von Koordinaten so eine art Karte:

```
ggplot(alte_baeume) +
    geom_point(size = 0.1, aes(x = RECHTSWERT, y = HOCHWERT))
```



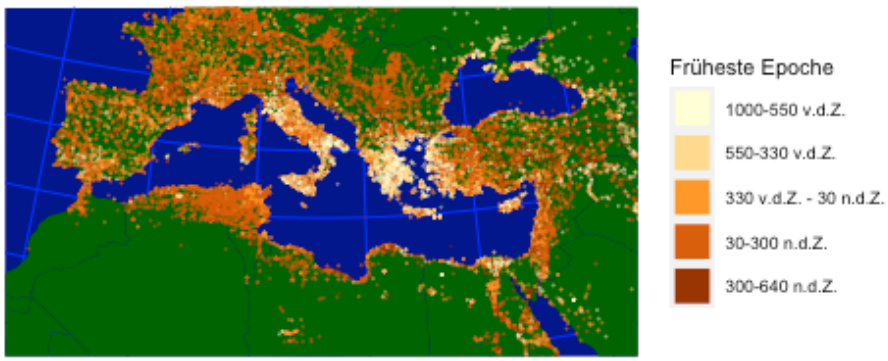
Diesen Ansatz werden wir in der nächsten Lektion vertiefen.

7.7 Aufgaben

1. Besuchen Sie https://pleiades.stoa.org/ - worum geht es hier?
2. Finden Sie den kompletten aktuellen Datensatz für „locations“ als CSV-Datei.
3. Importieren Sie ihn in R und weisen Sie dem Datensatz den Namen `pleiades` zu.
4. Finden Sie geeignete Werte für (einzelne) Längen- und Breitengrade im Datensatz.
5. Plotten Sie die Koordinaten auf x- und y-Achse mit `ggplot()`. Was erkennen Sie?
6. Halbieren Sie die Größe und setzen Sie den Alpha-Wert der Punkte auf 0,2.
7. Bringen Sie die Grafik in die Mercator-Projektion.
8. Schauen Sie sich diesen Befehl an:

```
map_data("world") %>%
  ggplot() +
    geom_polygon(mapping = aes(x = long,
                               y = lat,
                               group = group)) +
    coord_quickmap(xlim = c(-8, 40),
                   ylim = c(26, 48))
```

9. Versuchen Sie, jede einzelne Zeile nachzuvollziehen, indem Sie die entsprechenden Funktionen recherchieren.
10. Führen Sie den Befehl aus.
11. Ändern Sie die Farbe der Flächen in hellgrau.
12. Wählen Sie einen Kartenausschnitt, auf dem Portugal, Ägypten, Irak und Frankreich komplett zu sehen sind.
13. Plotten Sie auf diesem Hintergrund den Datensatz `pleiades`. Passen Sie dabei die Parameter so an, dass es Ihnen optisch zusagt.
14. Wählen Sie für die Karte die Bonnesche Projektion mit Standardparallele bei 40°N.
15. Entfernen Sie alle Achsenbeschriftungen.
16. (Achtung: extrem knifflig!) Bilden Sie diese Grafik nach, die die Orte geordnet nach ältestem Fund darstellt:



8 Choroplethen

8.1 Lernziele

Sie können…

- Geodaten als Simple Features importieren,
- CRS bestimmen und umwandeln,
- einfache Verschneidungen von Simple Features durchführen und
- Simple Features kartographisch darstellen.

8.2 Vorbereitung

Für diese Lektion werden zwei Pakete geladen:

```
library(tidyverse)
library(sf)
```

8.3 Ziel

Ziel ist, eine Choroplethenkarte von Frankfurt zu erstellen, die die Versorgung mit Kiosken darstellt.

8.4 Grundkarte

Eine Shapefile der Frankfurter Stadtteile findet sich hier: http://www.offenedaten.frankfurt.de/dataset/frankfurter-stadtteilgrenzen-fur-gis-systeme

Wir laden die Zip-Datei herunter und speichern den enthaltenen Ordner `stadtteile` in unserem Arbeitsverzeichnis. Es ist eine gute Angewohnheit, einen Unterordner für Ressourcen anzulegen.

Dann importieren wir den Geodatensatz als Simple Features (Paket `sf`):

```
stadtteile <- st_read("resources/stadtteile/Stadtteile_Frankfurt_am_Main.shp")
## Reading layer `Stadtteile_Frankfurt_am_Main' from data source
##   `/Users/till/teaching/2020x21_Data_Science/skript/resources/stadtteile/Stadtteile_Frank
##   using driver `ESRI Shapefile'
## Simple feature collection with 46 features and 2 fields
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: 462292.7 ymin: 5540412 xmax: 485744.8 ymax: 5563925
## Projected CRS: ETRS89 / UTM zone 32N
```

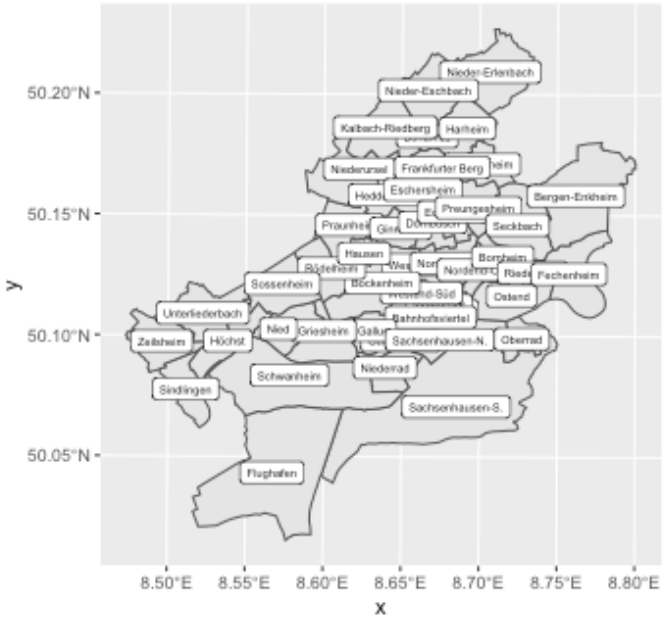
Simple Features sind Datensätze, die eine Spalte `geometry` enthalten, in der Geodaten in einem standardisierten Format hinterlegt sind.

```
str(stadtteile)
## Classes 'sf' and 'data.frame':   46 obs. of  3 variables:
```

```
##  $ STTLNR  : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ STTLNAME: chr  "Altstadt" "Innenstadt" "Bahnhofsviertel" "Westend-Süd" ...
##  $ geometry:sfc_POLYGON of length 46; first list element: List of 1
##   ..$ : num [1:46, 1:2] 476934 476890 476852 476813 476799 ...
##   ..- attr(*, "class")= chr [1:3] "XY" "POLYGON" "sfg"
##  - attr(*, "sf_column")= chr "geometry"
##  - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",..: NA NA
##   ..- attr(*, "names")= chr [1:2] "STTLNR" "STTLNAME"
```

Eine Vorschau:

```
ggplot(stadtteile) +
  geom_sf() +
  geom_sf_label(aes(label = STTLNAME), size = 2)
```



8.5 OSM-Daten

Im OSM Wiki suchen wir den richtigen *tag* heraus. In diesem Fall `shop=kiosk`

Dann bauen wir auf Overpass Turbo die Abfrage und laden den Datensatz herunter.

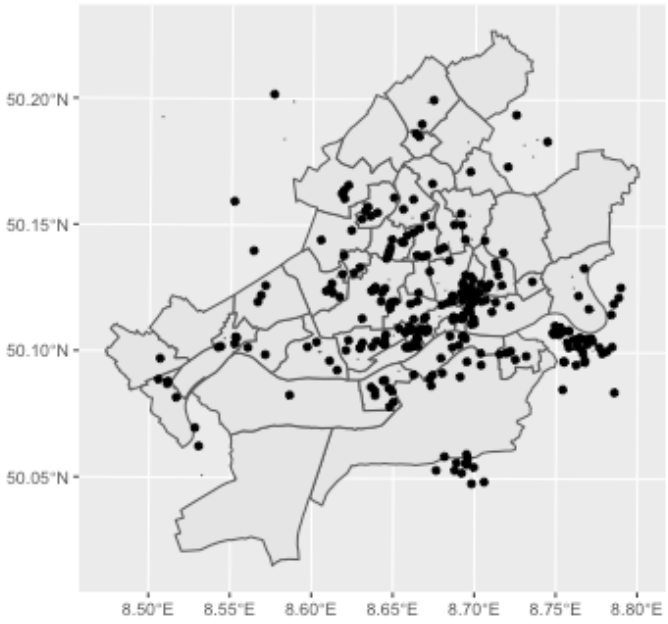
Schließlich importieren wir:

```
kioske <- st_read("resources/kioske.geojson")
## Reading layer `kioske' from data source
##   `/Users/till/teaching/2020x21_Data_Science/skript/resources/kioske.geojson'
```

```
##   using driver `GeoJSON'
## Simple feature collection with 325 features and 74 fields
## Geometry type: GEOMETRY
## Dimension:     XY
## Bounding box:  xmin: 8.505468 ymin: 50.04801 xmax: 8.789538 ymax: 50.20185
## Geodetic CRS:  WGS 84
```

Eine Vorschau:

```
ggplot() +
  geom_sf(data = stadtteile) +
  geom_sf(data = kioske)
```



8.6 Koordinatenreferenzsysteme

Der OSM-Datensatz ist mit WGS84 (EPSG 4326) referenziert:

```
st_crs(kioske)
## Coordinate Reference System:
##   User input: WGS 84
##   wkt:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]]],
##     PRIMEM["Greenwich",0,
```

```
##         ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##         AXIS["geodetic latitude (Lat)",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["geodetic longitude (Lon)",east,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433]],
##     ID["EPSG",4326]]
```

Die Stadtteilen hingegen sind sind in ETSR89 (EPSG 25832):

```
st_crs(stadtteile)
## Coordinate Reference System:
##   User input: ETRS89 / UTM zone 32N
##   wkt:
## PROJCRS["ETRS89 / UTM zone 32N",
##     BASEGEOGCRS["ETRS89",
##         DATUM["European Terrestrial Reference System 1989",
##             ELLIPSOID["GRS 1980",6378137,298.257222101,
##                 LENGTHUNIT["metre",1]]],
##         PRIMEM["Greenwich",0,
##             ANGLEUNIT["degree",0.0174532925199433]],
##         ID["EPSG",4258]],
##     CONVERSION["UTM zone 32N",
##         METHOD["Transverse Mercator",
##             ID["EPSG",9807]],
##         PARAMETER["Latitude of natural origin",0,
##             ANGLEUNIT["degree",0.0174532925199433],
##             ID["EPSG",8801]],
##         PARAMETER["Longitude of natural origin",9,
##             ANGLEUNIT["degree",0.0174532925199433],
##             ID["EPSG",8802]],
##         PARAMETER["Scale factor at natural origin",0.9996,
##             SCALEUNIT["unity",1],
##             ID["EPSG",8805]],
##         PARAMETER["False easting",500000,
##             LENGTHUNIT["metre",1],
##             ID["EPSG",8806]],
##         PARAMETER["False northing",0,
##             LENGTHUNIT["metre",1],
##             ID["EPSG",8807]]],
##     CS[Cartesian,2],
##         AXIS["(E)",east,
##             ORDER[1],
##             LENGTHUNIT["metre",1]],
```

```
##         AXIS["(N)",north,
##             ORDER[2],
##             LENGTHUNIT["metre",1]],
##     USAGE[
##         SCOPE["Engineering survey, topographic mapping."],
##         AREA["Europe between 6°E and 12°E: Austria; Belgium; Denmark - onshore and offsho
##         BBOX[38.76,6,83.92,12]],
##     ID["EPSG",25832]]
```

Der Datensatz lässt sich allerdings transformieren:

```
stadtteile %>%
  st_transform(4326) %>%
  st_crs()
## Coordinate Reference System:
##   User input: EPSG:4326
##   wkt:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##         AXIS["geodetic latitude (Lat)",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["geodetic longitude (Lon)",east,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433]],
##     USAGE[
##         SCOPE["Horizontal component of 3D system."],
##         AREA["World."],
##         BBOX[-90,-180,90,180]],
##     ID["EPSG",4326]]
```

Jetzt haben beide Datensätze den selben EPSG-Code. Das ist die Voraussetzung für den nächsten Schritt.

8.7 Verschneiden

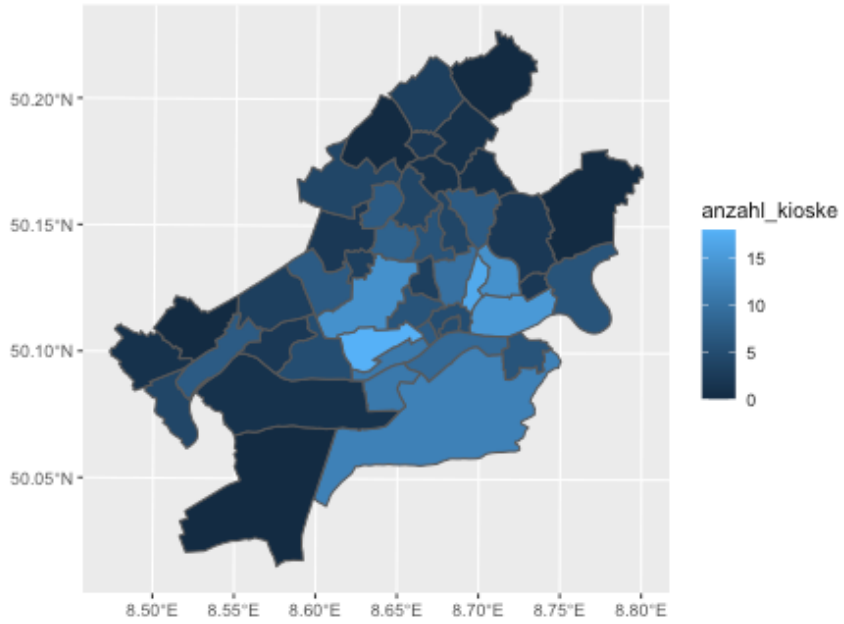
Mit `st_covers()` und `lengths()` lassen sich die Anzahl der Kioske in jedem Stadtteil zählen und einer neuen Spalte im Originaldatensatz zuordnen:

```
stadtteile %>%
  st_transform(4326) %>%
  st_covers(kioske) %>%
```

```
  lengths() -> stadtteile$anzahl_kioske
```

Auf einer Karte veranschaulicht:

```
ggplot(stadtteile) +
  geom_sf(aes(fill = anzahl_kioske))
```



Allerdings wäre es schöner, die Kioskdichte (nach Fläche) darzustellen. Dazu berechnen wir zunächst die Flächen der Stadtteile:

```
st_area(stadtteile) %>%
  as.numeric() / 1000 / 1000 ->
  stadtteile$qkm
```

Oder mit Pipes:

```
stadtteile %>%
  mutate(qkm = st_area(.) %>% as.numeric() / 1000 / 1000)
## Simple feature collection with 46 features and 4 fields
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: 462292.7 ymin: 5540412 xmax: 485744.8 ymax: 5563925
## Projected CRS: ETRS89 / UTM zone 32N
## First 10 features:
##    STTLNR        STTLNAME                       geometry anzahl_kioske
## 1       1        Altstadt POLYGON ((476934.3 5550541,...             5
## 2       2      Innenstadt POLYGON ((477611.9 5552034,...             5
```

```
## 3       3 Bahnhofsviertel POLYGON ((475831 5550785, 4...            7
## 4       4     Westend-Süd POLYGON ((475745.4 5552373,...            6
## 5       5    Westend-Nord POLYGON ((476497.9 5553910,...            3
## 6       6    Nordend-West POLYGON ((478362.5 5553898,...           10
## 7       7     Nordend-Ost POLYGON ((478397.9 5551924,...           17
## 8       8          Ostend POLYGON ((481955.2 5552141,...           15
## 9       9        Bornheim POLYGON ((478959.8 5552336,...           14
## 10     10  Gutleutviertel POLYGON ((472942 5548802, 4...           11
##          qkm
## 1  0.5065673
## 2  1.4902009
## 3  0.5425421
## 4  2.4948957
## 5  1.6307925
## 6  3.0977694
## 7  1.5305338
## 8  5.5573382
## 9  2.7840413
## 10 2.1982354
```

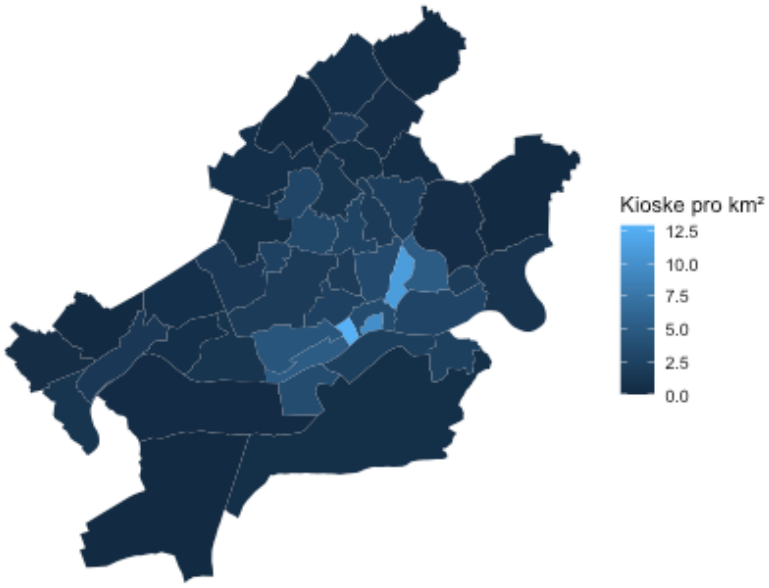
Und dann die Kioskdichte:

```
stadtteile %>%
  mutate(qkm = st_area(.) %>% as.numeric() / 1000 / 1000,
         kioskdichte = anzahl_kioske / qkm)
## Simple feature collection with 46 features and 5 fields
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: 462292.7 ymin: 5540412 xmax: 485744.8 ymax: 5563925
## Projected CRS: ETRS89 / UTM zone 32N
## First 10 features:
##    STTLNR        STTLNAME                       geometry anzahl_kioske
## 1       1        Altstadt POLYGON ((476934.3 5550541,...             5
## 2       2      Innenstadt POLYGON ((477611.9 5552034,...             5
## 3       3 Bahnhofsviertel POLYGON ((475831 5550785, 4...             7
## 4       4     Westend-Süd POLYGON ((475745.4 5552373,...             6
## 5       5    Westend-Nord POLYGON ((476497.9 5553910,...             3
## 6       6    Nordend-West POLYGON ((478362.5 5553898,...            10
## 7       7     Nordend-Ost POLYGON ((478397.9 5551924,...            17
## 8       8          Ostend POLYGON ((481955.2 5552141,...            15
## 9       9        Bornheim POLYGON ((478959.8 5552336,...            14
## 10     10  Gutleutviertel POLYGON ((472942 5548802, 4...            11
##          qkm kioskdichte
## 1  0.5065673    9.870357
## 2  1.4902009    3.355252
## 3  0.5425421   12.902225
```

```
## 4  2.4948957    2.404910
## 5  1.6307925    1.839596
## 6  3.0977694    3.228129
## 7  1.5305338   11.107236
## 8  5.5573382    2.699134
## 9  2.7840413    5.028661
## 10 2.1982354    5.004014
```

