Einführung in R

Andrew Ellis Boris Mayer

19.03.25

Inhaltsverzeichnis

Vorwort

Voraussetzungen

Bitte installieren Sie sowohl die aktuelle Version von `R`: R version 4.4.3 (2025-02-28)

als auch RStudio: RStudio Desktop

Wir verwenden für diese Vorlesung RStudio, ein **integrated development environment** (IDE), welches die Arbeit mit `R` sehr angenehm macht. Das `R` Programm muss separat installiert werden, wir werden aber nicht direkt damit arbeiten.

Was ist `R`?

`R` ist sowohl eine Programmiersprache als auch eine Statistikumgebung. `R` ist open-source, d.h. der Source Code ist unter der GNU Public License frei verfügbar. Ausserdem ist `R` kostenlos.

Woher kommt der Name `R`? `R` wurde als open-source Variante einer kommerziellen Sprache entwickelt, welche S heisst (Programmiersprachen haben oft nur einen Buchstaben als Namen, z.B. C). Die beiden `R` Entwickler (Ross Ihaka und Robert Gentleman) nannten angeblich die Sprache `R`, weil ihre Vornamen beide mit dem Buchstaben R beginnen.

Wir werden `R` primär als Statistikumgebung kennenlernen, werden uns jedoch auch teilweise mit `R` als Programmiersprache beschäftigen. Das bedeutet, dass wir am Anfang verschiedene Datentypen und ein wenig `R` Syntax kennenlernen, damit wir richtig damit arbeiten können.

Die `R` Sprache gilt als relativ "schwierig" zu lernen, unter anderem, weil man sich viele verschiedene Funktionsnamen merken muss, und diese eine etwas inkonsistente Namensgebung haben. Wir orientieren uns deswegen, soweit möglich, an einer Sammlung von modernen `R` Packages (Erweiterungspakete), welche von RStudio, insbesondere von Hadley Wickham, entwickelt wurden.

Aber bitte lassen Sie sich nicht abschrecken. Diese Packages repräsentieren den 'state-of-the-art', was Datenanalyse anbelangt, und die Arbeit damit ist einfach erlernbar. Es gibt beinahe für alle Probleme ein dafür entsprechendes Paket mit einer Lösung. Nach wenigen Anwendungen wird die "Programmierung" intuitiv.

Weiterführende Literatur

Wir orientieren uns inhaltlich teilweise an dem Buch R für Einsteiger: Einführung in die Statistiksoftware für die Sozialwissenschaften von Maike Luhmann (2020), verwenden aber nicht deren `R` Code.

In Bezug auf `R` Code orientieren uns wir und an dem online frei verfügbaren Buch R for Data Science von Wickham, Çetinkaya-Rundel und Grolemund (2023). Dieses Buch ist jedoch weit umfangreicher, als wir es für diese Vorlesung brauchen.

Für diejenigen, welche sich mit `R` als Programmiersprache auseinandersetzen wollen, empfehlen wir die Bücher Hands-On Programming with R von Garrett Grolemund (2014) und Advanced R von Hadley Wickham (2019). Das letztere ist jedoch wirklich nur für Vertiefer.

DataCamp bietet verschiedene Online-Kurse an (teilweise kostenpflichtig). Dieser Einführungskurs ist jedoch kostenlos.

Typographische Konventionen

Wir verwenden zusätzlich zum Haupttext folgende Textblöcke:

> **Hinweis**
>
> In diesem Block stehen Kommentare und Erläuterungen.

> **Vertiefung**
>
> In diesem Block stehen vertiefende Zusatzinformationen.

> **Übung**
>
> In diesem Block stehen Übungen.

> **Lösung**
>
> In diesem Block stehen Lösungen.

Die Lehrmaterialien in den folgenden Kapiteln bestehen zu einem grossen Teil aus `R` Code. Code-Chunks sehen so aus:

```r
x <- seq(from = 1, to = 10, by = 1)
```