Schließlich die Karte:

```
stadtteile %>%
  mutate(qkm = st_area(.) %>% as.numeric() / 1000 / 1000,
         kioskdichte = anzahl_kioske / qkm) %>%
  ggplot() +
    geom_sf(aes(fill = kioskdichte), color=NA) +
    scale_fill_continuous("Kioske pro km²") +
    theme_void()
```

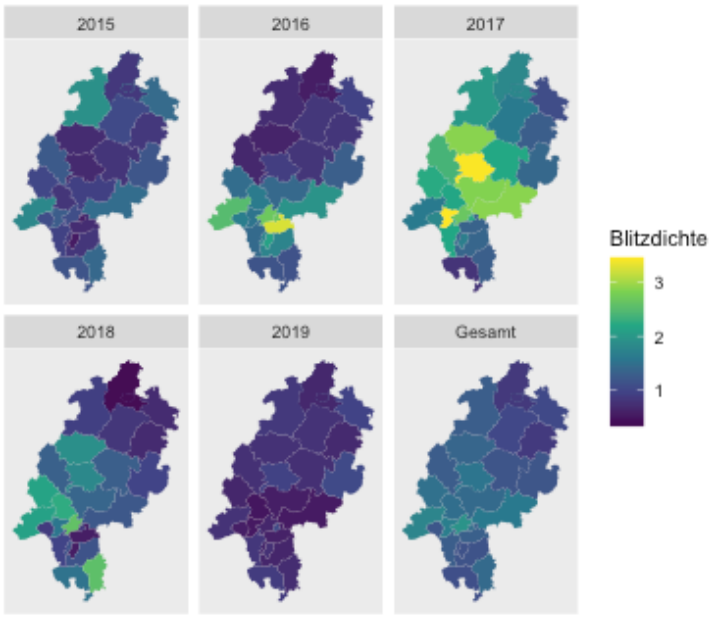


8.8 Aufgaben

1. Erstellen Sie eine Choroplethenkarte der Frankfurter Stadtteile, in der Sie die Anzahl bzw. die Dichte von Apotheken darstellen. (Schritte analog zu oben.)
2. Welche Stadtteile haben mehr Kioske? Welche mehr Apotheken? Wie ausgeprägt ist das Verhältnis? Erstellen Sie eine Karte, die das zum Ausdruck

bringt.

3. (Achtung, knifflig!) Siemens veröffentlicht einen Blitzatlas. Laden Sie den Datensatz herunter und bauen Sie die folgende Ansicht nach:



9 Text: Chandra 2014

9.1 Lesetext

Chandra, Vikram. 2014. *Geek Sublime: The Beauty of Code, the Code of Beauty.* Graywolf Press, Minneapolis.

Daraus:

- Kapitel 1: Hello, World! (S. 1–8)
- Kapitel 3: The Language of Logic (S. 19–40)

9.2 Fragen an den Text

1. Um welche Art von Text handelt es sich? Wer ist der Autor, und an wen wendet er sich?
2. Was ist “literate programming”, und was könnte das in R konkret bedeuten?
3. Auf S. 37 ist die Rede von “our journey down the stack of languages”. Was ist damit gemeint?

4. Was findet der Autor an Code und Computern so faszinierend? Wo können Sie das nachvollziehen, und wo nicht?

9.3 Themenfindung

- Bereiten Sie *ein* Thema vor, zu dem Sie sich vorstellen könnten im Sommer zu arbeiten.
- Achten Sie darauf, dass das Thema nicht zu allgemein ist ("Finanzmarkt") aber auch nicht zu speziell ("Zusammenhang zwischen Quadratmeterzahl und Mietpreis für Ladenflächen in Ginnheim").
- Es wird in einem nächsten Schritt darum gehen, interessante Datenquellen zu finden (vielleicht haben Sie schon eine Idee?) und einer Fragestellung näher zu kommen.
- Sie werden die Themen kurz vorstellen um Überschneidungen zu identifizieren und ggf. Gruppen zu bilden (wenn gewünscht).

10 HTML-Tabellen

10.1 Lernziele dieser Sitzung

Sie können…

- sich den Quellcode einer Webseite anzeigen lassen und interpretieren.
- HTML-Tabellen als Datensatz einlesen.
- Fortgeschrittene Methoden der Datenbereinigung nachvollziehen.

10.2 Vorbereitung

Am Beispiel der Küstenlängen verschiedener Länder besprechen wir Techniken der Datenerhebung/-erfassung und -visualisierung. Unser Ziel ist es, die Daten zu den Küstenlängen in einer Grafik darzustellen.

Für die folgenden Aufgaben benötigen wir die Pakete `rvest` und `tidyverse`. Zunächst müssen diese installiert und in unsere Umgebung geladen werden.

```
library(tidyverse)
library(rvest)
```

10.3 Datenbeschaffung

Auf dem Internetauftritt der CIA gab es es eine Tabelle, welche die Küstenlänge (inklusive der Inseln) der einzelnen Länder enthält.

Über die Archivierungsplattform WayBackMachine ist die Seite immer noch abrufbar: https://web.archive.org/web/20190802010710/https://www.cia.gov/library/publications/the-world-factbook/fields/282.html

In einem ersten Schritt wird die URL der Tabelle der Variable `url` zugewiesen, sodass der Quellcode mit dem Befehl `read_html()` eingelesen werden kann.

```
url <- "https://web.archive.org/web/20190802010710/https://www.cia.gov/library/publications/
reply <- read_html(url)
```

Der Befehl `html_table()` ermöglicht das Auslesen *aller* Tabellen auf der Seite. Mithilfe des Befehls `str()` sehen wir, dass die Seite genau eine Tabelle enthält, welche die Informationen zu den Küstenlängen enthält.

```
tables <- html_table(reply, fill = TRUE)
str(tables)
## List of 1
##  $ : tibble [266 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ Country  : chr [1:266] "Afghanistan" "Akrotiri" "Albania" "Algeria" ...
##   ..$ Coastline: chr [1:266] "0 km\n          (landlocked)" "56.3 km" "362 km" "998 km" .
```

Durch die Umformung zu einem tibble erhalten wir eine Tabelle mit den gewünschten Informationen:

```
as_tibble(tables[[1]])
## # A tibble: 266 x 2
##    Country             Coastline
##    <chr>               <chr>
##  1 Afghanistan         "0 km\n          (landlocked)"
##  2 Akrotiri            "56.3 km"
##  3 Albania             "362 km"
##  4 Algeria             "998 km"
##  5 American Samoa      "116 km"
##  6 Andorra             "0 km\n          (landlocked)"
##  7 Angola              "1,600 km"
##  8 Anguilla            "61 km"
##  9 Antarctica          "17,968 km"
## 10 Antigua and Barbuda "153 km"
## # ... with 256 more rows
```

Mit pipes können wir die obigen Befehle zusammenfassen und somit das Ganze auf einmal ausführen.

```
"https://web.archive.org/web/20190802010710/https://www.cia.gov/library/publications/the-wor
  read_html() %>%
  html_table(fill = T) %>%
  .[[1]] %>%
  as_tibble() -> coast
```

10.4 Datenformatierung

Zur Datenformatierung nutzen wir Funktionen aus dem Paket `stringr`. Die Spalte mit der Küstenlänge soll keinen Text, keine Einheit direkt hinter den Zahlenwerten und keine Kommata zur Trennung der Zahlenwerte enthalten.

Der Befehl "str_extract()" sucht nach vorgegebenen Mustern (engl. *patterns*) und wählt diese aus. Diese patterns werden auch reguläre Ausdrücke (*regular expressions / regex*) genannt und sind eigentlich ein Thema für sich. Das Pattern `[0-9,.]+ km` extrahiert die Kilometerangaben.

```
km <- str_extract(coast$Coastline, "[0-9,.]+ km")
```

Die ausgewählten Muster (in unserem Fall Kommata und Text) können durch den Befehl `str_replace_all()` gelöscht oder ersetzt werden. Wir ersetzen alle Zeichen *außer* Zahlen und Dezimalpunkt mit einem leeren String, so dass sie verschwinden.

```
str_replace_all(km, "[^0-9.]", "")
##   [1] "0"       "56.3"    "362"     "998"     "116"     "0"       "1600"
##   [8] "61"      "17968"   "153"     "45389"   "4989"    "0"       "68.5"
##  [15] "74.1"    "111866"  "25760"   "0"       "0"       "3542"    "161"
##  [22] "580"     "97"      "0"       "66.5"    "386"     "121"     "103"
##  [29] "0"       "0"       "20"      "0"       "29.6"    "7491"    "698"
##  [36] "80"      "161"     "354"     "0"       "1930"    "0"       "965"
##  [43] "443"     "402"     "202080"  "160"     "0"       "0"       "6435"
##  [50] "14500"   "138.9"   "11.1"    "26"      "3208"    "340"     "37"
##  [57] "169"     "120"     "3095"    "1290"    "515"     "5835"    "3735"
##  [64] "364"     "648"     "0"       "7314"    "27.5"    "314"     "148"
##  [71] "1288"    "2237"    "2450"    "307"     "296"     "2234"    "3794"
##  [78] "0"       "0"       "65992.9" "1288"    "1117"    "1129"    "1250"
##  [85] "4853"    "2525"    "28"      "885"     "80"      "40"      "310"
##  [92] "2389"    "539"     "12"      "13676"   "44087"   "121"     "125.5"
##  [99] "400"     "50"      "320"     "350"     "459"     "1771"    "101.9"
## [106] "0"       "823"     "733"     "6.4"     "0"       "4970"    "7000"
## [113] "66526"   "54716"   "2440"    "58"      "1448"    "160"     "273"
## [120] "7600"    "1022"    "124.1"   "29751"   "8"       "70"      "34"
## [127] "26"      "0"       "536"     "3"       "1143"    "2495"    "2413"
## [134] "0"       "499"     "0"       "0"       "498"     "225"     "0"
## [141] "579"     "1770"    "0"       "90"      "0"       "41"      "4828"
## [148] "0"       "4675"    "644"     "0"       "196.8"   "370.4"   "754"
## [155] "177"     "9330"    "6112"    "15"      "0"       "4.1"     "0"
## [162] "293.5"   "40"      "1835"    "2470"    "1572"    "30"      "8"
## [169] "0"       "451"     "2254"    "15134"   "910"     "0"       "853"
## [176] "64"      "32"      "0"       "1482"    "25148"   "2092"    "135663"
## [183] "1046"    "1519"    "14.5"    "2490"    "5152"    "518"     "0"
## [190] "2414"    "36289"   "51"      "440"     "1793"    "501"     "563"
## [197] "225"     "37653"   "0"       "60"      "135"     "158"     "58.9"
```

```
## [204] "120"     "84"      "403"     "0"       "209"     "2640"    "531"
## [211] "0"       "491"     "402"     "193"     "58.9"    "0"       "46.6"
## [218] "5313"    "3025"    "2798"    NA        "0"       "17968"   "4964"
## [225] "926"     "1340"    "853"     "386"     "3587"    "3218"    "0"
## [232] "193"     "1566.3"  "0"       "1424"    "3219"    "706"     "56"
## [239] "101"     "419"     "362"     "1148"    "7200"    "0"       "389"
## [246] "24"      "0"       "2782"    "1318"    "12429"   "19924"   "4.8"
## [253] "660"     "0"       "2528"    "2800"    "3444"    "188"     "19.3"
## [260] "129"     "0"       "1110"    "356000"  "1906"    "0"       "0"
```

Auch hier kann alles in einen Befehl gepackt werden:

```
coast$Coastline %>%
  str_extract("[0-9,.]+ km") %>%
  str_replace_all("[^0-9.]", "") %>%
  as.numeric() -> coast$coast_num
```

10.5 Datenaufbereitung

Mit dem Befehl `arrange()` kann die Tabelle sortiert werden. Zunächst auftseigend,

```
coast %>%
  arrange(coast_num)
## # A tibble: 266 x 3
##    Country     Coastline                                              coast_num
##    <chr>       <chr>                                                      <dbl>
##  1 Afghanistan "0 km\n          (landlocked)"                                 0
##  2 Andorra     "0 km\n          (landlocked)"                                 0
##  3 Armenia     "0 km\n          (landlocked)"                                 0
##  4 Austria     "0 km\n          (landlocked)"                                 0
##  5 Azerbaijan  "0 km\n          (landlocked); note - Azerbaijan borde~         0
##  6 Belarus     "0 km\n          (landlocked)"                                 0
##  7 Bhutan      "0 km\n          (landlocked)"                                 0
##  8 Bolivia     "0 km\n          (landlocked)"                                 0
##  9 Botswana    "0 km\n          (landlocked)"                                 0
## 10 Burkina Fa~ "0 km\n          (landlocked)"                                 0
## # ... with 256 more rows
```

und schließlich absteigend, sodass die größten Werte an erster Stelle stehen.

```
coast %>%
  arrange(desc(coast_num))
## # A tibble: 266 x 3
##    Country     Coastline                                              coast_num
##    <chr>       <chr>                                                      <dbl>
##  1 World       "356,000 km\n          \n          \n        \n\n  \n~   356000
```

```
##  2 Canada       "202,080 km\n          \n          \n        \n\n  \n~   202080
##  3 Pacific Oce~ "135,663 km"                                          135663
##  4 Atlantic Oc~ "111,866 km"                                          111866
##  5 Indian Ocean "66,526 km"                                            66526
##  6 European Un~ "65,992.9 km"                                          65993.
##  7 Indonesia    "54,716 km"                                            54716
##  8 Arctic Ocean "45,389 km"                                            45389
##  9 Greenland    "44,087 km"                                            44087
## 10 Russia       "37,653 km"                                            37653
## # ... with 256 more rows
```

Bevor wir jedoch eine vollständig sortierte Liste haben, muss der Datensatz noch von falschen Einträgen gesäubert werden. Dafür benutzen wir den Befehl `filter()`. Wir suchen wieder nach einem bestimmten Muster (hier zum Beispiel dem Wort "Ocean") und filtern es aus dem Datensatz.

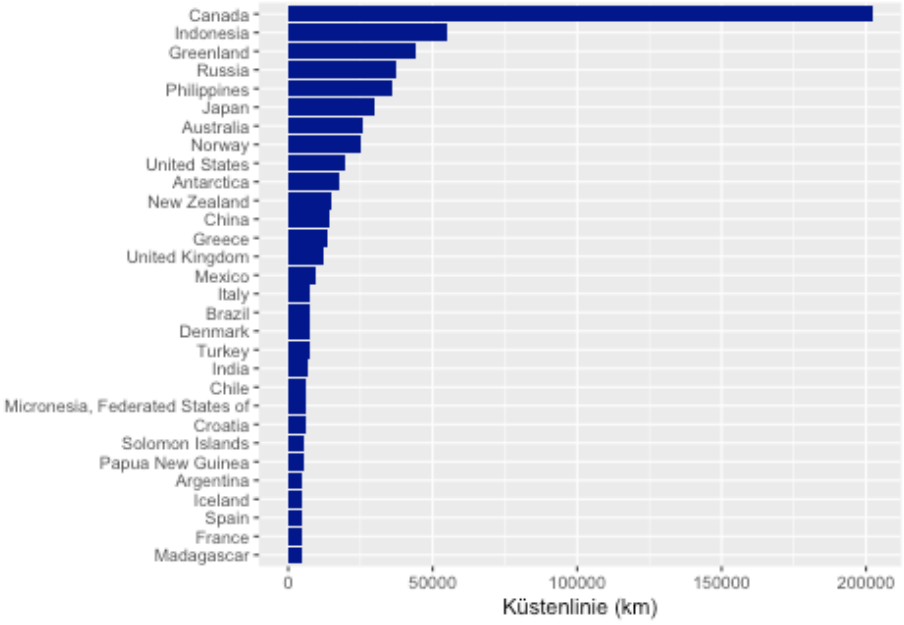
Die Grafik soll nur aus den ersten 30 Einträgen der Tabelle bestehen, welche uns der Befehl `head()` ausgibt.

```
coast %>%
  arrange(desc(coast_num)) %>%
  filter(!str_detect(Country, "Ocean")) %>%
  filter(!Country %in% c("World", "European Union")) %>%
  head(30) -> top_30
```

10.6 Datenvisualisierung

Das Balkendiagramm erhalten wir durch den "ggplot" Befehl. Hierbei gibt es verschiedenste Einstellmöglichkeiten. Wichitg sind vor allem die Angabe des verwendeten Datensatzes und die Art der Grafik (ob Kartendarstellung oder Balkendiagramm). Desweiteren kann man noch Farben der Eigenschaften, eine Achsenbeschriftung u. v. m. bestimmen.

```
ggplot(top_30, aes(x = reorder(Country, coast_num), y=coast_num)) +
  geom_bar(stat='identity', fill="darkblue") +
  coord_flip() +
  scale_x_discrete(NULL) +
  scale_y_continuous("Küstenlinie (km)")
```



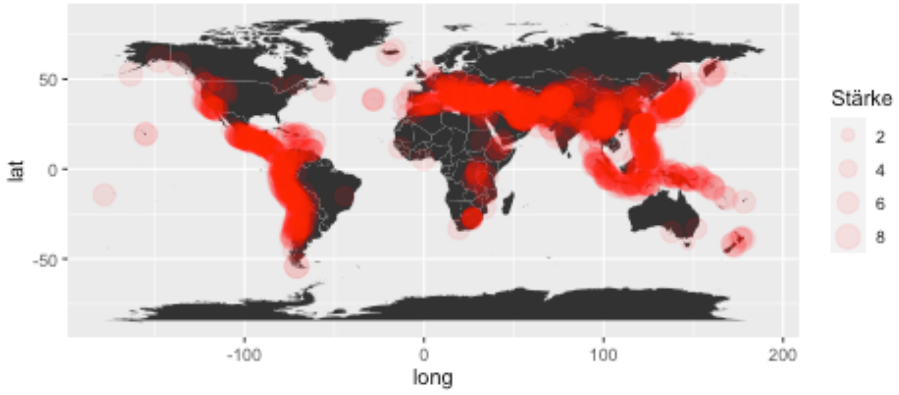
10.7 Aufgaben

1. Importieren Sie die Daten zu den tödlichen Erdbeben auf Wikipedia und formen sie diese zu einem tibble um.

```
"https://en.wikipedia.org/wiki/List_of_deadly_earthquakes_since_1900" %>%
  read_html %>%
  html_table(fill = T) %>%
  .[[5]] %>%
  as.tibble() -> earthquakes_raw
```

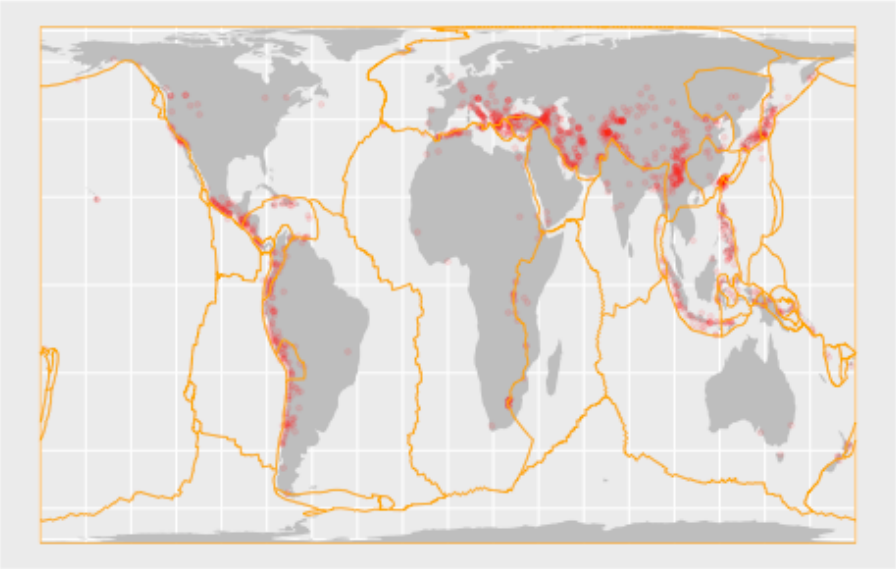
2. Erstellen Sie mit den erhaltenen Daten eine Karte, welche die Lage und die Stärke der Erdbeben angibt:

```
earthquakes_raw %>%
  mutate(Lat = as.numeric(Lat), Long = as.numeric(Long)) %>%
  mutate(magnitude_num = as.numeric(str_extract(Magnitude, "[0-9.]+"))) -> earthquakes

ggplot() +
  geom_polygon(data = map_data("world"), aes(x = long, y = lat, group = group)) +
  geom_point(data = earthquakes,
             aes(x = Long, y = Lat, size = magnitude_num),
             color = "red", alpha = 0.1) +
  coord_quickmap() +
  scale_size_area("Stärke")
```



3. Wandeln Sie den Erdbeben-Datensatz in das Simple Features Format um. Laden Sie zusätzlich eine Weltkarte mit dem Paket `rnaturalearth` und wandeln Sie auch diese in Simple Features um. Finden Sie außerdem einen Geodatensatz zu tektonischen Platten. Visualiseren Sie alles auf einer Welktarte (Projektion: Gall-Peters).

```
library(sf)

earthquakes %>%
  filter(! is.na(Long)) %>%
  st_as_sf(coords=c("Long", "Lat")) %>%
  st_set_crs(4326) -> quakesf

library(rnaturalearth)

ne_download(type="land", category = "physical") %>%
  st_as_sf() %>%
  st_transform('+proj=cea +lon_0=0 +x_0=0 +y_0=0 +lat_ts=45 +ellps=WGS84 +datum=WGS84 +units

st_read("https://raw.githubusercontent.com/fraxen/tectonicplates/master/GeoJSON/PB2002_plate

ggplot() +
  geom_sf(data = earthsf, fill = "gray", color = NA) +
  geom_sf(size = 1, data = quakesf, color = "red", alpha = 0.1) +
  geom_sf(data = plates, color = "orange", fill = NA, lwd = 0.3)
```



11 Web scraping

11.1 Lernziele dieser Sitzung

Sie können…

- HTML in seiner Grundstruktur interpretieren.
- gezielt einzelne Elemente einer Seite mit R auslesen.

11.2 Vorbereitung

Für diese Lektion werden folgende Pakete benötigt:

```
library(tidyverse)
library(rvest)
```

11.3 Exkurs: HTML

Wenn man eine Webseite ganz normal in einem Browser aufruft, erscheint sie als eine Mischung aus formatiertem Text, Bildern, Designelementen, ggf. Videos, usw. Was aber im Hintergrund eigentlich vom Server an den Browser übertragen wird, ist eine Textdatei in einem bestimmten Format – HTML (Hyptertext Markup Language). Darin wird Text auf eine genau festgelegte Art und Weise annotiert, damit der Browser weiß, wie er ihn anzeigen soll. Im HTML-Dokument

kann auch stehen: Lade ein Bild von einer bestimmten Stelle und zeig es an dieser Stelle an.

Einen brauchbaren Überblick über die HTML-Elemente und die Struktur einer HTML-Datei gibt es hier: https://www.tutorialspoint.com/de/html/

An dieser Stelle ist wichtig ist zu wissen: HTML-Elemente ("Tags" oder "Nodes") sind streng hierarchisch angeordnet. Sie bestehen oft aus einem Anfangs- und einem End-Tag in spitzen Klammern:

```
<html>
  <head>
    <title>Titel meiner Webseite</title>
  </head>
  <body>
    <h1>Überschrift</h1>
    <p>Erster Absatz mit <b>fettem Text</b></p>
    <p>Zweiter Absatz mit <i>kursivem Text</i></p>
    <img src="path/to/image.jpg alt="Ein Bild" />
  </body>
</html>
```

In diesem Beispiel ist das Bild mit `<img />` das einzige Element, das nicht geöffnet und wieder geschlossen wird. Außerdem hat dieses Element Attribute (`src` und `alt`) mit bestimmten Werten. Eine echte Webseite ist weitaus komplexer und unübersichtlicher.

Dem Browser kann man sagen: Zeig mir nicht wie üblich die „gerenderte" Seite an, sondern die zu Grunde liegende HTML-Datei. Das geht mit „Quelltext anzeigen" / „View Source" o.ä.

Viele Browser (hier seien Chrome und Firefox empfohlen) haben auch einen Modus namens „Entwicklertools" / „Developer tools", in dem die HTML-Elemente hierarchisch geordnet sind.

11.4 Seite laden

Beim so genannten Web Scraping ist die Grundidee, dass wir eine Webseite nicht im Browser öffnen, sondern den HTML-Quelltext direkt in R laden. R kann dann aus dem Quelltext bestimmte Elemente extrahieren.

In der letzten Sitzung haben wir schon gesehen, wie Tabellen nach genau diesem Prinzip von einer Webseite direkt in R geladen werden können. Jetzt soll es darum gehen, noch präziser zu sagen, welche Elemente wir von einer bestimmten Webseite ziehen wollen.

Als Beispiel soll die Infoseite eines Wohnheims des Studentenwerks Frankfurt dienen. Die Adresse ist: https://www.studentenwerkfrankfurt.de/wohnen/wohnheime/frankfurt-am-main/kleine-seestrasse-11

Zunächst laden wir den Quelltext in R und nennen ihn `quelltext`:

```
quelltext <- read_html("https://www.studentenwerkfrankfurt.de/wohnen/wohnheime/frankfurt-am-

quelltext
## {html_document}
## <html lang="de" dir="ltr" class="no-js">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body id="p187" class="page-187 pagelevel-4 language-0 backendlayout-wohn ...
```

In diesem Schritt hat R den HTML-Quelltext schon "geparsed", d.h. ihn nicht nur als Text gespeichert, sondern als hierarchische Konstruktion mit den beiden Grundelementen `head` und `html`.

11.5 Elemente suchen

Uns soll jetzt das Baujahr interessieren. Auf der Seite stehen die Informationen rechts neben dem Bild. Mit den Entwicklertools können wir schauen, wie die Elemente in HTML genau heißen. Ein geeigneter Ausgangspunkt wäre das `<div>`-Element mit dem Attribut `id="c599"`.

In R können wir dieses einzelne Element ansprechen mit:

```
quelltext %>%
  html_node("div#c599")
## {html_node}
## <div id="c599" class="frame frame-default frame-type-text frame-layout-0 frame-backgroun
## [1] <div class="frame-container"><div class="frame-inner">\n<p>Kleine Seestra ...
```

Dann gehen wir in der Hierarchie drei `<div>`- Elemente „tiefer". (`<div>`- Elemente sind abstrakte Container und werden im Webdesign oft angewendet.)

```
quelltext %>%
  html_node("div#c599") %>%
  html_node("div") %>%
  html_node("div")
## {html_node}
## <div class="frame-inner">
## [1] <p>Kleine Seestraße 11<br>60486 Frankfurt am Main\r</p>\n
## [2] <p>25 Wohnheimplätze</p>\n
## [3] <ul class="list-normal">\n<li>5 Wohnküchen</li>\n<li>Wintergärten</li>\n< ...
## [4] <p>Baujahr<strong> </strong>1995\r</p>\n
## [5] <p><strong></strong></p>
```

Alternativ könnten wir auch sagen: Darin das `div` mit `class="frame-inner"`:

```
quelltext %>%
  html_node("div#c599") %>%
  html_node("div.frame-inner")
```

```
## {html_node}
## <div class="frame-inner">
## [1] <p>Kleine Seestraße 11<br>60486 Frankfurt am Main\r</p>\n
## [2] <p>25 Wohnheimplätze</p>\n
## [3] <ul class="list-normal">\n<li>5 Wohnküchen</li>\n<li>Wintergärten</li>\n< ...
## [4] <p>Baujahr<strong> </strong>1995\r</p>\n
## [5] <p><strong></strong></p>
```

mit `html_nodes()` (Mehrzahl) werden alle Unterelemente eines Typs (hier `<p>` = Paragraph) angesprochen. Davon dann den Textinhalt (`html_text()`) gibt uns die relevanten Informationen:

```
quelltext %>%
  html_node("div#c599") %>%
  html_node("div.frame-inner") %>%
  html_nodes("p") %>%
  html_text()
## [1] "Kleine Seestraße 1160486 Frankfurt am Main\r"
## [2] "25 Wohnheimplätze"
## [3] "Baujahr 1995\r"
## [4] ""
```

11.6 Elemente reinigen

Jetzt ließe sich der dritte Eintrag dieses Ergebnisvectors reinigen und als Ergebnis "speichern".

Der Befehl `str_extract()` wendet dabei einen Regulären Ausdruck an, der nach einer Folge von vier Zahlen sucht.

```
quelltext %>%
  html_node("div#c599") %>%
  html_node("div.frame-inner") %>%
  html_nodes("p") %>%
  html_text() %>%
  .[3] %>%
  str_extract("[0-9]{4}") %>%
  as.numeric() -> baujahr

baujahr
## [1] 1995
```

Wie diese Technik automatisiert auf eine Reihe von Seiten angewendet werden kann, wird zu einem späteren Zeitpunkt besprochen.

11.7 Aufgaben

1. Lesen Sie die Anzahl der Wohnheimplätze aus.

```
quelltext
## {html_document}
## <html lang="de" dir="ltr" class="no-js">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body id="p187" class="page-187 pagelevel-4 language-0 backendlayout-wohn ...
```

```
quelltext %>%
  html_node("div#c599") %>%
  html_node("div.frame-inner") %>%
  html_nodes("p") %>%
  html_text() %>%
  .[2] %>%
  str_extract("[0-9]+") %>%
  as.numeric() -> anzahl_plaetze

anzahl_plaetze
## [1] 25
```

2. Lesen Sie Baujahr und Anzahl der Wohnheimplätze von einem anderen Wohnheim aus. Was muss angepasst werden?

```
# z.B.: https://www.studentenwerkfrankfurt.de/wohnen/wohnheime/wiesbaden/max-kade-haus-adol

"https://www.studentenwerkfrankfurt.de/wohnen/wohnheime/wiesbaden/max-kade-haus-adolfsallee-
  read_html() -> quelltext_wi

# Baujahr nicht vorhanden!
baujahr_wi <- NA

quelltext_wi %>%
  html_node("div#c1563") %>%
  # Auf dieser Seite ist es eine andere div-ID!
  # Außerdem: Appartment = Platz? Scheint aber so zu sein...
  html_node("div.frame-inner") %>%
  html_nodes("p") %>%
  html_text() %>%
  .[2] %>%
  str_extract("[0-9]+") %>%
  as.numeric() -> anzahl_plaetze_wi

anzahl_plaetze_wi
## [1] 87
```

3. Ändern Sie das Script so, dass es auf beiden (allen) Wohnheimseiten funk-

tioniert.

```
# Hier kann (hoffentlich) auch jede andere Wohnheim-URL eingesetzt werden.
"https://www.studentenwerkfrankfurt.de/wohnen/wohnheime/frankfurt-am-main/kleine-seestrasse-
  read_html() -> quelltext_x

quelltext_x %>%
  # So funktioniert es unabhänging von ID:
  html_node("div.wohnheim-2 > div:nth-child(2)") %>%
  html_nodes("p") %>%
  html_text() -> items

items[3] %>%
  str_extract("[0-9]{4}") %>%
  as.numeric() -> baujahr_x

items[2] %>%
  # Funktioniert leider nicht bei der getrennten Angabe mehrerer Kategorien:
  str_extract("[0-9]+") %>%
  as.numeric() -> anzahl_plaetze_x
```

4. Lesen Sie die Adresse eines Wohnheims aus. Speichern Sie dabei Straße, Hausnummer, Postleitzahl und Ort getrennt.

```
# Wie in Aufgabe 5 ersichtlich lassen sich die beiden Adresszeilen eigentlich
# recht einfach aus der Übersichtsseite auslesen.

# Eine Schwierigkeit beim Auslesen aus der Einzelseite liegt darin, dass die
# Funktion html_text() den Zeilenumbruch <br> "verschluckt". Die neueste Version
# von rvest beinhaltet die Funktion html_text2(), die genau dieses Probem löst.

# Die neue Version kann installiert werden mit:
# install.packages("devtools")
# devtools::install_github("tidyverse/rvest")
# (Danach R neu starten)

quelltext_x %>%
  html_node("div.wohnheim-2 > div:nth-child(2)") %>%
  html_nodes("p") %>%
  html_text2() %>%
  .[1] %>%
  trimws() %>%
  str_split("\n") %>%
  .[[1]] -> adresse

# Die Hausnummer sind die Zahlen am Ende der ersten Zeile:
adresse[1] %>%
```

```
  str_extract("[-0-9]+$") -> hausnummer # Funktioniert aber nicht bei 1b u.ä...

# Die Straße ist der Rest:
adresse[1] %>%
  str_remove(" [0-9]+$") -> strasse

# Die PLZ sind die 5 Zahlen am Anfang der zweiten Zeile:
adresse[2] %>%
  str_extract("^[0-9]{5}") -> plz

# Der Ort ist der Rest:
adresse[2] %>%
  str_remove("^[0-9]{5} ") -> ort
```

5. Sammeln Sie eine Liste aller Wohnheime mit Link.

```
"https://www.studentenwerkfrankfurt.de/wohnen/wohnheime" %>%
  read_html() %>%
  html_nodes("div.thumbnail") -> items

items %>%
  html_node("a") %>%
  html_attr("href") -> links

items %>%
  html_node("h4") %>%
  html_text -> strasse_nr

items %>%
  html_node("p") %>%
  html_text() -> plz_ort

tibble(links, strasse_nr, plz_ort) -> index
```

6. Sammeln Sie einen Datensatz (tibble) aller "Nutzungsentgelte" mit Wohnheim, Baujahr, Anzahl Wohneinheiten und Adresse. Stellen Sie sich vor, es handelte sich um Tausende Wohnheime – vermeiden Sie also Copy-Paste-Strategien.

```
# Hierfür gibt es einige Strategien. Meine präferierte Variante erfordert
# zunächst eine eigene Funktion, die eine URL nimmt und die gewünschten
# Informationen ausgibt:

scrape <- function(url) {
  url %>%
    read_html -> quelltext
```

```
  quelltext %>%
    html_table() %>%
    last() %>%
    # Die Größe soll immer ein String sein, weil es sonst später Probeme gibt:
    mutate(`Größe m²` = as.character(`Größe m²`)) -> Nutzungsentgelte

  quelltext %>%
    html_node("div.wohnheim-2 > div:nth-child(2)") %>%
    html_nodes("p") -> items

  items[[1]] %>%
    html_text2() %>%
    trimws() %>%
    str_split("\n") %>%
    .[[1]] -> adresse

  Nutzungsentgelte$strasse <- adresse[1]
  Nutzungsentgelte$ort <- adresse[2]

  items[3] %>%
    html_text() %>%
    str_extract("[0-9]{4}") %>%
    as.numeric() -> Nutzungsentgelte$baujahr

  items[2] %>%
    html_text() %>%
    str_extract("[0-9]+") %>%
    as.numeric() -> Nutzungsentgelte$anzahl_plaetze

  return(Nutzungsentgelte)
}

# Dann kann ich die eigene Funktion "scrape" auf alle Links anwenden. Das
# Ergebnis ist eine Liste von Tibbles. Die lässt sich schließlich noch
# kombinieren:

index$links %>%
 paste0("https://www.studentenwerkfrankfurt.de", .) %>%
  map(scrape) %>%
  do.call(bind_rows, .) -> liste

liste
## # A tibble: 133 x 8
##    Art     `Größe m²` Ausstattung   `EUR*` strasse ort   baujahr anzahl_plaetze
##    <chr>   <chr>      <chr>         <chr>  <chr>   <chr>   <dbl>          <dbl>
```

```
##  1 "Einze~ "16"       "unmöbliert,~ "250,0~ Beetho~ 6032~    1962          37
##  2 "Einzi~ "18-20"    "unmöbliert,~ "311,0~ Beetho~ 6032~    1962          37
##  3 "Einzi~ "24"       "barrierefre~ "295,0~ Beetho~ 6032~    1962          37
##  4 ""      ""         ""            ""      Beetho~ 6032~    1962          37
##  5 "Einze~ "9"        "möbliert, S~ "207,0~ Bocken~ 6032~    1956          81
##  6 "Einze~ "15"       "unmöbliert,~ "236,0~ Bocken~ 6032~    1956          81
##  7 "Zweiz~ "45"       "unmöbliert,~ "Beleg~ Bocken~ 6032~    1956          81
##  8 "Einze~ "11 - 17"  "unmöbliert,~ "256,0~ Fröbel~ 6048~    1960          36
##  9 "Einze~ "11"       "möbliert, W~ "266,0~ Fröbel~ 6048~    1960          36
## 10 "Einzi~ "23"       "unmöbliert,~ "315,0~ Fröbel~ 6048~    1960          36
## # ... with 123 more rows
```

12 Text: Straube 2021

12.1 Lesetexte

Straube, Till. 2021. Datenbeschaffung. In: Tabea Bork-Hüffer, Henning Füller und Till Straube (Hrsg). *Handbuch Digitale Geographien.* Stuttgart: UTB.