Dieser Code kann in der R Konsole ausgeführt werden. Code-Chunks können auch einen Output haben:

```
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

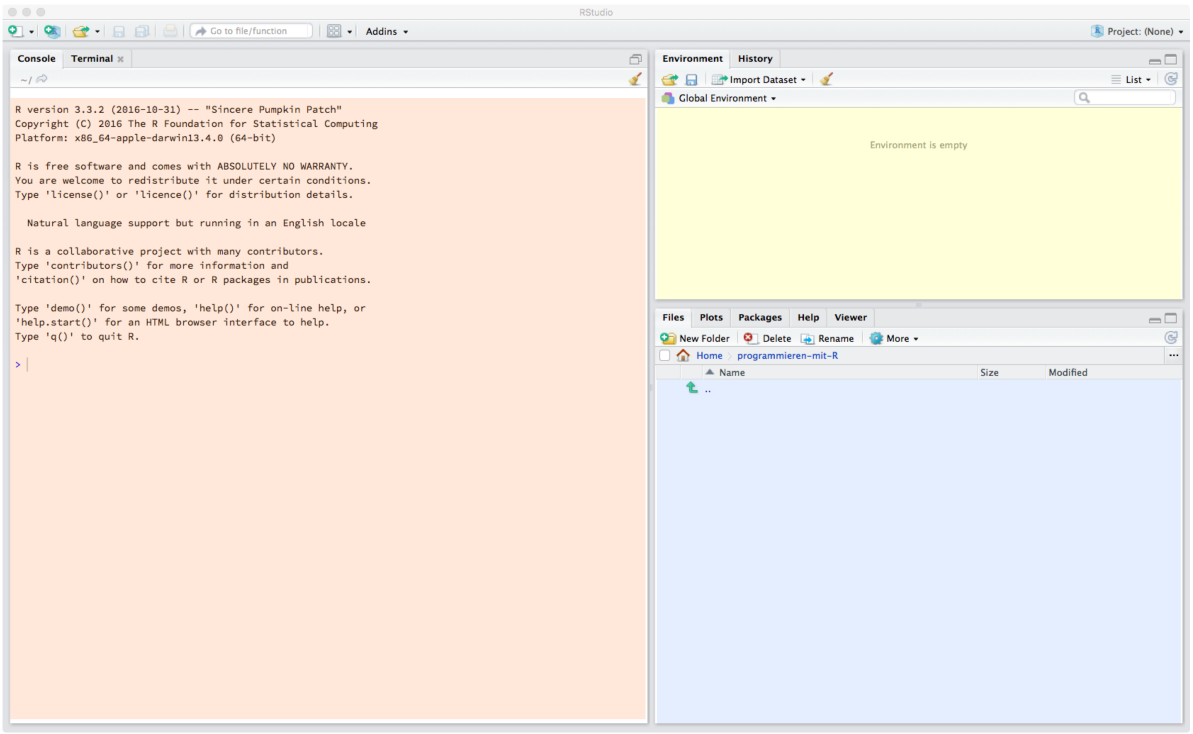
In einem solchen Block ist `x` der Input und `[1]  1  2  3  4  5  6  7  8  9 10` der Output (in diesem Beispiel haben wir eine Variable `x` kreiert und ihr die Sequenz von `1` bis `10` zugewiesen).

License

1 RStudio Workflow

1.1 Graphische Benutzeroberfläche

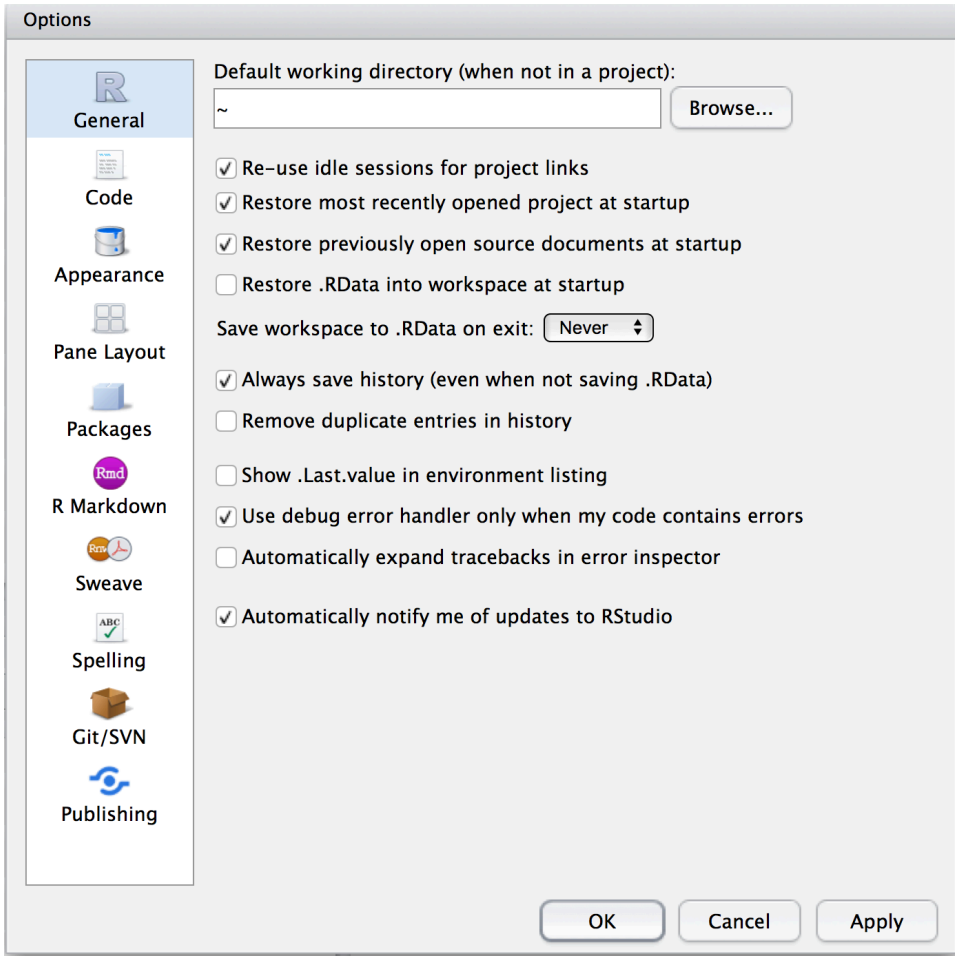
Bevor wir mit dem Inhalt anfangen, sollten wir uns ein wenig mit RStudio anfreunden. Öffnen Sie nun RStudio. Das Programm sollte ungefähr so aussehen:



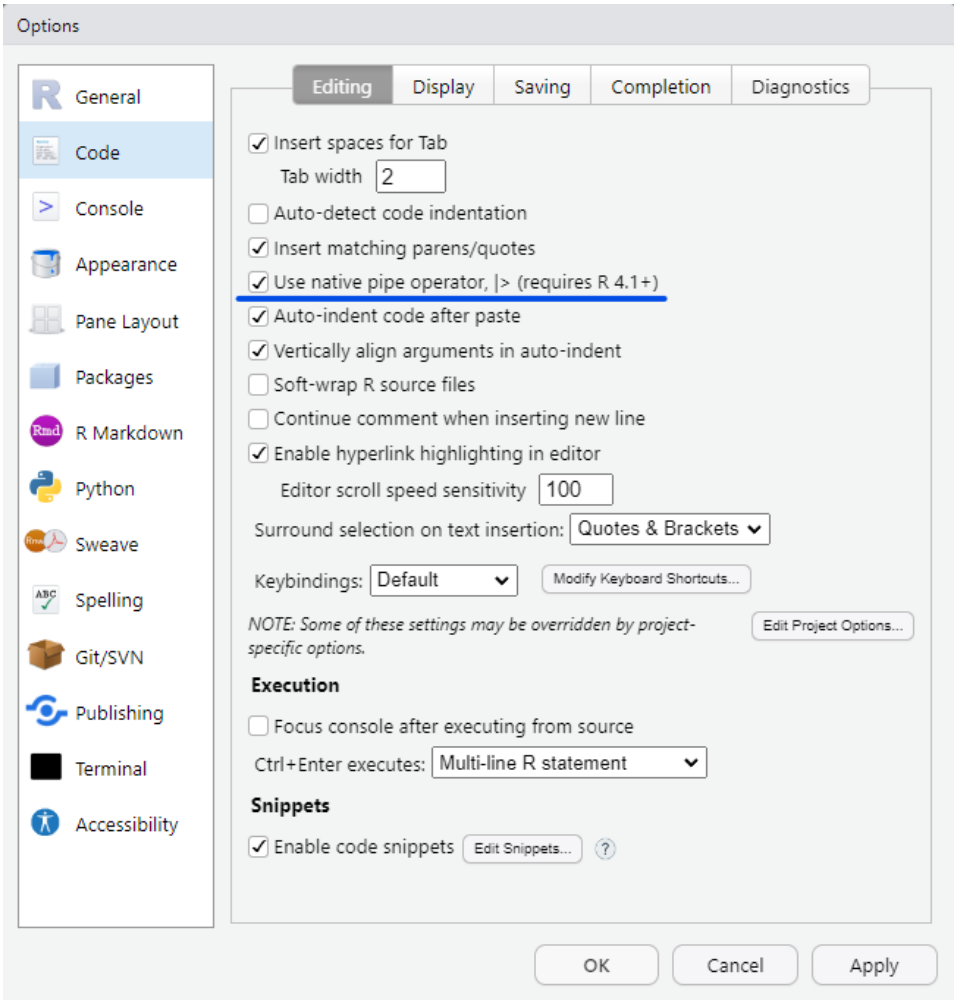
Unter dem Menu `Preferences` oder `Global Options` kann man RStudio den eigenen Präferenzen anpassen. Wir empfehlen aber dringend, folgende zwei Optionen **nicht** zu aktivieren:

1) Restore .RData into workspace at startup (sollte nicht ausgewählt sein)
2) Save workspace to .RData on exit (Never)

Diese Optionen sind zwar manchmal verlockend, da damit alle Daten und Variablen automatisch wieder geladen werden. Es ist aber meistens besser, mit einem leeren Workspace zu beginnen, und die ausgeführten `R` Befehle in einem Script oder Notebook festzuhalten.



Zudem müssen Sie in den Optionen unter `Code` die Option `use native pipe operator` aktivieren.



Das RStudio GUI (Graphical User Interface) besteht aus mehreren Bereichen. Auf der linken Seite (rot markiert) erscheint die **R Konsole**. Hier kann R Code eingegeben werden, der dann von R interpretiert (ausgeführt) wird. Das > Zeichen ist die sogenannte R Prompt (Aufforderungszeichen). Wir können so interaktiv mit R arbeiten, und R z.B. auch als Taschenrechner benutzen:

```
2 + 3
```

```
[1] 5
```

```
exp(1)
```

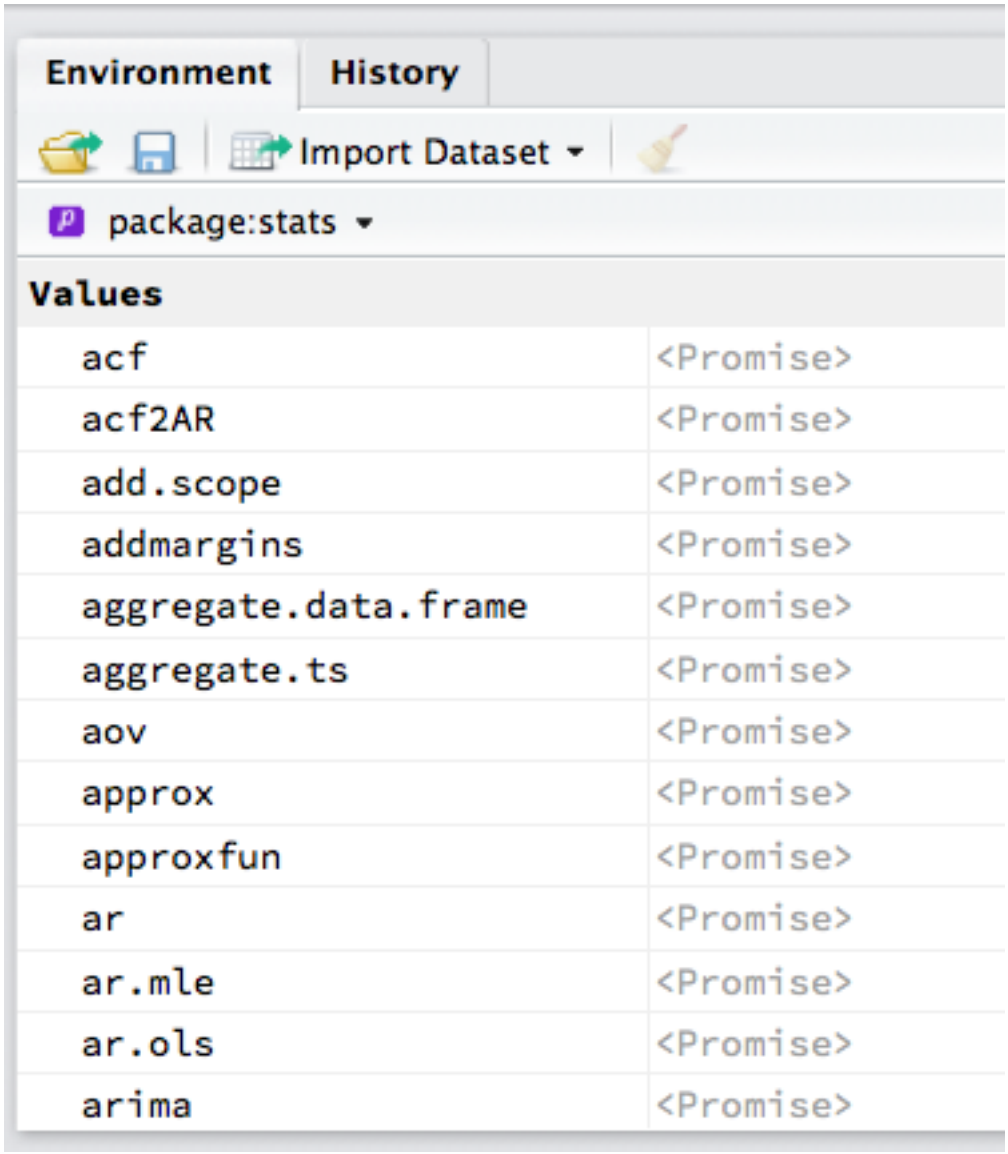
```
[1] 2.718282
```

```
3/4
```

```
[1] 0.75
```

> Hinweis
>
> Wenn wir `3/4` eingeben, erscheint im Output `[1] 0.75`. Das `[1]` bedeutet, dass der Output mit dem ersten Element eines Vektor beginnt. Dies ist gleichzeitig das einzige Element, und ist somit ein Skalar (ein Skalar in `R` ist einfach ein Vektor mit nur einem Element, d.h. mit der Länge 1).

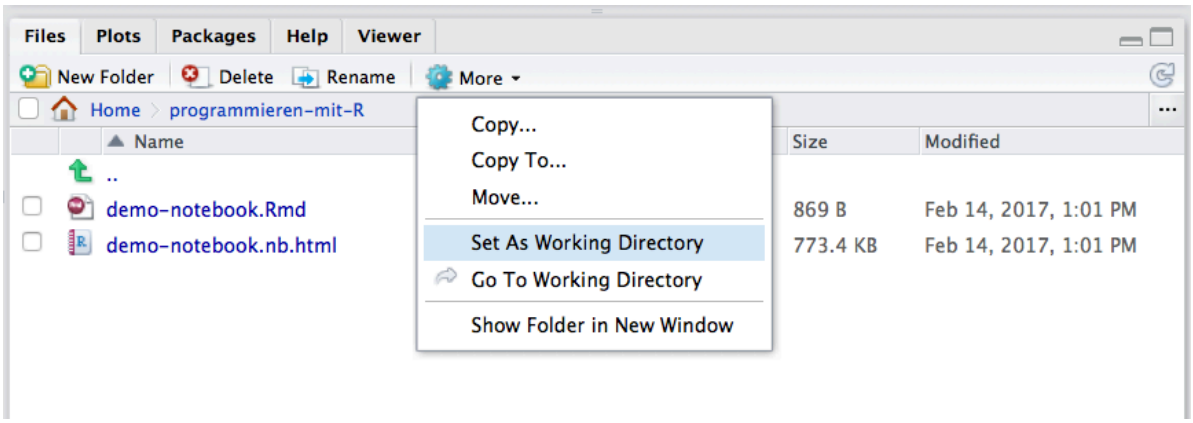
Rechts oben (gelb markiert) erscheinen zwei Bereiche, **Environment** und **History**. Im **Environment**-Bereich sehen wir alle Variablen, Datensätze und Funktionen, welche wir definiert haben. Diese erscheinen im **Global Environment** (Dropdown-Menü). Wenn Sie auf dieses Menü klicken, sehen Sie, welche **Packages** geladen sind. Wenn Sie z.B. das `stats` Package auswählen, sehen Sie alle Funktionen, die durch dieses Package verfügbar gemacht werden. Wir werden in Kürze mehr über Packages erfahren.



Im **History**-Bereich sehen Sie alle Befehle, die Sie bisher ausgeführt haben.

Rechts unten (blau markiert) erscheinen die Bereiche **Files** (Dateimanager), **Plots**, **Packages** und der **Help Viewer**.

Im Dateimanager können Sie das Arbeitsverzeichnis (working directory) wechseln und Dateien öffnen:



Hinweis

Wenn Sie im Dateimanager das Arbeitsverzeichnis wechseln, erscheint in der Konsole die Syntax für den Befehl, den `R` soeben ausgeführt hat. In diesem Fall ist das `setwd(...)`. Mit `getwd()` kann man `R` fragen, in welchem Arbeitsverzeichnis man sich befindet.

1.2 Packages

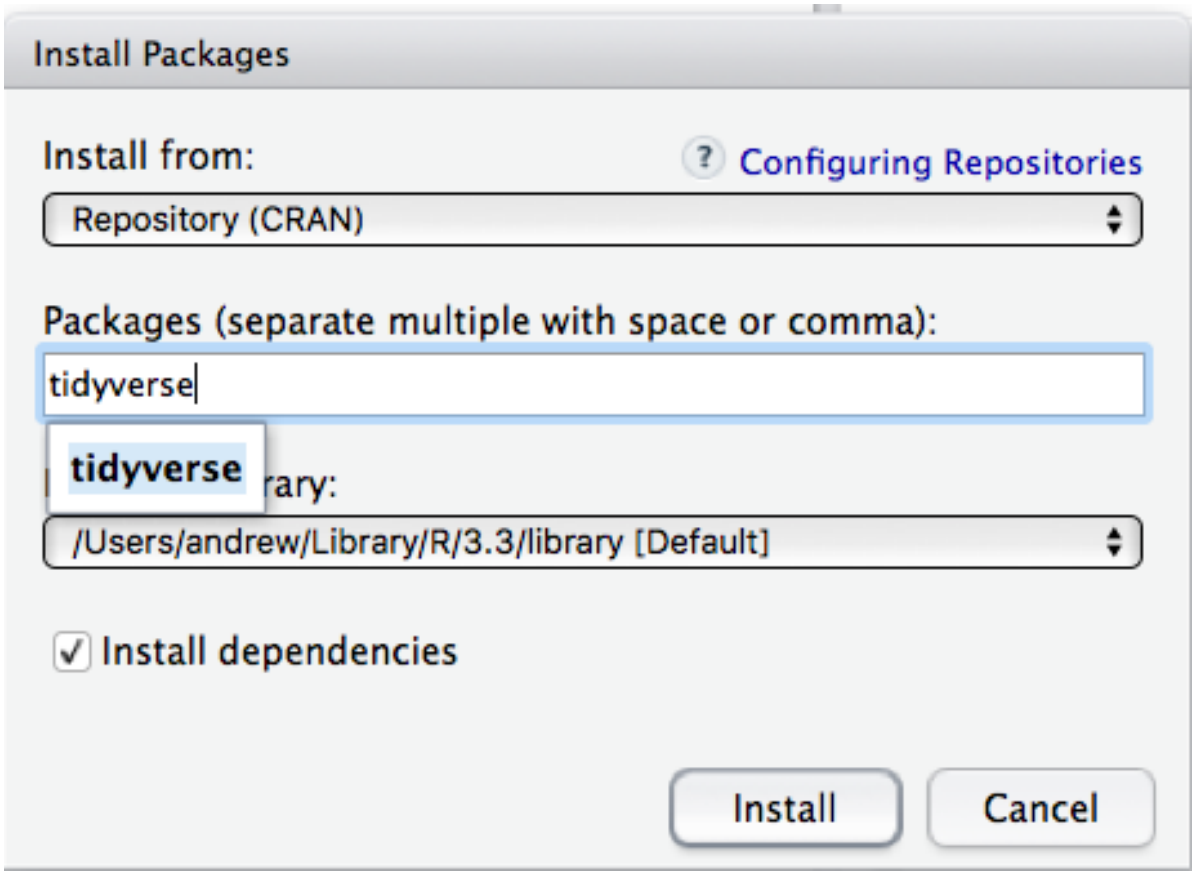
Bevor wir anfangen, mit RStudio zu arbeiten, müssen wir einige **Packages** installieren. Packages stellen zusätzliche Funktionen zur Verfügung, welche nicht in der Basisausstattung von `R` (base `R`) enthalten sind. Wir installieren zunächst eine Sammlung von Packages zur Datenmanipulation (`tidyr`, `dplyr`, `forcats`), zum Importieren von Datenfiles (`readr`) und für Grafiken (`ggplot2`). Da diese Packages alle Teil des Meta-Packages `tidyverse` sind, können sie alle zusammen mit folgendem Befehl installiert werden, welchen wir in die Konsole eingeben:

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

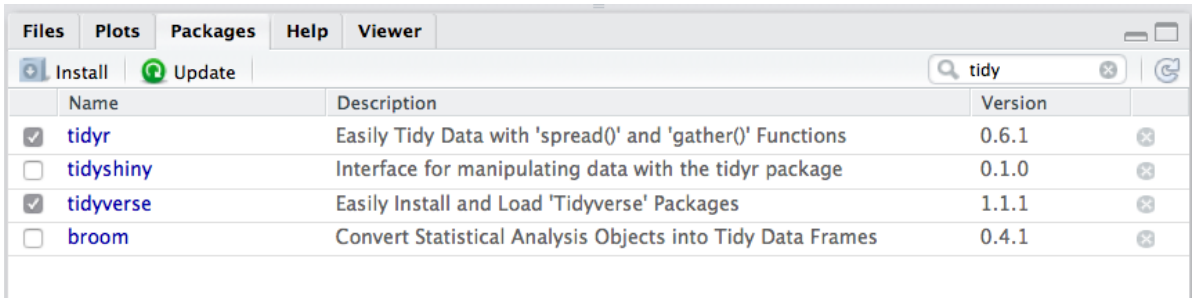
Anschliessend können wir die Packages so laden:

```
library(tidyverse)
```

Als Alternative dazu können Packages über das GUI installiert



und geladen werden.



Ausserdem können so alle Packages aktualisiert werden (**Update**) - es empfiehlt sich, dies regelmässig zu tun.

Hinweis

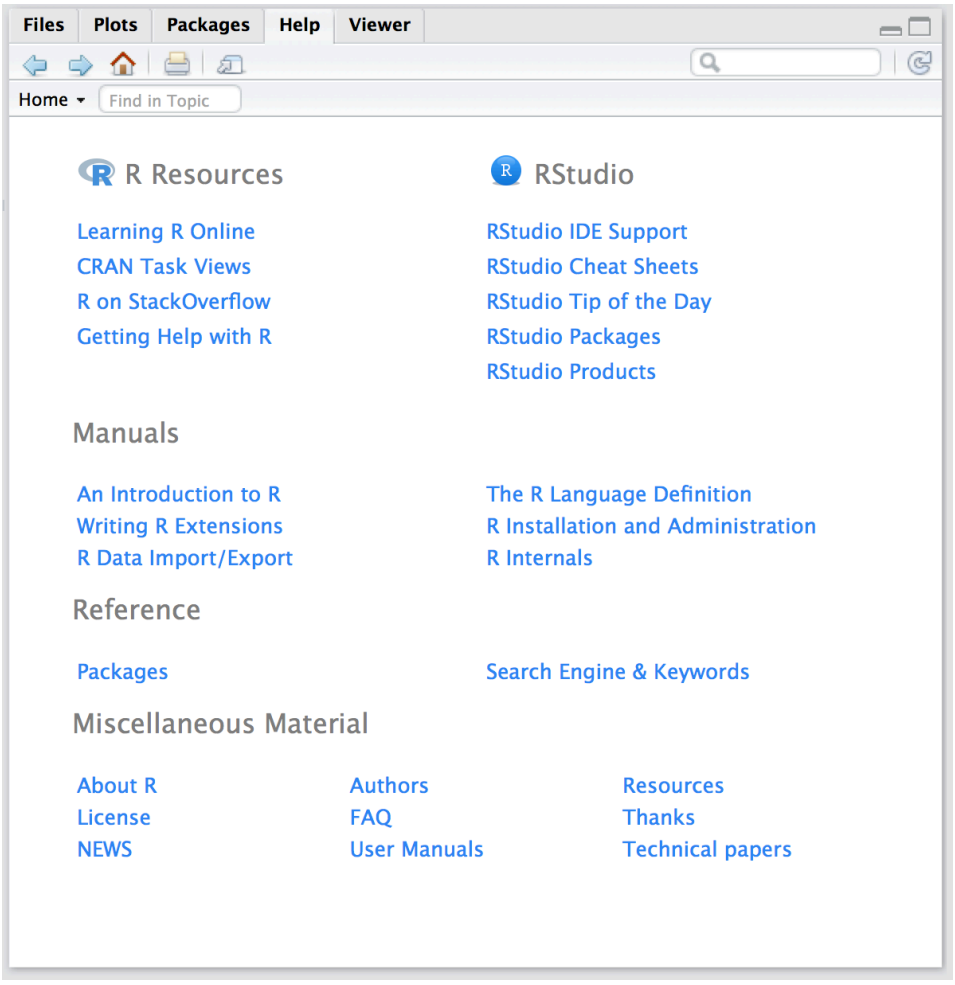
Bitte installieren Sie nun die `tidyverse` Packages. Weitere Packages werden wir installieren, wenn wir sie benötigen.

R Packages werden auf einem Server bereitgestellt: The Comprehensive R Archive Network, oder CRAN. Unter Task Views sehen Sie Sammlungen von Packages zu bestimmten Themen; die Task View zum Thema Psychometrik finden Sie hier: CRAN Task View: Psychometric Models and Methods.

Da die meisten Statistiker:innen mit R arbeiten, gibt es fast zu jedem Thema ein R Package, oder zumindest R Code, den man mit einer gezielten Suche im Internet finden kann (siehe auch nächster Abschnitt).

1.3 Help

Selbst wenn man täglich mit R arbeitet, ist es (fast) unmöglich, sich alle Funktionen zu merken. R hat ein sehr gutes eingebautes Hilfesystem, welches aber für Anfänger nicht leicht verständlich ist. Dies ist im Bereich **Help** zugänglich:



Sehr empfehlenswert ist die Question und Answer Website Stackoverflow - hier finden sich Antworten auf (fast) alle möglichen Fragen (es gibt meistens jemanden, der oder die das gleiche Problem schon einmal hatte).

Wenn man gezielt suchen will, kann man im Suchfenster des **Help** Viewers einen Begriff eingeben. Wenn man einfach herausfinden möchte, welche Funktionen in einem installierten Package verfügbar sind, kann man im **Packages**-Bereich nach einem Package suchen, und dann auf dieses klicken, um die Hilfeseiten zu sehen.

> Übung
>
> Probieren Sie es selber aus: suchen Sie nach dem Package `dplyr`. Wenn Sie darauf klicken, sehen sie zuoberst einen Link zu `User guides, package vignettes and other documentation`. Vignettes sind als Einführungen gedacht.

Wenn man in der Konsole schnell Hilfe zu einer Funktion erhalten will, kann man so vorgehen:

```
help(mean)
```

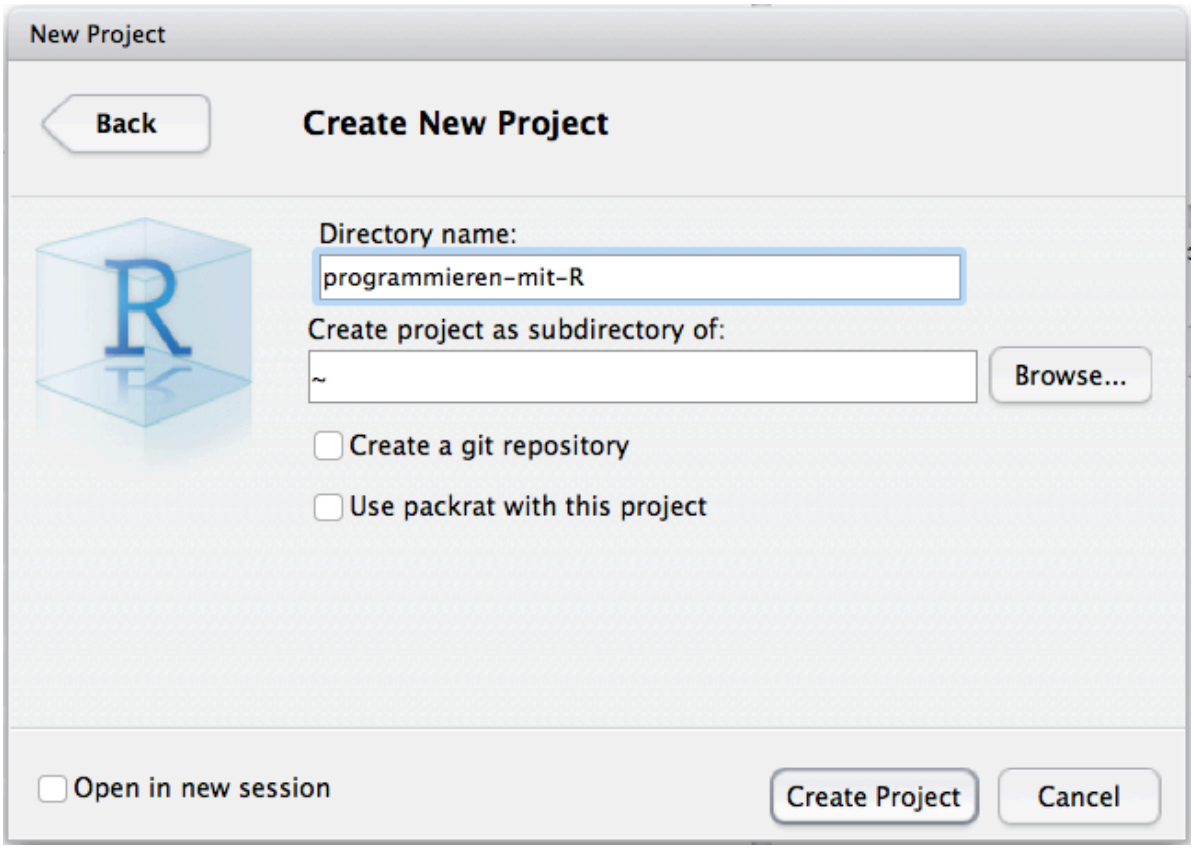
Dies öffnet die Hilfeseite für die `mean` Funktion. Alternativ dazu kann man auch `?mean` eingeben.

Man kann aber auch zu einem Thema Hilfe suchen. Wenn wir z.B. Hilfe zum Thema Zahlen auf- oder abrunden brauchen, können wir `help("round")` eingeben. Weitere Hilfe zur `help()` Funktion finden Sie so: `help(help)`.

1.4 Arbeiten mit RStudio

1.4.1 Projekte

Nun können wir mit der Arbeit beginnen. Es empfiehlt sich, ein **Projekt** anzulegen, über das Menu `File > New Project`. Sie können das Projekt nennen, wie Sie wollen - wir haben den Namen `programmieren-mit-R` gewählt.



Der Vorteil eines Projekts (im Vergleich zu einer einzelnen neuen R Script-Datei) besteht zunächst darin, dass RStudio beim einem Neustart wieder alle Files öffnet. Ausserdem kann man beliebig viele Projekte erstellen, und entweder gleichzeitig offen haben, oder zwischen ihnen wechseln. Des weiteren wird beim Öffnen eines Projekts das Arbeitsverzeichnis automatisch auf den Ordner definiert, in dem sich die Projekt-Datei befindet. Man muss sich also dann nicht mehr um das *working directory* kümmern und kann z.B. Daten- oder Skriptdateien direkt vom Projektordner aus aufrufen, ohne jedes Mal den vollständigen Pfad definieren zu müssen.

1.4.2 Konsole

Es gibt im Wesentlichen zwei Vorgehensweisen, wenn man mit `R` arbeiten will: man kann 1) entweder Befehle direkt in die Konsole eintippen, oder 2) eine Textdatei (`R` Script-Datei öffnen, Befehle in die Datei schreiben, und diese dann an die Konsole schicken.

Mit der Konsole arbeiten ist zu empfehlen, wenn man schnell etwas ausprobieren will, oder wenn man Befehle wiederholen will (dies kann man mit den `Up`/`Down`-Pfeiltasten machen), aber es ist meistens besser, wenn man mit einer Textdatei arbeitet.

Hinweis

Die **History** der eingegebenen Befehle kann man in der Konsole mit der Tastenkombination `CMD` + `Up` auf macOS oder `CTRL` + `Up` anzeigen lassen.

Manchmal passiert es, dass man einen Befehl unvollständig eingibt. `R` wartet dann auf weiteren Input, und zeigt dies mit einem `+` an.

Hier wurde z.B. die Klammer vergessen:

```
> mean(x
+
```

Nun muss man entweder die fehlende Klammer eingeben, um den Befehl zu vervollständigen, oder man kann mit der `ESCAPE`-Taste abbrechen.

Für den Fall, dass `R` abstürzt, kann man mit Hilfe des Menus **Session > Interrupt R** oder **Session > Terminate R** auswählen.

1.4.3 `R` Script

Wir öffnen und speichern ein neues `R` Script (Textdatei mit dem Suffix `.R`). Geben Sie hier nun z.B.

```
2 + 3
```

```
[1] 5
```

ein, wählen sie den Text aus, und klicken sie auf **Run**. Sie haben auch im Menü **Code** verschiedene Möglichkeiten (und Tastaturkürzel), um Code auszuführen.

Der Output erscheint in der Konsole.

Übung

Geben Sie folgendes ein: `x <- c(101, 105, 99, 87, 102, 98)`, und führen sie den Code aus.

Sie haben gerade einen Vektor definiert. Im Output erscheint

```
> x <- c(101, 105, 99, 87, 102, 98)
```

und im **Environment** sehen Sie, dass `x` ein `num [1:6]` ist. Dies bedeutet, dass `x` ein numerischer Vektor der Länge 6 ist.

Übung

Geben Sie nun auch folgendes ein: `boxplot(x)`, und führen sie den Code aus.

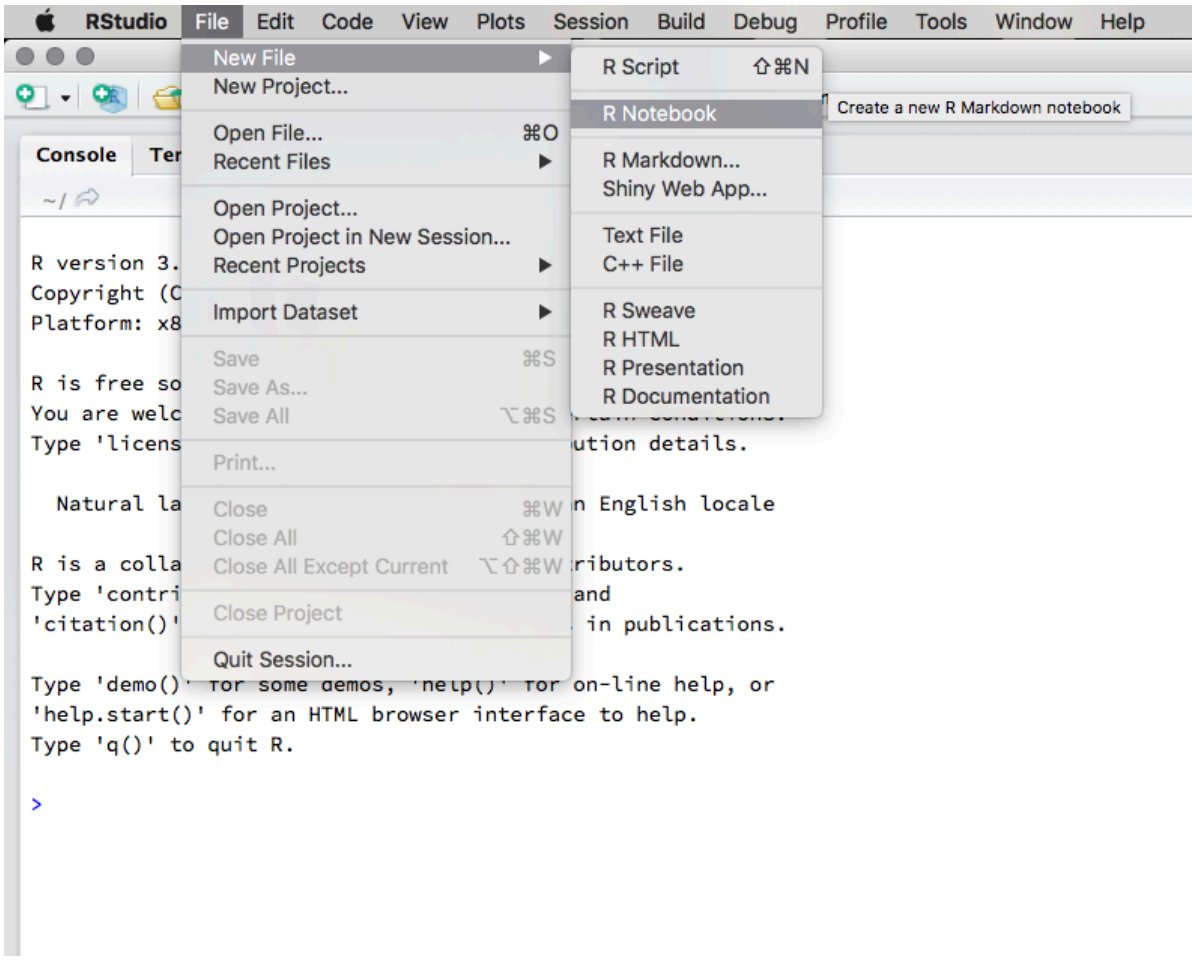
Der Boxplot erscheint im separaten **Plots** Viewer.

R Scripts eignen sich am besten, wenn man längere Programme schreiben oder ganze Datenanalysen speichern will. Für die interaktive Arbeit gibt es R Notebooks, welche wir uns als nächstes ansehen werden.

1.4.4 R Notebooks verwenden

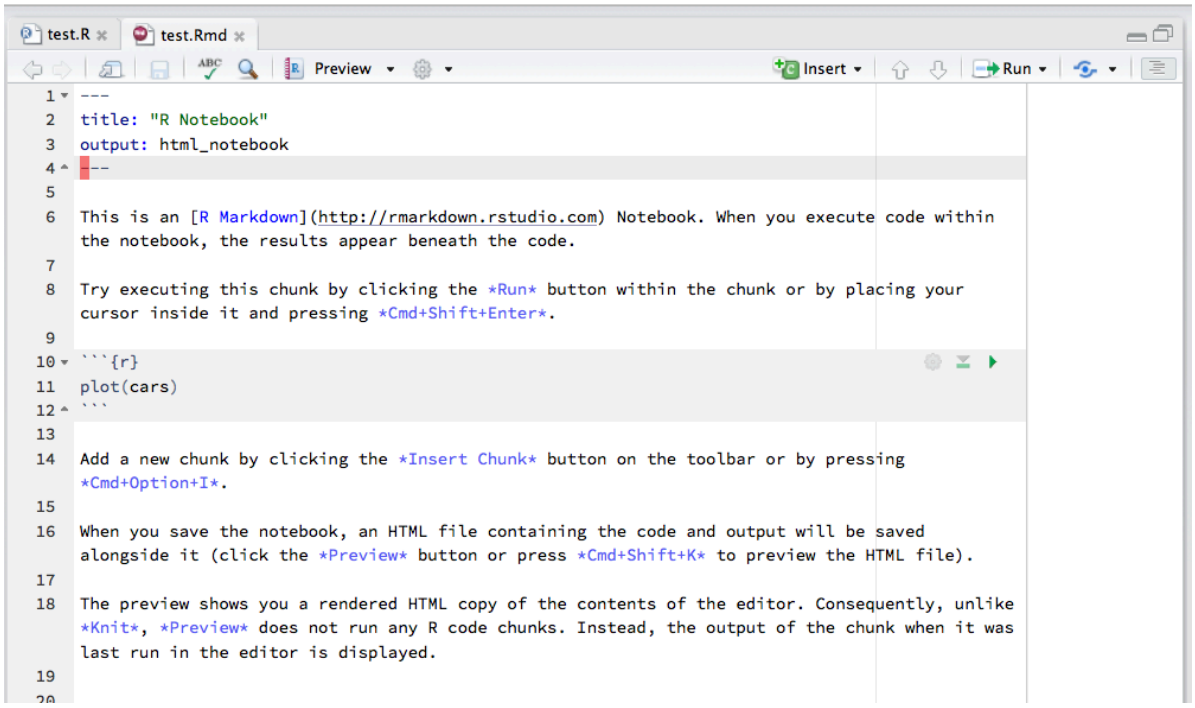
R Notebooks sind "RMarkdown-interaktiv" (RMarkdown ist eine simple Markup-Sprache), und zeigen Text (Prosa), R Code und Grafiken im selben Dokument an.

Öffnen Sie eine neues Notebook File:

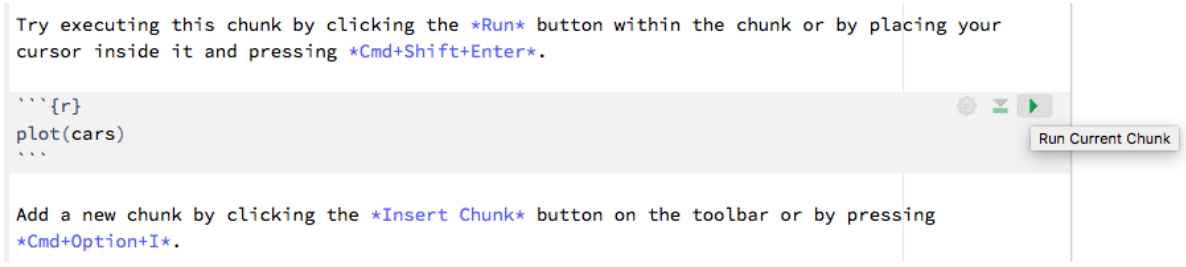


Sie können dies nun speichern. Notebooks (und alle RMarkdown Dokumente) erhalten das Suffix `.Rmd`.

Das Notebook sollte so aussehen:



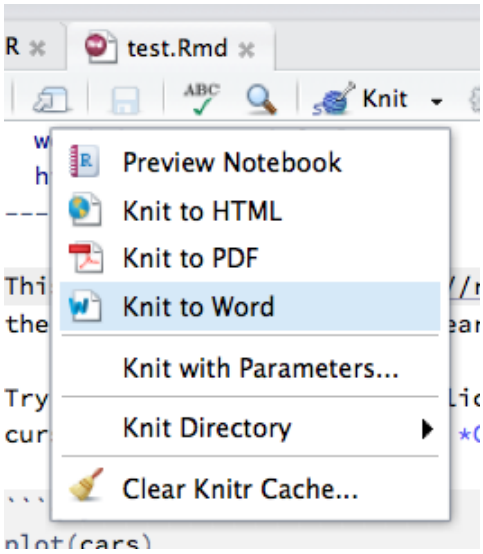
Im Notebook sehen Sie sowohl Text mit der R Markdown Markup-Sprache, als auch `R` Code-Chunks. Diese können ausgeführt werden.