Bauß, Jan-Luca und Felix Hiemeyer. 2021. Research Puzzle: Datenbeschaffung durch Web Scraping. In: Tabea Bork-Hüffer, Henning Füller und Till Straube (Hrsg). *Handbuch Digitale Geographien.* Stuttgart: UTB.

12.2 Fragen an den Text

1. Um was für eine Art von Text handelt es sich? An wen wenden sich die Autoren?
2. Welche Momente werden im Research Puzzle beschrieben, in denen die Autoren ihr Vorhaben geändert haben?
3. Wie verändert Datenbeschaffung (statt -erhebung) den wissenschaftlichen Prozess?
4. Welche Beispiele gibt es im Rahmen Ihres Projektvorhabens für…
   - offene Daten?
   - Daten, die sich durch web scraping abrufen lassen?
   - Daten, die sich über (öffentliche oder private) APIs abrufen lassen?

13 Präsentationen

Zum Semesterabschluss präsentieren die Gruppen Ihre Projektvorhaben und erhalten Feedback.

14 APIs

14.1 Vorbereitung

Für diese Lektion werden die Pakete benötigt:

```
library(tidyverse)
library(jsonlite)
```

14.2 SWAPI

Die Star Wars API ist eine eigens für Übungszwecke eingerichtete API, und steht uns deshalb (anders als andere APIs) ohne Login zur Verfügung. Wir sollten bei der Benutzung darauf achten, sie nicht zu überladen.

Jede gute API kommt mit einer ausführlichen Dokumentation in der die Endpunkte und Abfrageoptionen erklärt sind.

Bei den hier besprochenen REST-APIs geht es eigentlich nur darum, die richtige Abfrage **als URL** zu formulieren. Die Antwort des Servers gibt uns dann die Daten, die wir brauchen, und zwar üblicherweise im JSON-Format.

Zum Beispiel fragen wir so Informationen über Han Solo ab:

```
han_solo <- read_json("https://www.swapi.tech/api/people/14/")$result
```

Die Antwort ist eine Liste, deren Elemente sich wie gewohnt mit `$` ansprechen lassen und wiederum Hinweise auf API-Abfragen enthalten können:

```
han_solo$properties$eye_color
## [1] "brown"
han_solo$properties$homeworld
## [1] "https://www.swapi.tech/api/planets/22"
```

Diese Information ließe sich wiederum abfragen durch:

```
han_solo$properties$homeworld %>%
  read_json() %>%
  .$result %>%
  .$properties %>%
  .$name
## [1] "Corellia"
```

Eine (recht willkürlich gewählte) Herausforderung wäre es nun, die Namen aller Charaktere herauszufinden, die in *Return of the Jedi* vorkommen.

Zunächst können wir den richtigen Film suchen mit:

```
read_json("https://www.swapi.tech/api/films/")$result %>%
  map("properties") %>%
  map("title")
## [[1]]
## [1] "A New Hope"
```

```
##
## [[2]]
## [1] "The Empire Strikes Back"
##
## [[3]]
## [1] "Return of the Jedi"
##
## [[4]]
## [1] "The Phantom Menace"
##
## [[5]]
## [1] "Attack of the Clones"
##
## [[6]]
## [1] "Revenge of the Sith"
```

Dann lässt sich die gewünschte Liste ziehen mit:

```
read_json("https://www.swapi.tech/api/films/3")$result$properties$characters -> return_chara
```

Für jeden dieser Charaktere ließe sich der Name herausfinden mit

```
read_json("https://www.swapi.tech/api/people/1/")$result$properties$name
## [1] "Luke Skywalker"
read_json("https://www.swapi.tech/api/people/4/")$result$properties$name
## [1] "Darth Vader"
# usw.
```

Können wir aber auch die Namen nicht einzeln, sondern automatisch Abfragen?

14.3 Exkurs: Funktionen schreiben

Funktionen sind überall in R. Funktionen haben eine Eingabe (parameters) und eine Ausgabe (return values). Z.B. hat die Funktion `mean()` als Eingabe einen numerischen Vektor, und als Ausgabe das arithmetische Mittel dieses Vektors:

```
data(diamonds)
mean(diamonds$carat)
## [1] 0.7979397
```

Wir können auch eigene Funktionen schreiben. Die Definition einer eigenen Funktionen hat immer diese Form:

```
FUNKTIONSNAME <- function(EINGABE) {
  ...
  AUSGABE
}
```

Wenn es die Funktion `mean()` nicht gäbe, könnten wir sie (bzw. so etwas ähnliches) selbst schreiben, mit:

```
my_mean <- function(verteilung) {
  sum(verteilung) / length(verteilung)
}
```

Wenn wir die Definition ausführen, erscheint die Funktion in unserem Environment, genauso wie andere Objekte. Wir können sie dann genauso anwenden wie andere Funktionen:

```
my_mean(diamonds$carat)
## [1] 0.7979397
my_mean(diamonds$depth)
## [1] 61.7494
```

14.4 Abfragefunktion

Eine Funktion zur automatischen Abfrage der Charakternamen bräuchte als Eingabe die URL der API, und als Ausgabe den Charakternamen. Eigentlich geht es nur um eine *Abstraktion* des *konkreten* Befehls:

```
read_json("https://www.swapi.tech/api/people/1/")$result$properties$name
## [1] "Luke Skywalker"
```

Die Definition der Funktion könnte so aussehen:

```
get_character_name <- function(url) {
  read_json(url)$result$properties$name
}
```

Testweise lässt sie sich anwenden:

```
get_character_name("https://www.swapi.tech/api/people/1/")
## [1] "Luke Skywalker"
```

Leider lässt sie sich nicht so einfach (wie andere Funktionen) auf den Vektor von URLS anwenden, da die Funktion keinen Vektor als Eingabe erwartet:

Abhilfe schafft der Befehl `map()` aus dem `purrr`-Paket (Teil von `tidyverse`). Hier lässt sich ein Vektor (oder eine Liste) angeben, sowie der Name einer Funktion, die dann *auf jedes Element des Vektors* angewendet wird:

```
map(return_characters, get_character_name)
```

Resultat ist eine Liste, die sich mit `unlist()` auch zu einem Vektor wandeln ließe… oder man benutzt direkt die Abwandlung `map_chr()`, die nach Möglichkeit immer einen character vector ausgibt:

```
map_chr(return_characters, get_character_name)
##  [1] "Luke Skywalker"        "C-3PO"                 "R2-D2"
```

```
##  [4] "Darth Vader"           "Leia Organa"           "Obi-Wan Kenobi"
##  [7] "Chewbacca"             "Han Solo"              "Jabba Desilijic Tiure"
## [10] "Wedge Antilles"        "Yoda"                  "Palpatine"
## [13] "Boba Fett"             "Lando Calrissian"      "Ackbar"
## [16] "Mon Mothma"            "Arvel Crynyd"          "Wicket Systri Warrick"
## [19] "Nien Nunb"             "Bib Fortuna"
```

15 Serialisierung

15.1 Vorbereitung

Für diese Lektion werden die Pakete benötigt:

```
library(tidyverse)
library(rvest)
```

15.2 Zielsetzung

Auf https://www.wg-gesucht.de/ finden sich Anzeigen für WGs. In der Listenansicht werden pro Seite 20 Angebote überblicksartig angezeigt. Bei Click auf ein Angebot erscheinen Details der Anzeige, für die wir uns als Rohdaten interessieren.

Wir wollen…

1. … für die ersten drei Überblicksseiten automatisiert alle URLs der einzelnen Anzeigen auslesen (was sich aber in der Praxis erweitern ließe).
2. … für diese 60 URLs automatisiert die folgenden Details auslesen

- Gesamtmiete
- Zimmergröße
- Wer wohnt dort?

Beide Zielsetzungen können in folgende Schritte unterteilt werden:

1. An einem Beispiel konkret ausführen
2. Abstrahieren (hier: als Funktion)
3. Testen an weiteren einzelfällen
4. Serialisiert ausführen (hier: mit `map` o.ä.)

15.3 URLs der Anzeigen auslesen

15.3.1 Schritt 1: An einem Beispiel konkret ausführen

In Sitzung 11 haben wir gelernt, wie mit dem Paket `rvest` HTML-Seiten in R geladen und einzelne Elemente angesporchen werden können.

Wenn man sich die erste Listenansicht genau anschaut (mit Developer Tools / Inspect Element vom Browser), wird deutlich, dass die `<tr>`-Tags

mit `class=offer_list_item` die einzelnen Anzeigen enthalten. Ebenfalls im `<tr>`-Element enthält das Attribut `adid` den entscheidenden Teil der Anzeigen-URL.

Deshalb können wir schreiben:

```
"https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main.41.0.0.0.html" %>%
  read_html() %>%
  html_nodes("tr.offer_list_item") %>%
  html_attr("adid")
##  [1] "wg-zimmer-in-Frankfurt-am-Main-Nordend-West.6936308.html"
##  [2] "wg-zimmer-in-Frankfurt-am-Main-Nordend-West.6944572.html"
##  [3] "wg-zimmer-in-Frankfurt-am-Main-Westend-Nord.7952174.html"
##  [4] "wg-zimmer-in-Frankfurt-am-Main-Bornheim--Ostend.8718668.html"
##  [5] "wg-zimmer-in-Frankfurt-am-Main-Westend-Nord.7697498.html"
##  [6] "wg-zimmer-in-Frankfurt-am-Main-Dornbusch.8713209.html"
##  [7] "wg-zimmer-in-Frankfurt-am-Main-Bockenheim.8563767.html"
##  [8] "wg-zimmer-in-Frankfurt-am-Main-Westend-Nord.7695069.html"
##  [9] "wg-zimmer-in-Frankfurt-am-Main-Innenstadt.8181981.html"
## [10] "wg-zimmer-in-Frankfurt-am-Main-Bockenheim.8545727.html"
## [11] "wg-zimmer-in-Frankfurt-am-Main-Westend-Nord.8676840.html"
## [12] "wg-zimmer-in-Frankfurt-am-Main-Innenstadt.8704699.html"
## [13] "wg-zimmer-in-Frankfurt-am-Main-Westend-Nord.8523043.html"
## [14] "wg-zimmer-in-Frankfurt-am-Main-Westend-Nord.8555199.html"
## [15] "wg-zimmer-in-Frankfurt-am-Main-Bockenheim.6348718.html"
## [16] "wg-zimmer-in-Frankfurt-am-Main-Nordend-West.8369804.html"
## [17] "wg-zimmer-in-Frankfurt-am-Main-Innenstadt.8196826.html"
## [18] "wg-zimmer-in-Frankfurt-am-Main-Westend-Nord.8395290.html"
## [19] "wg-zimmer-in-Frankfurt-am-Main-Innenstadt.8376022.html"
## [20] "wg-zimmer-in-Frankfurt-am-Main-Westend-Nord.8376129.html"
```

15.3.2 Schritt 2: Abstrahieren

Der obige Code lässt sich als Funktion abstrahieren, die eine URL als Input hat (s. Sitzung 16):

```
get_url_list <- function(list_url) {
  "https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main.41.0.0.0.html" %>%
    read_html() %>%
    html_nodes("tr.offer_list_item") %>%
    html_attr("adid")
}
```

15.3.3 Schritt 3: Testen

15.3.4 Schritt 4: Serialisiert ausführen

Schließlich können wir die Funktion auf eine Reihe von Inputs anwenden. Einen Vektor mit den gewünschten Input-URLs können wir erstellen mit:

```
paste0("https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main.41.0.0.",0:4,".html")
## [1] "https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main.41.0.0.0.html"
## [2] "https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main.41.0.0.1.html"
## [3] "https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main.41.0.0.2.html"
## [4] "https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main.41.0.0.3.html"
## [5] "https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main.41.0.0.4.html"
```

Diese Liste ließe sich natürlich erweitern.

Mit `map()` aus dem `purrr`-Paket (Teil von `tidyverse`) lässt sich dann unsere Funktion `get_url_list()` auf alle Elemente dieses Vektors anwenden. Das vorläufige Resultat ist eine Liste der Länge 3, wobei jedes Element wiederum ein Vektor mit 20 Elementen ist.

Der Befehl `map_chr()` dampft das Ergebnis dann direkt auf einen einfachen Character-Vektor ein.

Am Ende wird das Resultat mit `paste0()` an den Domainnamen gehängt und dem Objektnamen `anzeige_urls` zugewiesen.

```
paste0("https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main.41.0.0.",0:2,".html") %>%
  map(get_url_list) %>%
  flatten_chr() %>%
  paste0("https://www.wg-gesucht.de/", .) -> anzeige_urls

str(anzeige_urls)
##  chr [1:60] "https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main-Nordend-West.693630
```

15.4 Informationen der Anzeigen auslesen

Jetzt ginge es darum, für jede dieser 60 URLs die relevanten Informationen rauszusuchen:

- Gesamtmiete
- Zimmergröße
- Wer wohnt dort?

Auch hier gehen wir für die Automatisierung in den drei Schritten vor.

15.4.1 Schritt 1: An einem Beispiel konkret ausführen

Zunächst eine Beispielanzeige laden und zwischenspeichern:

```
"https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main-Ginnheim.7659434.html" %>%
  read_html() -> site
```

Dann lassen sich Quadratmeterzahl und Gesamtmiete recht einfach auslesen, weil beide in einer `h2`-Überschrift mit `class="headline-key-facts"` stecken:

```
site %>%
  html_nodes("h2.headline-key-facts") %>%
  html_text() %>%
  trimws() -> key_facts
  qm <- key_facts[1]
  eur <- key_facts[2]

qm
## [1] "26m²"
eur
## [1] "520€"
```

Die Information, wer dort wohnt, steckt in einem span title:

```
site %>%
  html_node("h1#sliderTopTitle") %>%
  html_nodes("span")%>%
  .[[2]] %>%
  html_attr("title") -> bewohnerinnen

bewohnerinnen
## [1] "2er WG (0w,1m,0d)"
```

Geballt lassen sich die Daten für eine Anzeige so ausgeben:

```
c(qm, eur, bewohnerinnen)
## [1] "26m²"              "520€"              "2er WG (0w,1m,0d)"
```

15.4.2 Schritt 2: Abstrahieren

Wir abstrahieren die obigen Schritte als Funktion:

```
get_anzeige_details <- function(anzeige_url) {

  Sys.sleep(2)

  anzeige_url %>%
    read_html() -> site

  site %>%
    html_nodes("h2.headline-key-facts") %>%
    html_text() %>%
    trimws() -> key_facts
    qm <- key_facts[1]
    eur <- key_facts[2]
```

```
  site %>%
    html_nodes("h1#sliderTopTitle") %>%
    html_nodes("span") %>%
    .[[2]] %>%
    html_attr("title") -> bewohnerinnen
  c(qm, eur, bewohnerinnen)
}
```

Der Befehl `Sys.sleep(2)` sorgt dafür, dass die Funktion bei jeder Ausführung erst mal zwei Sekunden „schläft“. Das ist leider nötig um zu verhindern, dass unsere IP automatisch gesperrt wird. Dadurch verlängert sich die Ausführung natürlich enorm.

15.4.3 Schritt 3: Testen

Wir führen die Funktion probeweise für eine (andere) Anzeige aus:

```
"https://www.wg-gesucht.de/wg-zimmer-in-Frankfurt-am-Main-Westend-Sud.8401009.html" %>%
  get_anzeige_details
## [1] "20m²"              "800€"              "3er WG (1w,1m,0d)"
```

Klappt!

15.4.4 Schritt 4: Serialisiert ausführen

Jetzt noch der Trick, diese Funktion mit `map()` auf alle 60 URLs auszuführen. Damit das hier aber klappt, ohne dass wir gesperrt werden, beschränke ich vorab auf die ersten zehn Einträge mit `head()`:

```
anzeige_urls %>%
  head(10) %>%
  map(get_anzeige_details) -> results

results
## [[1]]
## [1] "52m²"              "1090€"             "2er WG (0w,1m,0d)"
##
## [[2]]
## [1] "38m²"              "850€"              "2er WG (0w,1m,0d)"
##
## [[3]]
## [1] "25m²"              "800€"              "3er WG (1w,1m,0d)"
##
## [[4]]
## [1] "13m²"              "648€"              "2er WG (1w,0m,0d)"
##
## [[5]]
```

```
## [1] "25m²"              "800€"              "2er WG (1w,0m,0d)"
##
## [[6]]
## [1] "22m²"              "550€"              "3er WG (2w,0m,0d)"
##
## [[7]]
## [1] "16m²"              "692€"              "4er WG (2w,1m,0d)"
##
## [[8]]
## [1] "20m²"              "900€"              "2er WG (1w,0m,0d)"
##
## [[9]]
## [1] "30m²"              "1000€"             "3er WG (1w,1m,0d)"
##
## [[10]]
## [1] "13m²"              "374€"              "3er WG (2w,0m,0d)"
```

Mit `map_chr()` können wir aus dem Ergebnis direkt einen tibble basteln:

```
tibble(link = head(anzeige_urls, 10),
       flaeche = map_chr(results, 1),
       preis = map_chr(results, 2),
       bewohnerinnen = map_chr(results, 3)) -> wgs

wgs
## # A tibble: 10 x 4
##    link                                          flaeche preis bewohnerinnen
##    <chr>                                         <chr>   <chr> <chr>
##  1 https://www.wg-gesucht.de/wg-zimmer-in-Frankfu~ 52m²    1090€ 2er WG (0w,1m,~
##  2 https://www.wg-gesucht.de/wg-zimmer-in-Frankfu~ 38m²    850€  2er WG (0w,1m,~
##  3 https://www.wg-gesucht.de/wg-zimmer-in-Frankfu~ 25m²    800€  3er WG (1w,1m,~
##  4 https://www.wg-gesucht.de/wg-zimmer-in-Frankfu~ 13m²    648€  2er WG (1w,0m,~
##  5 https://www.wg-gesucht.de/wg-zimmer-in-Frankfu~ 25m²    800€  2er WG (1w,0m,~
##  6 https://www.wg-gesucht.de/wg-zimmer-in-Frankfu~ 22m²    550€  3er WG (2w,0m,~
##  7 https://www.wg-gesucht.de/wg-zimmer-in-Frankfu~ 16m²    692€  4er WG (2w,1m,~
##  8 https://www.wg-gesucht.de/wg-zimmer-in-Frankfu~ 20m²    900€  2er WG (1w,0m,~
##  9 https://www.wg-gesucht.de/wg-zimmer-in-Frankfu~ 30m²    1000€ 3er WG (1w,1m,~
## 10 https://www.wg-gesucht.de/wg-zimmer-in-Frankfu~ 13m²    374€  3er WG (2w,0m,~
```

15.5 Aufbereiten

Um mit den Daten sinnvoll weiterzuarbeiten müssen sie in ein numerisches Format gebracht werden. Das funktioniert z. T. am besten mit regulären Ausdrücken (“regex”), denen wir uns in der nächsten Sitzung widmen werden.

```
wgs$bewohnerinnen
##  [1] "2er WG (0w,1m,0d)" "2er WG (0w,1m,0d)" "3er WG (1w,1m,0d)"
##  [4] "2er WG (1w,0m,0d)" "2er WG (1w,0m,0d)" "3er WG (2w,0m,0d)"
##  [7] "4er WG (2w,1m,0d)" "2er WG (1w,0m,0d)" "3er WG (1w,1m,0d)"
## [10] "3er WG (2w,0m,0d)"

wgs %>%
  mutate(flaeche = parse_number(flaeche),
         preis = parse_number(preis),
         bw_gesamt = parse_number(bewohnerinnen),
         bw_w = str_extract(bewohnerinnen, "[0-9]+w") %>% parse_number(),
         bw_m = str_extract(bewohnerinnen, "[0-9]+m") %>% parse_number(),
         bw_d = str_extract(bewohnerinnen, "[0-9]+d") %>% parse_number())
## # A tibble: 10 x 8
##    link                  flaeche preis bewohnerinnen bw_gesamt  bw_w  bw_m  bw_d
##    <chr>                   <dbl> <dbl> <chr>             <dbl> <dbl> <dbl> <dbl>
##  1 https://www.wg-gesuc~      52  1090 2er WG (0w,1~         2     0     1     0
##  2 https://www.wg-gesuc~      38   850 2er WG (0w,1~         2     0     1     0
##  3 https://www.wg-gesuc~      25   800 3er WG (1w,1~         3     1     1     0
##  4 https://www.wg-gesuc~      13   648 2er WG (1w,0~         2     1     0     0
##  5 https://www.wg-gesuc~      25   800 2er WG (1w,0~         2     1     0     0
##  6 https://www.wg-gesuc~      22   550 3er WG (2w,0~         3     2     0     0
##  7 https://www.wg-gesuc~      16   692 4er WG (2w,1~         4     2     1     0
##  8 https://www.wg-gesuc~      20   900 2er WG (1w,0~         2     1     0     0
##  9 https://www.wg-gesuc~      30  1000 3er WG (1w,1~         3     1     1     0
## 10 https://www.wg-gesuc~      13   374 3er WG (2w,0~         3     2     0     0
```

16 Text: Breuer 2005

16.0.1 Lesetext

Breuer, Ingo. 2005. Statistiken oder: Wie werden „Nomaden“ in Marokko gemacht? In: Gertel, Jörg (Hrsg). *Methoden als Aspekte der Wissenskonstruktion. Fallstudien zur Nomadismusforschung.* Halle: Orientwissenschaftliches Zentrum der Martin-Luther-Universität Halle-Wittenberg. S. 55–73.

16.0.2 Fragen an den Text

1. Um welche Art von Text handelt es sich? Wer ist der Autor, und an wen wendet er sich?
2. Was ist das zentrale Anliegen des Texts? Wo steht/stehen die Fragestellung/en?
3. Was macht den Begriff des „Nomaden“ so schwierig? Was den Begriff des „Haushalts“?

4. Was ist an dem Argument spezifisch für den empirischen Kontext? Was lässt sich auf andere Kontexte übertragen?
5. Welche Schlüsse zieht der Autor aus der Untersuchung? Ist dies überzeugend?

17 String manipulation

17.1 Vorbereitung

Für diese Lektion brauchen wir folgende Pakete:

```
library(tidyverse)
library(rvest)
library(sf)
library(rnaturalearth)
```

17.2 Aufgabe

Ziel soll sein, aus Wikipedia eine Liste der Vulkane in Japan auszulesen und diese in einer Karte zu visualisieren, die mehr Informationen enthält als die auf Wikipedia angebotenen Optionen.

17.3 Tabellen aus Wikipedia laden

Mit `rvest` lässt sich eine Liste der Tabellen auslesen:

```
"https://en.wikipedia.org/wiki/List_of_volcanoes_in_Japan" %>%
  read_html %>%
  html_table -> alle_tabellen
```

17.4 Tabellen kombinieren

Die Tabellen 1, und 8 sind dabei ergänzende Elemente auf der Wikipedia-Seite und hier uninteressant:

```
alle_tabellen[c(1, 8)]
## [[1]]
## # A tibble: 2 x 1
##   X1
##   <chr>
## 1 Map all coordinates using: OpenStreetMap
## 2 Download coordinates as: KML
##
## [[2]]
## # A tibble: 4 x 2
##   `.mw-parser-output .navbar{display:inl~ `.mw-parser-output .navbar{display:in~
##   <chr>                                 <chr>
```

```
## 1 "Sovereign states"                          "Afghanistan\nArmenia\nAzerbaijan\nBa~
## 2 "States withlimited recognition"            "Abkhazia\nArtsakh\nNorthern Cyprus\n~
## 3 "Dependencies andother territories"         "British Indian Ocean Territory\nChri~
## 4 "Category\nAsia portal"                     "Category\nAsia portal"
```

Mit den restlichen Tabellen (2–7) wollen wir weiterarbeiten:

```
relevante_tabellen <- alle_tabellen[2:7]
```

Zunächst sollen sie "untereinader" (zeilenweise) in einen Datensatz kombiniert werden.

Das geht eigentlich mit dem Befehl `bind_rows()` ganz gut — allerdings müssen dafür die Spalten die selben Namen (hier gegeben) und die selben Typen haben (hier *nicht* gegeben).

```
relevante_tabellen %>%
  bind_rows()
## Error: Can't combine `..1$Elevation (m)` <character> and `..3$Elevation (m)` <integer>.
```

In der 3. und 4. Tabelle haben nämlich die Spalten `Elevation (m)` und `Elevation (ft)` den Typ `int`, während sonst alles den Typ `chr` hat:

```
walk(relevante_tabellen, str, vec.len = 0)
## tibble [33 x 5] (S3: tbl_df/tbl/data.frame)
##  $ Name          : chr [1:33]  ...
##  $ Elevation (m) : chr [1:33]  ...
##  $ Elevation (ft): chr [1:33]  ...
##  $ Coordinates   : chr [1:33]  ...
##  $ Last eruption : chr [1:33]  ...
## tibble [87 x 5] (S3: tbl_df/tbl/data.frame)
##  $ Name          : chr [1:87]  ...
##  $ Elevation (m) : chr [1:87]  ...
##  $ Elevation (ft): chr [1:87]  ...
##  $ Coordinates   : chr [1:87]  ...
##  $ Last eruption : chr [1:87]  ...
## tibble [14 x 5] (S3: tbl_df/tbl/data.frame)
##  $ Name          : chr [1:14]  ...
##  $ Elevation (m) : int [1:14] NULL ...
##  $ Elevation (ft): int [1:14] NULL ...
##  $ Coordinates   : chr [1:14]  ...
##  $ Last eruption : chr [1:14]  ...
## tibble [15 x 5] (S3: tbl_df/tbl/data.frame)
##  $ Name          : chr [1:15]  ...
##  $ Elevation (m) : int [1:15] NULL ...
##  $ Elevation (ft): int [1:15] NULL ...
##  $ Coordinates   : chr [1:15]  ...
##  $ Last eruption : chr [1:15]  ...
```

```
## tibble [24 x 5] (S3: tbl_df/tbl/data.frame)
##  $ Name          : chr [1:24]  ...
##  $ Elevation (m) : chr [1:24]  ...
##  $ Elevation (ft): chr [1:24]  ...
##  $ Coordinates   : chr [1:24]  ...
##  $ Last eruption : chr [1:24]  ...
## tibble [11 x 5] (S3: tbl_df/tbl/data.frame)
##  $ Name          : chr [1:11]  ...
##  $ Elevation (m) : chr [1:11]  ...
##  $ Elevation (ft): chr [1:11]  ...
##  $ Coordinates   : chr [1:11]  ...
##  $ Last eruption : chr [1:11]  ...
```

Hier ist es zunächst das einfachste, die Spalten in character strings umzuwandeln, um sie kombinieren zu können.

Für eine Spalte hieße das:

```
as.character(relevante_tabellen[[3]]$`Elevation (m)`)
##  [1] "423"  "11"   "854"  "758"  "574"  "-110" "851"  "813"  "-50"  "432"
## [11] "99"   "136"  "508"  "394"
```

Für eine Tabelle:

```
relevante_tabellen[[3]] %>%
  mutate(across(c(2, 3), as.character))
## # A tibble: 14 x 5
##    Name     `Elevation (m)` `Elevation (ft)` Coordinates        `Last eruption`
##    <chr>    <chr>           <chr>            <chr>              <chr>
##  1 Aogashi~ 423             1388             32°27 N 139°46 E ~ AD 1785[† 1]
##  2 Bayonna~ 11              36               31°53 17 N 139°55~ AD 1970[† 14]
##  3 Hachijō~ 854             2802             33°08 N 139°46 E ~ AD 1605[† 1]
##  4 Izu-Ōsh~ 758             2507             34°43 34 N 139°23~ Mt. Mihara: AD ~
##  5 Kōzushi~ 574             1877             34°13 N 139°09 E ~ AD 838[† 1]
##  6 Kurose   -110            -361             33°24 N 139°41 E ~ Caldera: older ~
##  7 Mikuraj~ 851             2792             33°52 16 N 139°36~ 6.3 ka BP[† 1]
##  8 Miyakej~ 813             2674             34°05 10 N 139°31~ AD 2013[† 16]
##  9 Myōjins~ -50             -164             31°55 05 N 140°01~ AD 1970[† 1]
## 10 Niijima  432             1417             34°22 N 139°16 E ~ Mt. Mukaiyama: ~
## 11 Sofugan~ 99              325              29°47 35 N 140°20~ (Discolored wat~
## 12 Sumisuj~ 136             446              31°26 20 N 140°03~ AD 1916[† 1]
## 13 Toshima  508             1667             34°31 N 139°17 E ~ 9.1–4.0 ka BP[†~
## 14 Torishi~ 394             1293             30°29 02 N 140°18~ AD 2002[† 1]
```

Über mehrere Tabellen, mit gleichzeitigem Kombinieren:

```
relevante_tabellen %>%
  map(mutate, across(c(2, 3), as.character)) %>%
```

```
  bind_rows() -> komplett

komplett
## # A tibble: 184 x 5
##    Name     `Elevation (m)` `Elevation (ft)` Coordinates          `Last eruption`
##    <chr>    <chr>           <chr>            <chr>                <chr>
##  1 Akaigaw~ 725             2379             .mw-parser-output ~ 1.3 Ma BP[† 1]
##  2 Mount A~ 512             1680             43°36 36 N 144°26 ~ 1000-200 BP[† ~
##  3 Daisets~ 2290            7513             43°39 47 N 142°51 ~ AD 1739[† 1]
##  4 Mount E~ 1320            4331             42°47 35 N 141°17 ~ 17th century[†~
##  5 Mount E~ 613             2028             41°48 14 N 141°09 ~ AD 1874[† 1]
##  6 Akan Ca~ -               -                43°27 04 N 144°06 ~ 0.25 Ma BP[† 1]
##  7 Mount M~ 1499            4916             43°23 10 N 144°00 ~ AD 2008[† 1]
##  8 Mount O~ 1370            4495             43°27 11 N 144°09 ~ 5-2.5 ka BP[† ~
##  9 Mount Iō 1563            5128             44°07 52 N 145°09 ~ AD 1936[† 1]
## 10 Kusshar~ -               -                43°37 16 N 144°20 ~ 2.3 ka BP[† 1]
## # ... with 174 more rows
```

17.5 Tabellen säubern

17.5.1 Parse number

Für einige Spalten (Höhe, Koordinaten, letzte Aktivtät) ist ein numerisches Format aber eigentlich tatsächlich wünschenswert.

Eine robuste Möglichkeit dafür ist `parse_number()`:

```
parse_number("Temperatur: -8° C")
## [1] -8
```

Mit dem Befehl `mutate()` lassen sich neue Spalten erstellen, oder vorhandene Spalten überschreiben. Mit `select()` können Spalten selektiert oder (mit einem `-`) ‚gelöscht' werden.

Der folgende Befehl wandelt die Spalte `Elevation (m)` in Zahlen um, gibt dieser neuen Spalte den Namen `elevation_m` und selektiert dann alles außer den alten `Elevation`-Spalten:

```
komplett %>%
  mutate(elevation_m = parse_number(`Elevation (m)`)) %>%
  select(-`Elevation (m)`, -`Elevation (ft)`)  -> vulkane_elev
```

Wobei das Attribut "problems" darauf hinweist, dass in manchen Strings keine Zahl gefunden wurde. Funktioniert hat es trotzdem (mit `NA` für fehlende Werte).

```
vulkane_elev
## # A tibble: 184 x 4
##    Name          Coordinates                    `Last eruption` elevation_m
##    <chr>         <chr>                          <chr>                 <dbl>
```

```
##  1 Akaigawa Cal~ .mw-parser-output .geo-default,.mw~ 1.3 Ma BP[† 1]          725
##  2 Mount Atosan~ 43°36 36 N 144°26 17 E / 43.610°N ~ 1000-200 BP[† ~         512
##  3 Daisetsuzan ~ 43°39 47 N 142°51 14 E / 43.663°N ~ AD 1739[† 1]           2290
##  4 Mount Eniwa   42°47 35 N 141°17 06 E / 42.793°N ~ 17th century[†~        1320
##  5 Mount Esan    41°48 14 N 141°09 58 E / 41.804°N ~ AD 1874[† 1]            613
##  6 Akan Caldera~ 43°27 04 N 144°06 36 E / 43.451°N ~ 0.25 Ma BP[† 1]          NA
##  7 Mount Meakan  43°23 10 N 144°00 29 E / 43.386°N ~ AD 2008[† 1]           1499
##  8 Mount Oakan   43°27 11 N 144°09 47 E / 43.453°N ~ 5-2.5 ka BP[† ~        1370
##  9 Mount Iō      44°07 52 N 145°09 54 E / 44.131°N ~ AD 1936[† 1]           1563
## 10 Kussharo Cal~ 43°37 16 N 144°20 10 E / 43.621°N ~ 2.3 ka BP[† 1]           NA
## # ... with 174 more rows
```

17.5.2 Reguläre Ausdrücke

Spalten des Typs `chr` können außerdem mit Befehlen aus dem `stringr`-Paket (Teil von `tidyverse`) bearbeitet werden:

- `str_remove_all()` entfernt Teile, die einem Muster entsprechen
- `str_extract_all()` behält nur die Teile, die dem Muster entsprechen
- `str_detect()` gibt `TRUE` aus, wenn das Muster gefunden wird, sonst `FALSE`
- `str_replace_all()` ersetzt Teile, die dem Muster entsprechen, durch etwas anderes
- etc.

Die Muster müssen dabei im RegEx-Format (regular expressions, reguläre Ausdrücke) angegeben werden. Reguläre Ausdrücke sind nicht R-spezifisch, sondern kommen in allen geläufigen Programmiersprachen zum Einsatz.

Es kann eine große Herausforderung sein, ein RegEx-Muster zu basteln, das genau das macht, was man will. Dabei können browserbasierte Testumgebungen wie https://www.regexpal.com/ behilflich sein. Für einen systematischeren Zugang empfiehlt es sich, ein Tutorial wie https://regexone.com/ durchzuarbeiten.

Im folgenden Befehl werden reguläre Ausdrücke direkt in Kombination mit `mutate()` benutzt, um metrische Koordinaten zu extrahieren:

```
vulkane_elev %>%
  mutate(Coordinates = str_extract(Coordinates, "[0-9.]+; [0-9.]+"),
         latitude = str_remove(Coordinates, "; [0-9.]+") %>% as.numeric,
         longitude = str_remove(Coordinates, "[0-9.]+; ") %>% as.numeric) %>%
  select(-Coordinates) -> vulkane_geo

vulkane_geo
## # A tibble: 184 x 5
##    Name                      `Last eruption`   elevation_m latitude longitude
##    <chr>                     <chr>                    <dbl>    <dbl>     <dbl>
##  1 Akaigawa Caldera          1.3 Ma BP[† 1]             725     43.1      141.
##  2 Mount Atosanupuri         1000-200 BP[† 1]           512     43.6      144.
```

```
##  3 Daisetsuzan Volcanic Group AD 1739[† 1]              2290     43.7      143.
##  4 Mount Eniwa                17th century[† 1]         1320     42.8      141.
##  5 Mount Esan                 AD 1874[† 1]               613     41.8      141.
##  6 Akan Caldera [ja]          0.25 Ma BP[† 1]             NA     43.5      144.
##  7 Mount Meakan               AD 2008[† 1]              1499     43.4      144.
##  8 Mount Oakan                5-2.5 ka BP[† 1]          1370     43.5      144.
##  9 Mount Iō                   AD 1936[† 1]              1563     44.1      145.
## 10 Kussharo Caldera           2.3 ka BP[† 1]              NA     43.6      144.
## # ... with 174 more rows
```

Es gibt mehrere Formate, in denen die letzte Aktivität für die meisten Vulkane angegeben ist:

- AD … (Jahrszahl)
- … ka BP (vor soundsoviel tausend Jahren)
- … Ma BP (vor soundsoviel Millionen Jahren)

Einige Werte fallen dabei aus dem Rahmen, damit können wir aber leben.Außerdem können auch Zeiträume angegeben sein, oder verschiedene Werte von verschiedenen Bergen. Hier soll es aber darum gehen, nach Möglichkeit *einen* der angegebenen Werte auszulesen.

Für jedes mögliche Format wird dabei zunächst eine eigene numerische Spalte angelegt und umgerechnet in "Jahre seit dem letzten Ausbruch". Mit `str_match` können dafür Teile eines gefundenen Patterns extrahiert werden.