Wenn Sie auf den grünen Pfeil klicken erscheint der Output direkt unter dem Code-Chunk.

Klicken Sie nun auf **Preview**, oder benützen Sie die entsprechende Tastenkombination: `CMD` + `Shift` + `K` (macOS) oder `CTRL` + `Shift` + `K` (Windows). RStudio wird Sie nun darauf hinweisen, dass fehlende `R` Packages installiert werden müssen (dies ist nur beim ersten mal

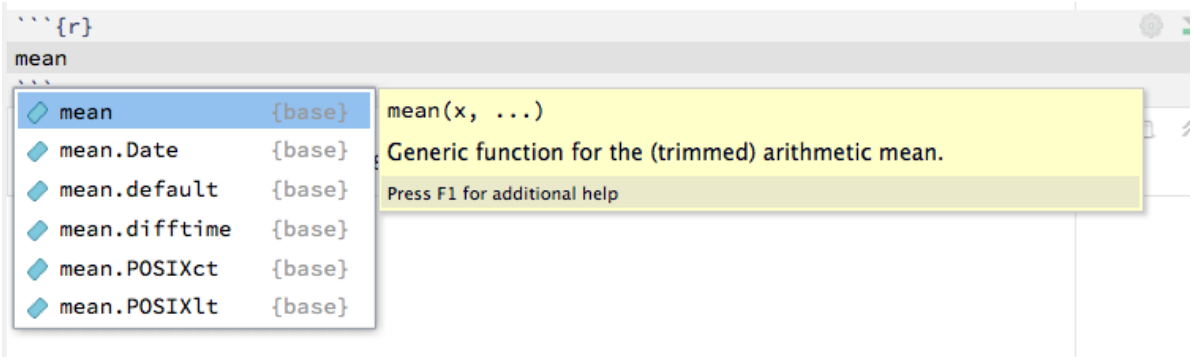
der Fall). Stimmen Sie der Installation zu, das Notebook wird dann anschliessend kompiliert, und es erscheint ein HTML Dokument im **Viewer**. Dieses beinhaltet sowohl Text als auch den evaluierten `R` Code, und kann als eigenständiges Dokument weitergegeben werden. Ein weiterer Vorteil eines Notebooks: Sie können daraus ein Word-Dokument erstellen:



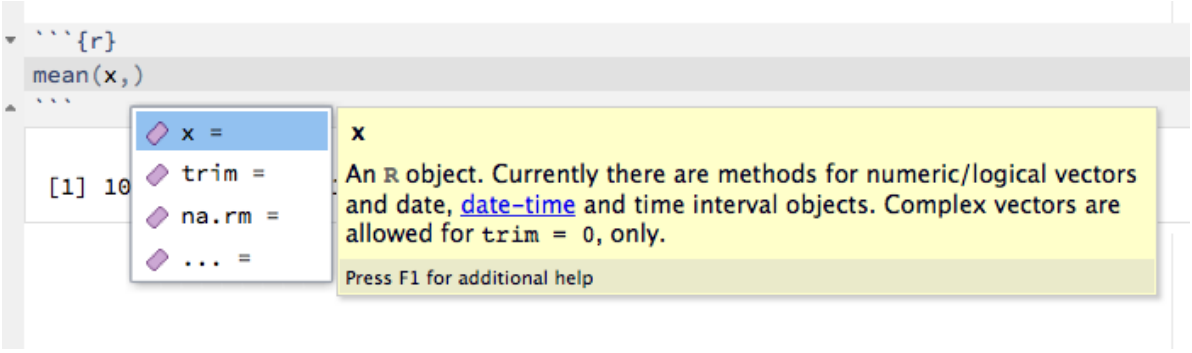
Probieren Sie es selber aus.

1.4.5 Tab completion

RStudio verfügt über eine weitere sehr hilfreiche Funktion: **Tab completion**. Wenn man in der Konsole oder in einem Script/Notebook den Anfang eines Befehls eingibt, und dann die **Tabulator**-Taste drückt, erscheint ein Menu mit möglichen **completions**. Wenn wir z.B. einen Mittelwert berechnen wollen, schreiben wir `mean`, und drücken dann `Tab`. Es erscheint eine Liste mit möglichen Optionen.



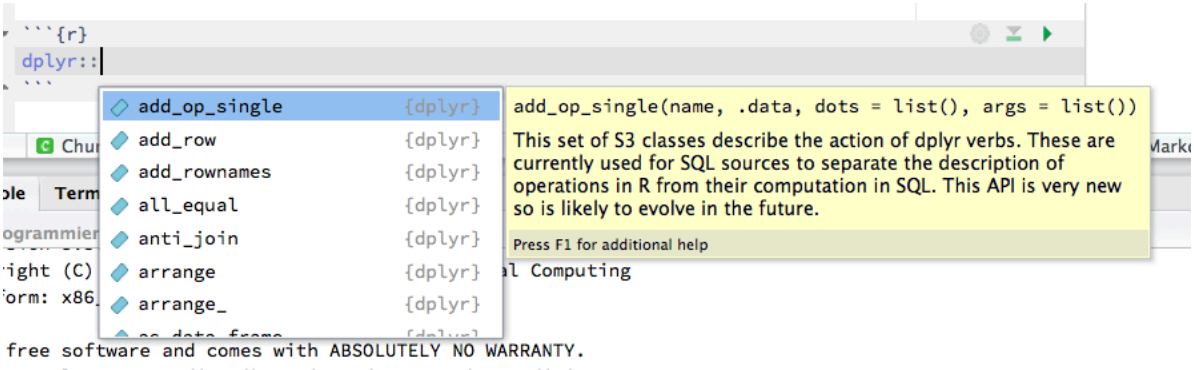
Wenn man ausserdem eine Funktion eingegeben hat, und innerhalb der Klammern `Tab` drückt, erscheint eine Liste mit den Argumenten dieser Funktion.



Eine weitere nützliche Funktion: wenn wir nicht sicher sind, wie eine Funktion heisst, aber wir wissen, in welchem Package wir sie finden, können wir den Package Namen schreiben, gefolgt von `::`, und dann `Tab` drücken. RStudio gibt uns eine Liste mit allen Funktion dieses Packages. Schreiben Sie:

```
dplyr::
```

drücken `Tab`, und RStudio übernimmt den Rest.



1.4.6 Tasten

Wenn wir mit `R` arbeiten, werden wir folgende Zeichen häufig brauchen:

```
[ ]   Square braces (eckige Klammern)
{ }   Curly braces (geschweifte Klammern)
$     Dollar key (Dollarzeichen - für das Subsetting/Auswählen von Variablen benötigt)
```

```
#     Hash (pound) key (Rautezeichen - für Kommentare in R Script Dateien)
~     Tilde (für die Model Notation/Modellformulierung)
|     Vertical bar (senkrechter Strich - als logischer Operator benötigt, siehe nächstes Kap
`     Backtick (Gravis-Akzent - wird hauptsächlich für code chunks benötigt, muss selten manu
```

Leider sind manche dieser Zeichen auf Tastaturen mit Schweizerischem Tastaturlayout nicht leicht auffindbar.

Übung

Nehmen Sie sich ein paar Momente Zeit, diese Zeichen auf Ihrer Tastatur zu suchen.

Hinweis

Die öffnende eckige Klammer erhält man mit `ALT + 5` (praktischerweise fügt `R` die öffnende und die schliessende Klammer in freudiger Erwartung eines Befehls gleich zusammen ein), die schliessende mit `ALT` + `6`. Für geschweifte Klammern benutzen Sie analog `ALT` + `8` und `ALT` + `9`. Das Rautezeichen erhalten Sie mit `ALT` + `3`, die Tilde mit `ALT` + `N`, den senkrechten Strich mit `ALT` + `7`.

2 Die R Sprache

2.1 Operatoren und Funktionen

Als erste Anwendung können wir R als Taschenrechner benützen. Zuvor müssen wir uns aber noch ein wenig Vokabular aneignen, und dann ein bisschen Syntax lernen.

Zum Basisvokabular gehören einige eingebaute Operatoren (sowohl arithmetische als auch logische).

2.1.1 Arithmetische Operatoren

Die ersten fünf Operatoren sollten selbsterklärend sein:

```
+                Addition
-                Subtraktion
*                Multiplikation
/                Division
^ oder **        Potenz

x %*% y          Matrixmultiplikation c(5, 3) %*% c(2, 4) == 22
x %% y           Modulo (x mod y) 5 %% 2 == 1
x %/% y          Ganzzahlige Teilung: 5 %/% 2 == 2
```

> Hinweis
>
> Die letzten drei Operatoren sind vielleicht nicht allen geläufig. `%*%` ist der Operator für die Matrixmultiplikation. Der einfachste Fall ist die Multiplikation zweier gleich langer Vektoren (eines Zeilen- und eines Spaltenvektors). Dabei werden die Elemente der beiden Vektoren elementweise miteinander multipliziert und anschliessend die Produkte aufsummiert. Ergebnis ist ein Skalar (eine einzelne Zahl bzw. ein Vektor mit nur einem Element), z.B. `c(5, 3) %*% c(2, 4)` ergibt 22 (5*2 + 3*4). `%%` ist der Modulo Operator und gibt den Rest nach einer ganzzahligen Division an. So gibt z.B. `5 %% 2` (sprich 5 modulo 2) den Wert `1`. `%/%` gibt das Resultat der ganzzahligen Division, d.h. `5 %/% 2` gibt den Wert `2` (wie oft kann man 2 von 5 subtrahieren?). Diese Operatoren werden beim Program-

mieren oft gebraucht, spielen in diesem Einführungskurs aber nur eine untergeordnete Rolle.

2.1.2 Logische Operatoren und Funktionen

Die letzten beiden Funktionen, `xor()` und `isTRUE()`, werden Sie wahrscheinlich selten brauchen, sie sind nur der Vollständigkeit halber aufgelistet.

```
<               Kleiner
<=              Kleiner gleich
>               Grösser
>=              Grösser gleich
==              Gleich (testet auf Äquivalenz)
!=              Ungleich
!x              Nicht x (Verneinung)
x | y           x ODER y
x & y           x UND y

xor(x, y)       Exklusives ODER (entweder in x oder y, aber nicht in beiden)
isTRUE(x)       testet ob x wahr ist
```

Die folgende Grafik zeigt die Verwendung der logischen Operatoren anhand von Venn-Diagrammen. `x` bezieht sich dabei immer den linken Kreis, `y` auf den rechten. Der ausgewählte Bereich ist immer dunkel dargestellt.

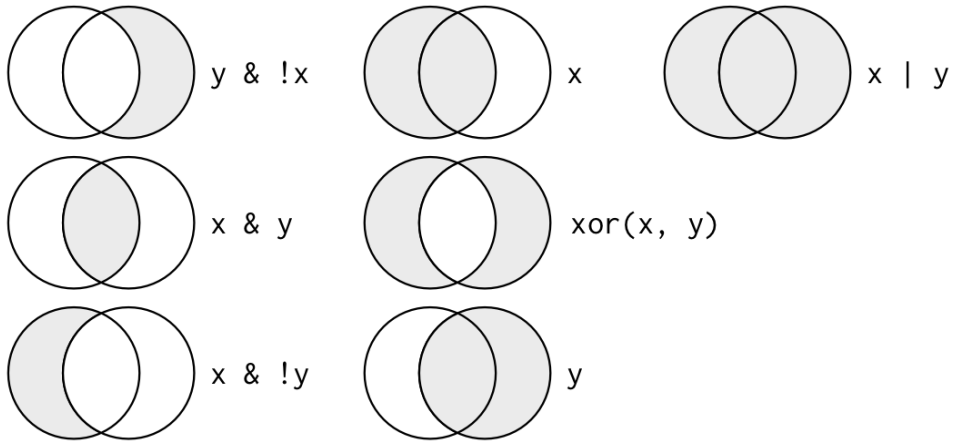


Abbildung 2.1: Veranschaulichung der logischen Operatoren. Aus *R for Data Science* (Wickham & Grolemund, 2017).

Es kommen noch weitere numerische Funktionen hinzu. Diese werden jedoch als Funktionen gebraucht, und nicht als Operatoren.

> Hinweis
>
> Operatoren werden so gebraucht: `1 + 2` (sie stehen zwischen den Operanden). Funktionen sehen so aus: `abs(x)` (sie werden auf Argumente angewendet).

2.1.3 Numerische Funktionen

```
abs(x)              Betrag
sqrt(x)             Quadratwurzel
ceiling(x)          Aufrunden: ceiling(3.475) ist 4
floor(x)            Abrunden: floor(3.475) ist 3
round(x, digits = n) Runden: round(3.475, digits = 2) ist 3.48
log(x)              Natürlicher Logarithmus
log(x, base = n)    Logarithmus zur Basis n
log2(x)             Logarithmus zur Basis 2
log10(x)            Logarithmus zur Basis 10
exp(x)              Exponentialfunktion: e^x
```

> Hinweis
>
> `round()` verwendet das *mathematische Runden*, nicht das im Alltag übliche *kaufmännische Runden*. Beim *mathematischen Runden* wird eine 5 (als letzte Ziffer) **nicht immer** aufgerundet, sondern nur dann, wenn die Ziffer davor eine *ungerade* Zahl ist (z.B. `round(3.475, digits = 2) == 3.48`). Ist die Ziffer vor der 5 dagegen eine *gerade* Zahl, wird abgerundet (`round(3.465, digits = 2) == 3.46`). Da die letzte Ziffer des gerundeten Resultats dann immer eine **gerade Zahl** ist, wird diese Rundungsmethode im Englischen auch als "rounding half to even" bezeichnet. Der Vorteil der mathematischen Rundung ist, dass mit dieser Methode beim Weiterrechnen (z.B. Bildung einer Summe gerundeter Zahlen) geringere Fehler verbunden sind als beim kaufmännischen Runden.

> Vertiefung
>
> Das ist aber noch nicht alles: Da `R` ein Computerprogramm ist, werden (reelle) Zahlen dort als Gleitkommazahlen *(floating point numbers)* repräsentiert und tatsächlich wendet `round()` die Rundungsfunktion auf diese Gleitkommazahlen an. Bei 64-bit Computersystemen, zu denen fast alle heute verwendeten Betriebssysteme gehören, werden reelle Zahlen als *double precision floating point numbers* oder kurz *doubles* repräsentiert. Aus diesem Grund werden numerische Vektoren in `R` auch als `double` bezeichnet (siehe Abschnitt

zu numerischen Vektoren). Nach der oben beschriebenen mathematischen Rundungsregel sollte `round(0.15, digits = 1)` eigentlich `0.2` ergeben, als Resultat erhalten Sie aber `0.1` (wenn Sie ein 64-bit Computersystem haben). Der Grund ist, dass `0.15` als `double` intern als `0.14999999999999999444888487687421729788184165954589843750` abgespeichert ist und daher abgerundet wird (bei einem 32-bit Computersystem ist es als *single precision floating point number* als `0.150000005960464477539062` gespeichert und wird daher aufgerundet; das können Sie hier nachprüfen). Das ist aber die Ausnahme: Bei den meisten (Dezimal-)Zahlen verhält es sich so, dass die (interne) *double*-Definition der "rounding half to even"-Regel entspricht und daher korrekt *mathematisch* gerundet wird.

Vertiefung

Es gibt in `R` streng genommen keinen Unterschied zwischen Operatoren und Funktionen. Jeder Operator ist eigentlich eine Funktion mit einer speziellen Infix-Notation. So kann z.B. der Operator `+` auch als Funktion benützt werden, allerdings muss dieser dann mit ‘ ‘ (backticks) umgeben werden. Sie können dies in der Konsole ausprobieren.