Schließlich wird mit `pmin` das Minimum der so gefundenen Werte ermittelt.

```
vulkane_geo %>%
  mutate(ad_year = 2019 - str_match(`Last eruption`,
                                    "AD ([0-9]+)")[,2] %>%
                          parse_number,
         years_bp = str_match(tolower(`Last eruption`),
                              "([.0-9]+) bp")[,2] %>%
                           parse_number,
         ka_bp = str_match(tolower(`Last eruption`),
                           "([.0-9]+) ka bp")[,2] %>%
                     as.numeric * 1000,
         ma_bp = str_match(tolower(`Last eruption`),
                           "([.0-9]+) ma bp")[,2] %>%
                     as.numeric * 1000 * 1000,
         years_since_last_eruption = pmin(ad_year,
                                          years_bp,
                                          ka_bp,
                                          ma_bp,
                                          na.rm = TRUE)) %>%
  select(Name,
         elevation_m,
```

```
         latitude,
         longitude,
         years_since_last_eruption) -> vulkane_clean
```

17.6 Visualisierung

Zunächst werden die Vulkane in eine Simple Feature Collection umgewandelt und das CRS gesetzt:

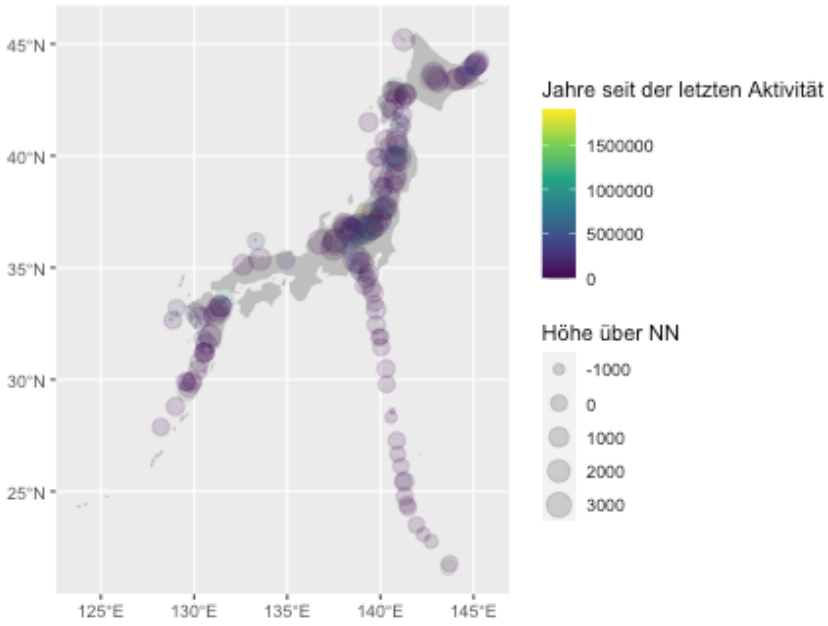
```
vulkane_clean %>%
  st_as_sf(coords = c("longitude", "latitude")) %>%
  st_set_crs(4326) -> vulkane_sf
```

Das `rnaturalearth` Paket lässt uns einfach die Polygone für die japanischen Inseln laden:

```
japan <- ne_countries(scale = "medium",
                      country = "Japan",
                      returnclass = "sf")
```

Dann lässt sich eine schnelle Karte zeichnen durch:

```
ggplot() +
  geom_sf(data = japan, fill = "gray", color = NA) +
  geom_sf(data = vulkane_sf,
          aes(color = years_since_last_eruption,
              size  = elevation_m),
          alpha = 0.2) +
  scale_colour_viridis_c("Jahre seit der letzten Aktivität") +
  scale_size_continuous("Höhe über NN")
```



18 Join und group

18.1 Vorbereitung

Für diese Lektion brauchen wir folgende Pakete:

```
library(tidyverse)
library(sf)
library(tmap)
```

18.2 Aufgabe

Ziel soll sein, eine Deutschlandkarte mit Tankstellenpreisen für Diesel zu erstellen.

18.3 Daten einlesen

Das "Tankerkönig"-Projekt veröffentlicht aktuelle Tankstellenpreise über eine API, und stellt historische Preise hier bereit: https://dev.azure.com/tankerkoenig/_git/tankerkoenig-data

Wir laden die Dateien für `prices` und `stations` von einem Tag (hier: 26.5.2019) herunter und speichern sie im Unterordner `resources` des Projektordners.

Dann können wir sie einlesen:

```
preise <- read_csv("resources/2019-05-26-prices.csv")
preise
```

```
## # A tibble: 231,174 x 8
##    date                station_uuid     diesel    e5   e10 dieselchange e5change
##    <dttm>              <chr>             <dbl> <dbl> <dbl>        <dbl>    <dbl>
##  1 2019-05-25 22:01:06 51e171d0-1a9c-4~   1.37  1.58  1.56            1        1
##  2 2019-05-25 22:02:06 8a796af1-8d78-4~   1.32  1.56  1.54            1        1
##  3 2019-05-25 22:02:06 2d658127-11b5-4~   1.28  1.52  0               0        1
##  4 2019-05-25 22:03:06 904d3a45-df30-4~   1.32  1.54  1.52            1        1
##  5 2019-05-25 22:03:06 a98ed5d0-261b-4~   1.34  1.57  1.55            1        1
##  6 2019-05-25 22:04:06 7671d5ad-4c7d-4~   1.30  1.54  0               1        1
##  7 2019-05-25 22:04:06 44fa4d12-5571-4~   1.34  1.58  1.56            1        1
##  8 2019-05-25 22:04:06 da9abcda-3218-4~   1.38  1.52  1.50            0        1
##  9 2019-05-25 22:04:06 bcba0c2b-fbe7-4~   1.35  1.55  1.53            0        1
## 10 2019-05-25 22:04:06 00061000-0001-4~   1.20  1.48  1.46            1        1
## # ... with 231,164 more rows, and 1 more variable: e10change <dbl>
```

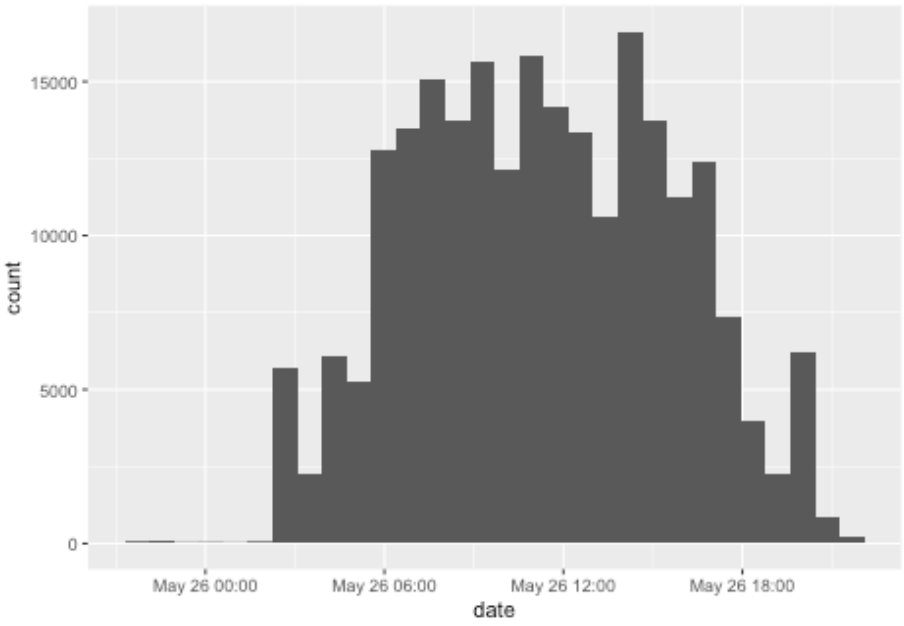
18.4 Überblick verschaffen

Beim nähreren Betrachten fällt auf, dass im Datensatz `preise` 231.174 Zeilen enthalten sind, in `stations` nur 15.668. Das liegt daran, das für jede Station *mehrere* Preisupdates im Datensatz `preise` stehen, jedoch nur *einmal* die gleichbleibenden Informationen (Name, Marke, Adresse, Koordinaten) in `stations`.

Beide Datensätze sind über einen eindeutigen „Key“ verbunden: In `preise` heißt er `station_uuid`, in `stations` einfach nur `uuid`.

Um ein besseres Gefühl für den Datensatz zu bekommen, könnten wir uns z. B. anschauen, zu welcher Uhrzeit wie viele Preise aktualisiert wurden:

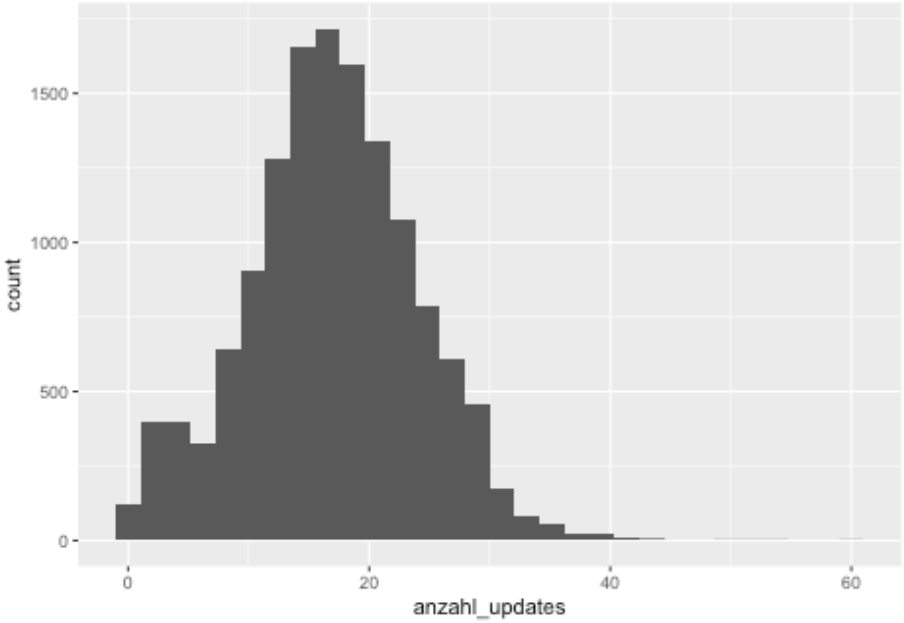
```
ggplot(preise) +
  geom_histogram(aes(x = date))
```



18.5 Zusammenfassen

Eine weitere Frage könnte sein: Wie sieht die Verteilung der Anzahl der Preisupdates je Tankstelle aus? Hierfür müssen wir den Datensatz `preise` anhand der Spalte `station_uuid` zusammenfassen und die Einträge zählen. Das geht mit `group_by()` und `summarize()`:

```
preise %>%
  group_by(station_uuid) %>%
  summarize(anzahl_updates = n()) %>%
  ggplot() +
    geom_histogram(aes(x = anzahl_updates))
```



Um unserem Ziel der Dieselkarte etwas näher zu kommen, sollten wir aber nicht die Anzahl der Updates zusammenfassen, sondern den Dieselpreis. Aber nach welchem Schema? Einfach nur den Durchschnitt (mit `mean()`) zu nehmen, könnte das Bild verfälschen: Man stelle sich z. B. vor, ein besonders teurer (oder günstiger) Preis sei nur wenige Sekunden gültig gewesen.

Wir orientieren uns einfach an der Börse und nehmen einfach den letzten gültigen Preis (wie der Aktienwert bei Börsenschluss). Dafür müssen wir den Datensatz erst mit `arrange()` chronologisch sortieren, dann entsprechend gruppieren und mit `last()` zusammenfassen:

```
preise %>%
  arrange(date) %>%
  group_by(station_uuid) %>%
  summarize(dieselpreis = last(diesel),
            e5preis     = last(e5),
            e10preis    = last(e10)) ->
  preise_nach_tankstelle

preise_nach_tankstelle
## # A tibble: 13,701 x 4
##    station_uuid                         dieselpreis e5preis e10preis
##    <chr>                                      <dbl>   <dbl>    <dbl>
##  1 00006210-0037-4444-8888-acdc00006210        1.33    1.55     1.53
##  2 00016899-3247-4444-8888-acdc00000007        1.31    1.53     1.51
##  3 00060001-d387-4444-8888-acdc00000001        1.37    1.62     1.60
```

```
##  4 00060009-3adf-4444-8888-acdc00000001        1.35    1.64     1.62
##  5 00060014-b0d9-4444-8888-acdc00000002        1.32    1.61     1.59
##  6 00060015-0090-4444-8888-acdc00000090        1.27    1.52     1.50
##  7 00060016-ed96-4444-8888-acdc00000001        1.35    1.57     1.55
##  8 00060034-0011-4444-8888-acdc00000011        1.37    1.61     1.59
##  9 00060051-533e-75a1-87f9-8a9f00060051        1.25    1.50     1.48
## 10 00060055-0001-4444-8888-acdc00000001        1.26    1.53     1.51
## # ... with 13,691 more rows
```

18.6 Verschneiden

Jetzt haben wir für jede Station nur noch eine Zeile mit den Preisen. Um das zu kartieren, fehlen noch die Informationen zu den Tankstellen. Dafür laden wir auch den `stations`-Datensatz für den richtigen Tag herunter und importieren ihn in R:

```
tankstellen <- read_csv("resources/2019-05-26-stations.csv")
tankstellen
```

```
## # A tibble: 15,668 x 11
##    uuid    name    brand  street house_number post_code city  latitude longitude
##    <chr>   <chr>   <chr>  <chr>  <chr>        <chr>     <chr>    <dbl>     <dbl>
##  1 0e18d0~ OIL! T~ OIL!   Evers~ <NA>         80999     Münc~     48.2     11.5
##  2 ad8122~ bft Bo~ bft    Godes~ 55           53175     Bonn      50.7      7.14
##  3 44e2bd~ bft Ta~ <NA>   Schel~ 53           36304     Alsf~     50.8      9.28
##  4 1a8e4d~ Hessol  Hessol Frank~ 65           61279     Gräv~     50.4      8.46
##  5 005056~ star T~ STAR   Leipz~ 11           06217     Mers~     51.4     12.0
##  6 d435f7~ ROSDOR~ Shell  A7 GÖ~ <NA>         37124     Rosd~     51.5      9.88
##  7 88a23d~ AVIA T~ AVIA   Burgs~ 8            63637     Joss~     50.2      9.48
##  8 f0e93f~ Aral T~ ARAL   Eicke~ 357          41063     Mönc~     51.2      6.45
##  9 005056~ star T~ STAR   Celle~ 55           29303     Berg~     52.8      9.97
## 10 8e47dd~ Aral T~ ARAL   Crail~ 32           74532     Ilsh~     49.2      9.93
## # ... with 15,658 more rows, and 2 more variables: first_active <dttm>,
## #   openingtimes_json <chr>
```

Wir verschneiden mit `inner_join()` unter Angabe der relevanten Spaltennamen und wählen die Spalten aus, mit denen wir weiterarbeiten wollen:

```
inner_join(preise_nach_tankstelle, tankstellen,
           by = c("station_uuid" = "uuid")) %>%
  select(dieselpreis, e5preis, e10preis, name, brand, latitude, longitude) ->
  preise_geo

preise_geo
```

```
## # A tibble: 13,700 x 7
##    dieselpreis e5preis e10preis name              brand       latitude longitude
##          <dbl>   <dbl>    <dbl> <chr>             <chr>          <dbl>     <dbl>
```

```
##  1        1.33    1.55     1.53 Beducker - Quali~ Beducker       48.6     10.9
##  2        1.31    1.53     1.51 Röttenbach        BFT Pickel~    49.7     10.9
##  3        1.37    1.62     1.60 Haisch Mineralöl~ TankCenter~    48.0      7.59
##  4        1.35    1.64     1.62 Tank-Kontor Wilh~ <NA>           47.9      9.42
##  5        1.32    1.61     1.59 Tank-Kontor Baie~ <NA>           47.8      9.65
##  6        1.27    1.52     1.50 Schindele, Lochb~ <NA>           47.7      9.53
##  7        1.35    1.57     1.55 bft-Tankstelle H~ BFT            48.1      7.78
##  8        1.37    1.61     1.59 EXTROL Tank- & W~ EXTROL         48.0      7.79
##  9        1.25    1.50     1.48 Wingenfeld Energ~ Wingenfeld~    50.8     10.2
## 10        1.26    1.53     1.51 Wilhlem Heim GmbH Oel - Heim     48.6      9.03
## # ... with 13,690 more rows
```

`inner_join` hat die Besonderheit, dass nur Zeilen im kombinierten Datensatz übrigbleiben, deren Key in *beiden* Datensätzen gefunden wurde. Mit `left_join` würden hier alle Preise behalten werden (und die fehlenden Koordinaten mit `NA` ergänzt), mit `right_join` würden alle Stationen behalten werden (und fehlende Preise mit `NA` ergänzt). `full_join` löscht gar keine Informationen.

18.7 Kartieren

Den georeferenzierten Datensatz der Preise wandeln wir in eine Simple Feature Collection um:

```
preise_geo %>%
  st_as_sf(coords = c("longitude", "latitude")) -> preise_sf
```

`ggplot()` kartiert so einen großen Datensatz nur langsam. Wir nehmen stattdessen das Paket `tmap()` zur Hand, das mit einer ähnlichen Grammatik funktioniert.

Interaktive Karten lassen sich mit `tmap` produzieren, wenn die Option

```
tmap_mode("view")
```

gesetzt ist. Aus technischen Gründen wird an dieser Stelle im Skript darauf verzichtet und wir bleiben beim `plot`-Modus:

```
tm_shape(preise_sf) +
  tm_dots()
```

Zwei Koordinaten sind quatsch! Wir finden ihre ungefähren Werte mit `summary`:

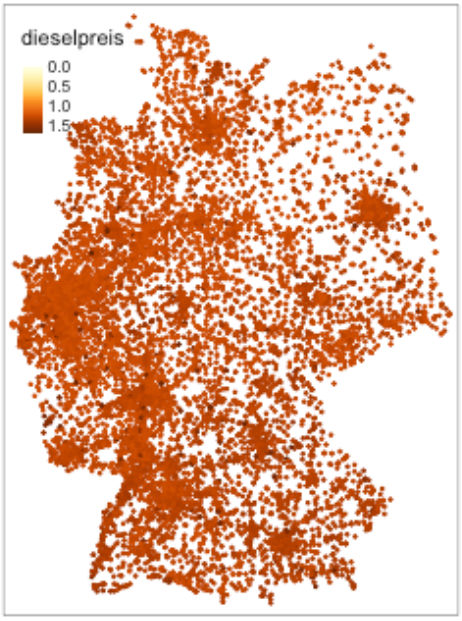
```
summary(preise_geo$longitude)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   5.901   8.021   9.275   9.607  11.058  97.364
```

Und filtern sie raus, und wiederholen die Umwandlung (diesmal auch mit CRS)

```
preise_geo %>%
  filter(longitude < 80) %>%
  st_as_sf(coords = c("longitude", "latitude")) %>%
  st_set_crs(4326) -> preise_sf
```

Dann mappen wir nochmal:

```
tm_shape(preise_sf) +
  tm_dots("dieselpreis", style = "cont")
```



Schon ganz hübsch, aber die Skala wird nun verzerrt durch sehr teure Autobahntankstellen einerseits, und falsche Null-werte andererseits:

```
summary(preise_sf$dieselpreis)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   1.249   1.289   1.290   1.329   1.679
```

18.8 Choroplethen

Eine Lösung wäre, die Daten auf Kreisebene zusammenzufassen, und zwar anhand ihres Medians. Damit würden diese Ausreißer keine Role mehr spielen.

Das `eurostat`-Paket macht es einfach, diese Geodaten einzulesen. NUTS3 ist die Ebene der Stadt- und Landkreise bzw. ihrer europäischen Equivalente.

```
kreise <- eurostat::get_eurostat_geospatial(nuts_level = 3) %>%
  filter(CNTR_CODE == "DE")
```

Mal schauen wie es aussieht:

```
tm_shape(kreise) +
  tm_polygons()
```

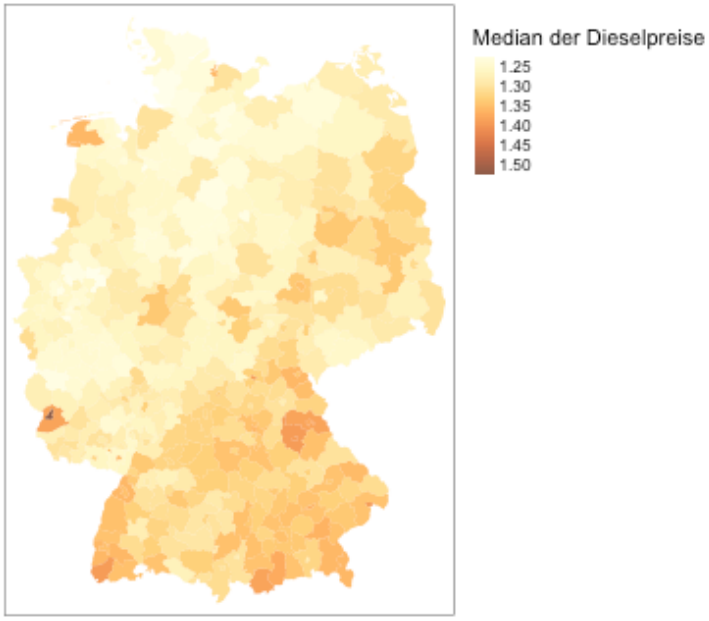
18.9 Räumliches Verschneiden

mit `st_join` werden Datensätze nicht mit einem Key verschnitten, sondern anhand ihrer Geolokation. Dann können wir wieder ganz normal `group_by` und `summarise` verwenden:

```
st_join(kreise, preise_sf) %>%
  group_by(NUTS_ID) %>%
  summarise(dieselpreis = median(dieselpreis),
            e5preis     = median(e5preis),
            e10preis    = median(e10preis)) -> preise_kreise
```

Und so könnte vielleicht ein vorläufiges Ergebnis aussehen:

```
tm_shape(preise_kreise) +
  tm_fill("dieselpreis",
          title = "Median der Dieselpreise",
          style = "cont",
          alpha = 0.8) +
  tm_layout(legend.outside = TRUE)
```



19 Rmarkdown und Kollaboration

19.1 Kollaboration

Im Folgenden sind einige Überlegungen und Tools aufgeführt, die bei der Kollaboration an einem Gruppenprojekt hilfreich sein könnten.

19.1.1 Find your workflow

Es gibt zwar effiziente und weniger effiziente Praktiken, sowie hilfreiche und weniger hilfreiche Tools – es gibt jedoch keinesfalls *den einen* perfekten Workflow. Letztendlich kommt es darauf an, sich gemeinsam für einen Workflow zu entscheiden und diesen klar festzulegen, oder sogar irgendwie festzuhalten.

Dazu gehört es z. B. auch, sich auf Formate von Dateinamen oder R-Objekten zu einigen, und einen ‚offiziellen' Kanal für die Gruppenkommunikation festzulegen – sei es E-Mail, Slack oder WhatsApp.

19.1.2 Zwischenstände speichern

Wenn z. B. ein (längeres) Script einen (größeren) Datensatz generiert und ihn dem Objektnamen `mein_datensatz` zuweist, dann erscheint der Datensatz im lokalen Environment. Um den Datensatz mit anderen zu teilen, muss er irgendwie exportiert werden. Hierfür ist es ratsam, die `save()`-Funktion zu nutzen:

```
save(mein_datensatz, file = "zwischenstand_mein_datensatz.Rdata")
```

So wird eine Datei erstellt, die den Datensatz enthält und verschoben oder geteilt werden kann. Eine solche Datei kann auch andere Objekte (Funktionen, Listen, …) und mehrere Objekte auf einmal enthalten. Die Dateiendung ist dabei eigentlich egal, `.Rdata` scheint aber Usus zu sein.

Aus der Datei können Objekte dann jederzeit wieder ins Environment geladen werden mit dem Befehl:

```
load("zwischenstand_mein_datensatz.Rdata")
```

Auch wenn ein Skript auf einmal nicht mehr funktioniert (etwa weil die API sich ändert), ist es hilfreich, auf solche Zwischenstände zurückgreifen zu können.

19.1.3 RStudio Cloud

Im Seminar haben wir RStudio Cloud über den Browser genutzt. Eine gute Möglichkeit der Kollaboration ist, ein Cloud-Projekt miteinander zu teilen. Aber Achtung: Dabei wird beim Teilen immer eine Kopie erstellt – daher kann es schwierig werden, gleichzeitig vorgenommene Änderungen wieder zusammenzuführen.

Wer die Cloud im Rahmen des Projektseminars nutzt, sollte keine vertraulichen Daten verarbeiten und sich sorgfältig absprechen, wer gerade an welcher Version des Projekts wie weiterarbeitet.

19.1.4 File sharing

Mit File-Sharing-Plattformen wie Dropbox (oder Google Drive, OneDrive, …) können ganze Ordner leicht zwischen verschiedenen Rechnern synchronisiert werden. Hierbei ist es jedoch keine gute Idee, den Projektordner (das *working directory*) selbst zu synchronisieren – dabei sind Probleme vorprogrammiert! Stattdessen ist es vielleicht eine Idee, einen Unterordner anzulegen, der dann mit diesen Diensten verknüpft werden kann.

Aber Vorsicht: Wenn mit vertraulichen Daten gearbeitet wird, ist der Austausch der Daten über diese Programme nicht zulässig!

19.1.5 E-Mail

Der Austausch von Scripts, Datensätzen, zip-Dateien etc. über E-Mail ist vielleicht umständlich, aber im Zweifel eine praktikable Lösung. Nur bei sehr großen Datensätzen ist man vielleicht irgendwann auf File-Sharing-Dienste wie WeTransfer angewiesen – aber auch hier mit den gleichen datenschutzrechtlichen Bedenken.

19.1.6 Git

Git ist ein mächtiges Tool für Version Control, und war lange aufgrund seiner etwas steilen Lernkurve berüchtigt. Mit dem richtigen Interface (RStudio bietet z. B. eine GUI an) ist das aber alles gar nicht so schwierig, und Portale wie Github oder Bitbucket erleichtern die zentrale Verwaltung von Repositories.

Git hat eine steile Lernkurve und erleichtert die Arbeit erst, wenn man damit sicher umgehen kann. Aber Git ist zurecht ein absolutes Standardwerkzeug für Entwickler*innen, und wer jemals ernsthaft in einer Gruppe (oder auch alleine!) an Code arbeiten will, sollte sich mit Git auseinandersetzen.

Eine ausführliche Anleitung für die Verknüpfung von RStudio und GitHub findet sich hier: https://happygitwithr.com/

19.1.7 Trello

Trello ist ein flexibles Projektmanagement-Tool, in dem man gemeinsame Listen anlegen kann.

Neben einer gemeinsamen To-do-Liste bietet es sich gerade bei größeren Teams an, wer gerade an was arbeitet (Liste: „Till arbeitet an…“).

Es ist erfahrungsgemäß ratsam, nicht nur kryptische Stichpunkte festzuhalten, sondern die gewünschten Resultate als „Stories“ zu beschreiben:

- „Wir haben ein API-Passwort“
- „Datensatz x ist mit Datensatz y verschnitten“
- „Wir haben fünf Quellen für den Theorieteil“
- …

19.2 Rmarkdown

19.2.1 Text formatieren

Wir arbeiten schon von Anfang an mit im Rmarkdown-Format. Wie Überschriften, Links, Bilder usw. in Rmarkdown genau funktionieren, ist in dieser Übersicht und auf diesem Cheat Sheet (Punkt: Pandoc's Markdown) gut festgehalten.

19.2.2 Der Knit-Button

Wenn wir im YAML-Header die Zeile

```
output: html_document
```

setzen, erscheint (nach Abspeichern) ein „Knit“-Button in der GUI. Durch Drücken auf diesen Knopf passiert folgendes:

- R erstellt („strickt“) ein HTML-Dokument aus dem vorliegenden Markdown und den Code Chunks

- Dabei spielen „gespeicherte“ Objekte keine Rolle, die Chunks werden einfach der Reihe nach (in einem neuen Environment) ausgeführt
- Externe Datensätze o. ä. müssen also am Anfang des Dokuments explizit geladen werden (etwa mit `load()`). Für den Abschlussbericht ist es ratsam, einen vorbereiteten Datensatz am Anfang des Rmarkdown-Dokuments so zu laden.

Im YAML-Header können noch viele weitere Angaben gemacht werden, die das Resultat verändern. Hier eine gute Dokumentation: https://bookdown.org/yihui/rmarkdown/html-document.html

19.2.3 Kable

Im `knitr`-Paket sorgt der Befehl `kable()` für eine schöne Darstellung von Tabellen:

```
ggplot2::diamonds %>%
  head() %>%
  knitr::kable()
```

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
0.24	Very Good	J	VVS2	62.8	57	336	3.94	3.96	2.48

Auch hier gibt es wieder vielfältige Möglichkeiten zur visuellen Gestaltung. Diesen Post finde ich immer besonders hilfreich: https://haozhu233.github.io/kableExtra/awesome_table_in_html.html

19.2.4 Chunk Options

Am Anfang eines Code Chunks kann genau festgelegt werden, ob der Code ausgeführt werden soll, ob er im finalen Dokument erscheinen soll, ob Warnungen oder Fehler ausgegeben werden sollen, etc. Wenn zum Beispiel Libraries „versteckt“ geladen werden sollen, geht das mit diesem Code Chunk:

````
```{r, include = FALSE}
library(tidyverse)
library(rvest)
```
````

Ein überblick über die Chunk-Optionen findet sich hier: https://bookdown.org/yihui/rmarkdown/r-code.html

19.2.5 Wissenschaftliches Zitieren

Wer die Vorzüge von Literaturverwaltungssoftware (wie Citavi, Zotero, …) schon schätzen gelernt hat, kann in Rmarkdown folgendermaßen vorgehen:

19.2.5.1 Schritt 1: Exportieren Die relevante Literatur in eine BibTex-Datei im R-Arbeitsverzeichnis exportieren, z.B. `literatur.bib`. BibTex (bzw. BibLatex) ist ein bewährtes und gut dokumentiertes Format. Ein Eintrag sieht dann z.B. so aus, wobei `bortz` der „Name“ des Eintrags ist, den wir frei wählen können:

```
@book{bortz,
  author = {Bortz, J{\"u}rgen and Schuster, Christof},
  title = {{Statistik f{\"u}r Human- und Sozialwissenschaftler}},
  publisher = {Springer},
  year = {2010},
  address = {Berlin},
  edition = {7},
}
```

19.2.5.2 Schritt 2: Verlinken Im YAML-Header des Rmarkdown-dokuments die Angabe ergänzen:

```
bibliography: literatur.bib
```

Damit weiß der „Knit“-Befehl, wo er nach Literatur suchen soll.

19.2.5.3 Schritt 3: Zitieren Im Text kann dann z. B. so zitiert werden:

```
Das zentrale Grenzwerttheorem besagt, dass die Stichprobenverteilung von
$\bar{x}$ mit steigender Stichprobengröße $n$ in eine Normalverteilung übergeht
[@bortz: 86].
```

19.2.5.4 Schritt 4: Stricken Beim „Knit“-Befehl wird ein Literaturverzeichnis automatisch erstellt und ans Ende des Dokuments gehängt. Deshalb beendet man das Dokument am besten mit der Zeile:

```
## Literaturverzeichnis
```

20 Text: Bowker und Star 1999

20.0.1 Lesetext

Bowker, Geoffrey C. und Susan Leigh Star. 1999. Introduction: To classify is human. In: *Sorting things out: Classification and its consequences.* Cambridge: MIT Press. S. 1–32.

20.0.2 Fragen an den Text

1. Um welche Art von Text handelt es sich? Wer sind die Autor*innen, und an wen wenden sie sich?
2. Was ist das zentrale Anliegen des Texts? Wo genau steht das?
3. Was meinen die Autor*innen, wenn sie von Infrastruktur sprechen?
4. Was sind zwei Beispiele für Kategorisierung (im Sinne des Texts) aus Ihren Alltagserfahrungen?
5. Jeweils: Wer sortiert? Würden Sie sagen, dass diese Praxis politisch ist?
6. Auf Seite 11 behaupten die Autor*innen, dass kein Klassifizierungssystem drei einfachen Anforderungen genügen würde. Sehen Sie das auch so? Prüfen Sie Ihre Beispiele!
7. Der letzte Abschnitt zieht einen ziemlich kryptischen Vergleich mit einem Wald… was wollen die Autor*innen uns damit sagen?

21 Interaktive Visualisierungen

21.1 Einleitende Bemerkungen

Bei den drei vorgestellten Paketen handelt es sich um sehr flexible Frameworks, deren ausführliche Besprechung (so wie ggplot2) problemlos ein eigenes Seminar füllen könnten. Hier werden sie nur anhand von einfachsten Beispielen schnell vorgestellt. Bei der weiterführenden Anwendung im Rahmen eines Projekts ist es sinnvoll, sich noch einmal zielgerichtet zu beraten.

21.2 Interaktive Karten mit `tmap`

21.2.1 Vorbereitung

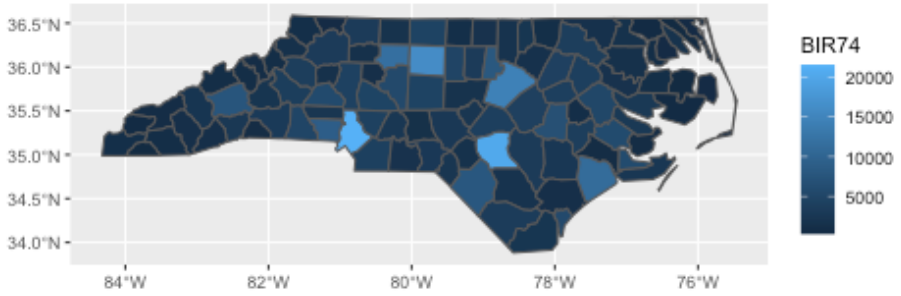
Wir laden `tmap` und importieren einen Beispieldatensatz (simple features) mit medizinische Statistiken aus North Carolina:

```
library(tmap)
library(sf)
demo(nc, ask = F, echo = F)
## Reading layer `nc.gpkg' from data source
##   `/usr/local/lib/R/4.1/site-library/sf/gpkg/nc.gpkg' using driver `GPKG'
## Simple feature collection with 100 features and 14 fields
## Attribute-geometry relationship: 0 constant, 8 aggregate, 6 identity
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
## Geodetic CRS:  NAD27
```

21.2.2 Bisher: Karten mit ggpot2

Wir haben schon gelernt, wie man mit dem Befehl `ggplot()` Karten erstellt. Zur Wiederholung zeichnen wir eine Choropletenkarte mit den Geburtsraten der Counties von 1974:

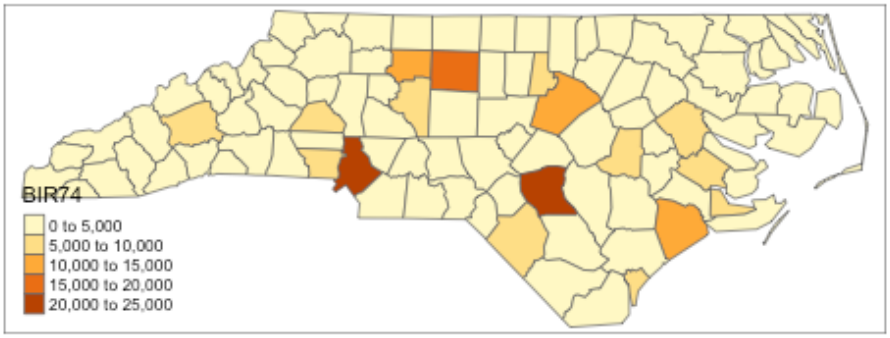
```
library(ggplot2)

ggplot(nc) +
  geom_sf(aes(fill = BIR74))
```



21.2.3 Neu: Karten mit tmap

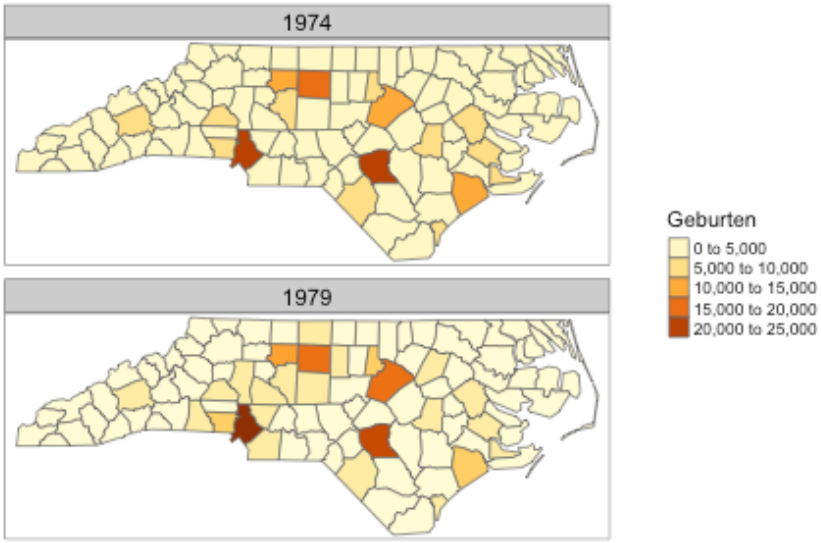
Der Befehl `qtm()` ("quick thematic map") ist ideal um Geodaten schnell zu visualisieren:

```
qtm(nc, fill = "BIR74")
```



Für ausführlichere Karten folgt `tmap` einer sehr ähnlichen Logik wie `ggplot2`, hat aber einige Unterschiede in der "Grammatik":

```
tm_shape(nc) +
  tm_polygons(col = c("BIR74", "BIR79"), title = "Geburten") +
  tm_facets() +
  tm_layout(panel.show = T,
            panel.labels = c("1974", "1979"),
            legend.outside = T, legend.position = c("center", "center"))
```



21.2.4 Interaktive Karten

Wenn wir den `tmap_mode` einmal wechseln,

```
tmap_mode("view")
```

zeichnen wir ab sofort interaktive Karten:

```
qtm(nc)
```

Hier lassen sich dann auch interaktive Elemente wie Popups, Layer und die Hintergrundkarte bearbeiten. (Hier nur als technische Demonstration, besonders hilfreich finde ich es nicht:)

```
tm_shape(nc, cache = F) +
  tm_basemap("OpenStreetMap") +
  tm_polygons(col = c("BIR74", "BIR79"), id = "NAME", popup.vars = c("AREA", "CNTY_ID")) +
  tm_facets(as.layers = T)
```

21.2.5 Weitere Informationen:

- tmap: get started!
- Making maps with R: tmap

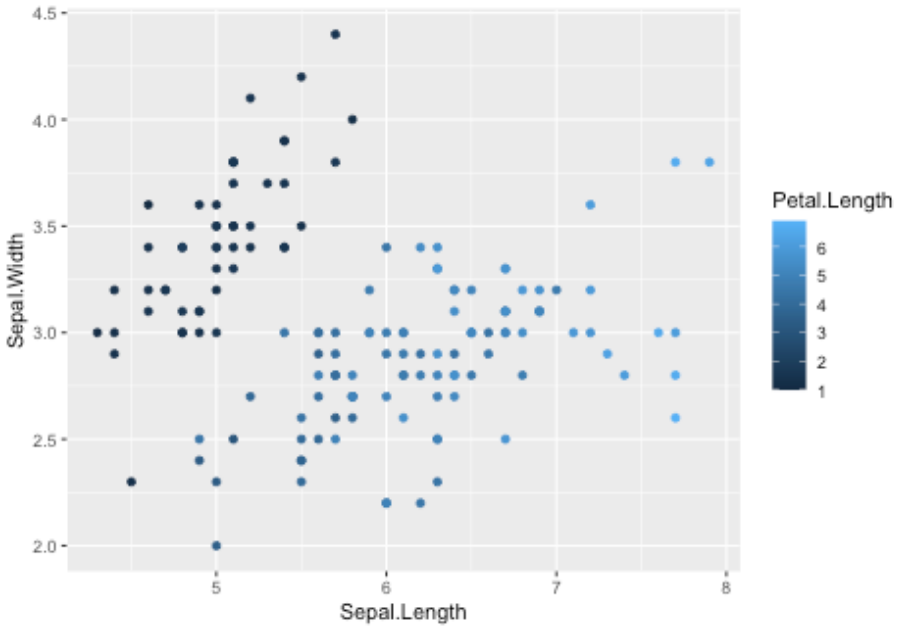
21.3 Interaktive Plots mit `plotly`

Plotly ist ein kommerzielles Produkt, das Framework lässt sich aber offen nutzen. (Verkauft werden sollen dann Dashboards, Beratungsleistungen und Infrastruktur für Firmenkunden.)

21.3.1 Bisher: Scatterplot mit `ggplot2`

Wir plotten den Beispieldatensatz `iris`:

```
ggplot(iris) +
  geom_point(aes(x = Sepal.Length, y = Sepal.Width, color = Petal.Length))
```



21.3.2 Neu: 3D mit `plotly`

```
library(plotly)

plot_ly(iris, x = ~ Sepal.Length, y = ~ Sepal.Width, z = ~ Petal.Width, color = ~ Petal.Leng
```

21.3.3 Weitere Informationen

- Plotly R Open Source Graphing Library
- Getting Started with Plotly in R

21.4 Web apps mit `shiny`

Shiny ist das umfangreichste und flexibelste der hier vorgestellten Pakete und für das Erstellen von Web-Apps zur interaktiven Visualisierung von Datensätzen gedacht.

Visualisierungen mit Shniny können jedoch nicht als Teil einer statischen HTML-Seite angezeigt werden, sondern müssen zur Ansicht auf einem entsprechend konfigurierten Server laufen.

Wir haben in der Sitzung eine einfache Shiny-App erstellt und lokal angesehen, zur Veröffentlichung müsste sie jedoch z. B. auf https://shinyapps.io hochgeladen werden.

21.4.1 Weitere Informationen

- Learn Shiny
- Mastering Shiny

22 Clusteranalyse

22.1 Voraussetzungen

```
library(tidyverse)
library(plotly)
library(ggdendro)
```

22.2 Datensatz und Überblick

Mitgeliefert in R ist ein beliebter Beispieldatensatz mit Daten aus einem Automagazin:

```
data(mtcars)
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

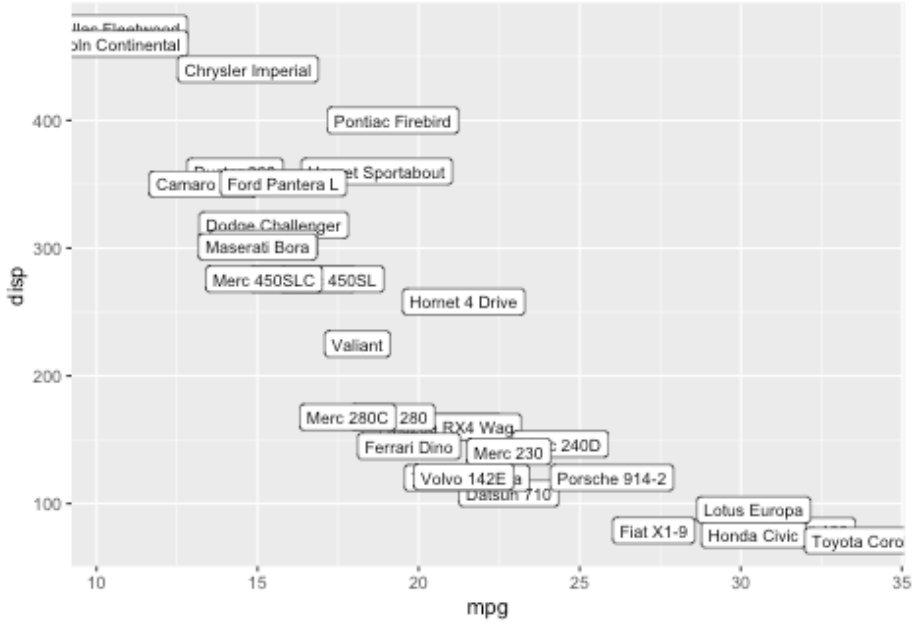
Im Folgenden geht es um die technischen Aspekte einer Clusteranalyse, und diese Daten dienen zur Veranschaulichung.

Einen ersten Überblick über den Datensatz kriegen wir mit:

```
summary(mtcars)
##       mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0  
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##       drat             wt             qsec             vs        
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
##        am              gear            carb      
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

Oder mit Scatterplots in verschiedenen denkbaren Variationen:

```
ggplot(mtcars) +
  geom_label(aes(x = mpg, y = disp, label = rownames(mtcars)), size = 3)
```



22.3 Clusteranalyse

Grundüberlegung der Clusteranalyse ist, in wie viele und welche Gruppen die Merkmalsträger (hier: die Autos) einer (multivariaten) Verteilung sinnvoll eingeteilt werden können. Dabei gibt es viele verschiedene mathematische Methoden – hier soll nur die Grundvariante (k-means-Clustering anhand euklidischer Distanz) besprochen werden.

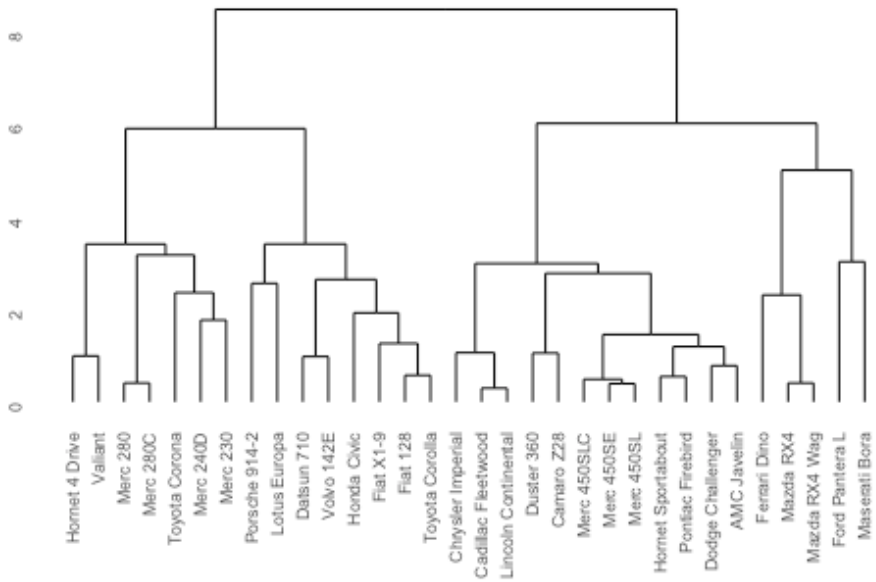
Zunächst führen wir mit `scale()` eine z-Transformation aller Variablen durch, damit werden sie alle gleich stark gewichtet:

```
mtcars %>%
  scale() %>%
  head()
##                          mpg        cyl        disp         hp       drat
## Mazda RX4          0.1508848 -0.1049878 -0.57061982 -0.5350928  0.5675137
## Mazda RX4 Wag      0.1508848 -0.1049878 -0.57061982 -0.5350928  0.5675137
## Datsun 710         0.4495434 -1.2248578 -0.99018209 -0.7830405  0.4739996
## Hornet 4 Drive     0.2172534 -0.1049878  0.22009369 -0.5350928 -0.9661175
## Hornet Sportabout -0.2307345  1.0148821  1.04308123  0.4129422 -0.8351978
## Valiant           -0.3302874 -0.1049878 -0.04616698 -0.6080186 -1.5646078
##                           wt       qsec         vs         am       gear
## Mazda RX4         -0.610399567 -0.7771651 -0.8680278  1.1899014  0.4235542
## Mazda RX4 Wag     -0.349785269 -0.4637808 -0.8680278  1.1899014  0.4235542
## Datsun 710        -0.917004624  0.4260068  1.1160357  1.1899014  0.4235542
## Hornet 4 Drive    -0.002299538  0.8904872  1.1160357 -0.8141431 -0.9318192
```

```
## Hornet Sportabout  0.227654255 -0.4637808 -0.8680278 -0.8141431 -0.9318192
## Valiant            0.248094592  1.3269868  1.1160357 -0.8141431 -0.9318192
##                          carb
## Mazda RX4           0.7352031
## Mazda RX4 Wag       0.7352031
## Datsun 710         -1.1221521
## Hornet 4 Drive     -1.1221521
## Hornet Sportabout -0.5030337
## Valiant            -1.1221521
```

Dann wird mit dem `dist()`-Befehl anhand der vier Variablen eine „Distanz“ zwischen den Staaten berechnet, mit `hclust()` mögliche Cluster berechnet und mit `ggdendrogram()` eine Visualisierung von mögliche Clusteranordnungen ausgegeben:

```
mtcars %>%
  scale() %>%
  dist() %>%
  hclust() %>%
  ggdendrogram()
```



Dabei befindet sich auf der y-Achse die Distanz, bei der verschiedene Cluster zusammenfallen. Ganz unten sind es 32 Cluster mit je einem Staat, dann werden nach und nach Cluster zusammengefasst, bis es ganz oben nur noch ein Cluster mit 32 Autos ist. Je länger die parallelen Vertikalen Striche, desto beständiger die Cluster.

Hier würden sich vier oder fünf Cluster anbieten, wir entscheiden uns für 5 Cluster.

Mit `kmeans()` bilden wir die Cluster (nach z-Transformation) und können direkt Größe, Charakteristika und "Mitglieder" der Cluster einsehen:

```
mtcars %>%
  scale() %>%
  kmeans(4)
## K-means clustering with 4 clusters of sizes 12, 10, 7, 3
##
## Cluster means:
##          mpg        cyl       disp         hp        drat          wt
## 1 -0.8363478  1.0148821  1.0238513  0.6924910 -0.88974768  0.90635862
## 2  1.0900003 -1.0008838 -0.9991381 -0.8632588  0.99206769 -1.05671433
## 3  0.1082193 -0.5849321 -0.4486701 -0.6496905 -0.04967936 -0.02346989
## 4 -0.5404546  0.6415922  0.2819522  1.6235100  0.36801694 -0.04829030
##         qsec         vs         am       gear       carb
## 1 -0.3952280 -0.8680278 -0.8141431 -0.9318192  0.1676779
## 2  0.1450802  0.5208167  1.1899014  0.6946289 -0.5030337
## 3  1.1854841  1.1160357 -0.8141431 -0.1573201 -0.4145882
## 4 -1.6688182 -0.8680278  1.1899014  1.7789276  1.9734398
##
## Clustering vector:
##           Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive
##                   2                   2                   2                   3
##   Hornet Sportabout             Valiant          Duster 360           Merc 240D
##                   1                   3                   1                   3
##            Merc 230            Merc 280           Merc 280C          Merc 450SE
##                   3                   3                   3                   1
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Lincoln Continental
##                   1                   1                   1                   1
##   Chrysler Imperial            Fiat 128         Honda Civic      Toyota Corolla
##                   1                   2                   2                   2
##       Toyota Corona    Dodge Challenger         AMC Javelin          Camaro Z28
##                   3                   1                   1                   1
##    Pontiac Firebird           Fiat X1-9       Porsche 914-2        Lotus Europa
##                   1                   2                   2                   2
##      Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E
##                   4                   4                   4                   2
##
## Within cluster sum of squares by cluster:
## [1] 23.083489 35.653610 21.287980  9.978908
##  (between_SS / total_SS =  73.6 %)
##
## Available components:
```

```
##
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
## [6] "betweenss"    "size"         "iter"         "ifault"
```

Damit R die Clusterzugehörigkeit (1 bis 5) im Folgenden nicht als metrische sondern als nominalskalierte Variable versteht, ziehen wir sie aus dem Ergebnis heraus und wandeln sie in einen Factor um (s. [Exkurs Factors]).

```
mtcars %>%
  scale %>%
  kmeans(5) %>%
  .$cluster %>%
  factor() -> memberships
```

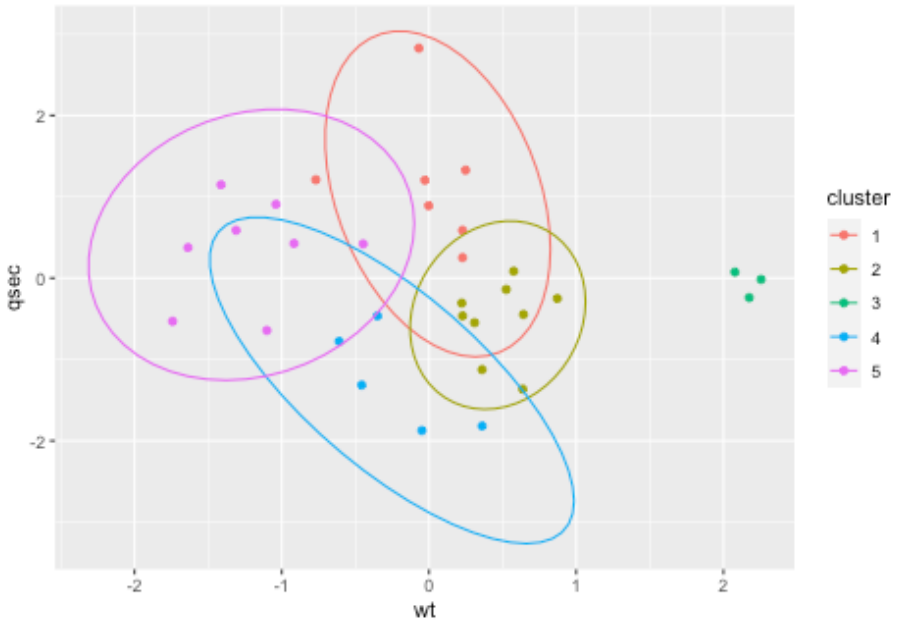
Diesen „faktorierten“ Cluster-Vektor fügen wir dem Datensatz hinzu:

```
mtcars_cluster <- mtcars %>%
  scale %>%
  as_tibble() %>%
  mutate(cluster = memberships)
```

22.4 Visualisierung

Die verschiedenen Cluster lassen sich dann z.B. farblich voneinander abgrenzen:

```
ggplot(mtcars_cluster, aes(x = wt, y = qsec, color = cluster)) +
  geom_point() +
  stat_ellipse()
```



Zu einer Clusteranalyse würde dann auch noch gehören, die einzelnen Cluster anhand ihrer Charakteristika zu beschreiben.

23 Statistische Analyseverfahren

Anhand von Testdatensätzen in R und den Unterlagen aus der Statistikvorlesung haben wir in dieser Sitzung wiederholend Techniken zur statistischen Analyse und deren Interpretation zusammengetragen:

- Korrelation und mit `cor()` und Korrelationstest `cor.test()`
- t-Test mit `t.test()` (1- und 2-Stichproben-Varianten)
- F-Test mit `var.test()`

24 Text: Beer 2016

24.0.1 Lesetext

Beer, David. 2016. Measurement. In: *Metric Power.* London: Palgrave Macmillan. S. 37–75.

Daraus: Seiten 37 – 60

24.0.2 Fragen an den Text:

24.0.3 Fragen an den Text

1. Wie ging es Ihnen mit dem Text? Was war vielleicht schwierig? Wie sind Sie damit umgegangen?
2. Um welche Art von Text handelt es sich? In welcher Form und wo ist er erschienen, wer ist der Autor, und an wen wendet er sich?
3. Was ist das zentrale Anliegen des Texts? Wo genau steht das?
4. Welche theoretischen Referenzen werden herangezogen? Was davon kommt Ihnen bekannt vor? Was können Sie wie einordnen?
5. Welche Entwicklung im späten 18. und frühen 19. Jahrhundert wird beschrieben? Wie war die Welt vorher?
6. Welche Rolle spielen im Text die folgenden Beriffe?
   - “chance”
   - “comparison”
   - “norms”
   - “visibility”
7. Finden Sie die Erzählung schlüssig? Was fehlt Ihnen?
8. Gibt es Momente in Ihrem Alltag, in denen Sie “metric power” erleben?