```
# Als Operator mit Infix Notation
2 + 3
```

```
[1] 5
```

```
# Als Funktionsaufruf
`+`(2, 3)
```

```
[1] 5
```

```
# Die beiden Schreibweisen sind äquivalent
2 + 3 == `+`(2, 3)
```

```
[1] TRUE
```

2.1.4 `R` als Taschenrechner

Wir können nun mit `R` rechnen. Im nächsten Code-Chunk sehen Sie einige Beispiele. `R` verwendet den hash key `#` als Kommentarzeichen, d.h. Zeilen, welche mit einem `#` beginnen, werden nicht evaluiert, und werden dazu benützt, den Code zu kommentieren.

Führen Sie diese Befehle aus, in dem Sie die einzelnen Zeilen auswählen, und dann gleichzeitig die `CMD` + `Enter` (macOS) oder `CTRL` + `Enter` (Windows) Tasten drücken.

```r
# Addition
5 + 5
```

```
[1] 10
```

```r
99 + 89
```

```
[1] 188
```

```r
12321 + 34324324
```

```
[1] 34336645
```

```r
# Subtraktion
6 - 5
```

```
[1] 1
```

```r
5 - 89
```

```
[1] -84
```

```r
# Multiplikation
3 * 5
```

```
[1] 15
```

```r
34 * 54
```

```
[1] 1836
```

```r
# Division
4 / 9
```

```
[1] 0.4444444
```

```r
(5 + 5) / 2
```

```
[1] 5
```

```r
# Operatorpräzedenz: Klammern beachten
# Punktrechnung vor Strichrechnung

(3 + 7 + 2 + 8) / (4 + 11 + 3)
```

```
[1] 1.111111
```

```r
1/2 * (12 + 14 + 10)
```

```
[1] 18
```

```r
1/2 * 12 + 14 + 10
```

```
[1] 30
```

```r
# Potenzieren
3^2
```

```
[1] 9
```

```r
2^12
```

```
[1] 4096
```

```r
# Exponentialfunktion
exp(5)
```

```
[1] 148.4132
```

```r
# Das nächste Resultat ist so gross,
# dass es in wissenschaftlicher Notation erscheint:
# 1.068647 * 10^13
exp(30)
```

```
[1] 1.068647e+13
```

```r
# Ganzzahlige Division
# 28 ist vier mal durch 6 teilbar, mit Rest 4
28 %% 6 # Rest: 4
```

```
[1] 4
```

```r
28 %/% 6 #  vier mal teilbar
```

```
[1] 4
```

```r
5 %% 2 # Rest: 1
```

```
[1] 1
```

```r
5 %/% 2 # zwei mal teilbar
```

```
[1] 2
```

```r
# Logische Operatoren
3 > 2
```

```
[1] TRUE
```

```r
4 > 5
```

```
[1] FALSE
```

```r
4 < 4
```

```
[1] FALSE
```

```r
4 <= 4
```

```
[1] TRUE
```

```r
5 >= 5
```

```
[1] TRUE
```

```r
6 != 6
```

```
[1] FALSE
```

```r
9 == 5 + 4
```

```
[1] TRUE
```

```r
!(3 > 2)
```

```
[1] FALSE
```

```r
(3 > 2) & (4 > 5) # UND
```

```
[1] FALSE
```

```r
(3 > 2) | (4 > 5) # ODER
```

```
[1] TRUE
```

```r
xor((3 > 2), (4 > 5))
```

```
[1] TRUE
```

> Hinweis
>
> Bei `xor()` darf nur *entweder der eine oder der andere* Ausdruck wahr sein, damit der gesamte Ausdruck wahr ist (exklusives ODER, ausschliessende Disjunktion), während bei `|` *mindestens einer* der Ausdrücke wahr sein muss, damit der Gesamtausdruck wahr ist (inklusives ODER, nicht-ausschliessende Disjunktion). Während also `(3 > 2) | (4 > 5)` und `xor((3 > 2), (4 > 5))` beide `TRUE` ergeben, ergibt `(3 > 2) | (5 > 4)` `TRUE`, aber `xor((3 > 2), (5 > 4))` ergibt `FALSE`.

Übung

Berechnen Sie:

1) $\frac{1}{3} \cdot \frac{1+3+5+7+2}{3+5+4}$
2) e (wie lässt sich das berechnen?)
3) $\sqrt{2}$
4) $\sqrt[3]{8}$
5) $log_2(8)$

Lösung

```
(1/3) * (1 + 3 + 5 + 7 + 2) / (3 + 5 + 4)
```

```
[1] 0.5
```

```
exp(1)
```

```
[1] 2.718282
```

```
sqrt(2)
```

```
[1] 1.414214
```

```
8^(1/3)
```

```
[1] 2
```

```
log(8, base = 2)
```

```
[1] 3
```

```
log2(8)
```

```
[1] 3
```

2.1.5 Statistische Funktionen

Hier ist eine Auflistung statistischer Funktionen, die alle auf einen Vektor angewendet werden können (z.B. zur Berechnung des arithmetischen Mittels einer Messwertreihe). Diese Funktionen haben alle das Argument `na.rm`, welches standardmässig den Wert `FALSE` annimmt. Das bedeutet, dass mögliche fehlende Werte (na = not available) in diesem Fall *nicht* entfernt (rm = remove) werden, bevor die Funktion ausgeführt wird. Das führt allerdings dazu, dass diese Funktionen im Falle vorhandener `NA`s selbst das Ergebnis `NA` liefern, da sie nicht auf fehlende Werte angewendet werden können (vgl. Beispiel unten). Es empfiehlt sich daher, diese Funktionen mit dem Argument `na.rm = TRUE` zu verwenden. So werden fehlende Werte vor Anwendung einer Funktion entfernt, und die Funktion somit nur auf gültige Werte angewendet. Am besten ergänzt man solche Funktionen zur Berechnung des Mittelwerts, der Varianz etc. durch eine Funktion zur Zählung der fehlenden Werte in einem Vektor, da sonst nicht klar ist, wie gross die Zahl/der Anteil gültiger Werte bei der Berechnung der Statistik war. Darauf werden wir weiter unten zurückkommen.

```
mean(x, na.rm = FALSE)  Mittelwert
sd(x)                   Standardabweichung
var(x)                  Varianz

median(x)               Median
quantile(x, probs)      Quantile von x.  probs: Vektor mit Wahrscheinlichkeiten

sum(x)                  Summe
min(x)                  Minimalwert x_min
max(x)                  Maximalwert x_max
range(x)                x_min und x_max
```

2.1.6 Zentrieren und standardisieren mit `scale()`

Mit der Funktion `scale()` kann ein Vektor, d.h. eine Reihe von Messwerten zentriert oder z-standardisiert werden. Bei der Zentrierung (Argument: `center = TRUE`) wird ein Vektor mit Abweichungswerten vom Mittelwert ausgegeben, bei der z-Standardisierung (mit dem zusätzlichen Argument: `scale = TRUE`) werden die Abweichungswerte noch durch die Stichprobenstandardabweichung geteilt, so dass standardisierte Werte entstehen.

Da `center = TRUE` und `scale = TRUE` die Default-Einstellungen der Funktion `scale()` sind, berechnet die Funktion ohne explizite Angabe der Argumente z-standardisierte Werte. Wenn man nur zentrieren will, muss `scale = FALSE` gesetzt werden. Wenn man (in seltenen Fällen) die ursprünglichen Werte durch die Standardabweichung teilen will, aber ohne vorherige Zentrierung, muss `center = FALSE` gesetzt werden.

```
# wenn center = TRUE: zentrieren
# wenn scale = TRUE: durch SD teilen
scale(x, center = TRUE, scale = FALSE)  Zentrieren
scale(x, center = TRUE, scale = TRUE)   Standardisieren (Default-Einstellung)
```

2.1.7 Ziehen einer Zufallsstichprobe mit `sample()`

Mit der Funktion `sample()` kann eine (Pseudo-)Zufallsstichprobe von Werten/Elementen aus dem Vektor `x` (Datenargument, "Urne") der Grösse `size` gezogen werden, und zwar "mit Zurücklegen" (`replace = TRUE`) oder "ohne Zurücklegen" (`replace = FALSE`, default). Ohne Angabe des Argumentes `prob` werden alle Elemente aus `x` mit gleicher Wahrscheinlichkeit gezogen. Wenn gewichtet gezogen werden soll, muss in `prob` ein Vektor mit Wahrscheinlichkeiten für jedes Element in `x` definiert werden. Statt Wahrscheinlichkeiten, die sich zu `1` addieren, können auch Zahlen verwendet werden, die die Verhältnisse "der Kugeln in der Urne" widerspiegeln. Mit anderen Worten: `sample()` rechnet die in `prob` angegebenen Werte automatisch in Wahrscheinlichkeiten um. Um keine Fehler zu machen, empfiehlt es sich aber, die Wahrscheinlichkeiten immer explizit anzugeben.

Wichtig: Beim Ziehen ohne Zurücklegen (`replace = FALSE`) darf die in `size` angegebene Grösse der Zufallsstichprobe nicht grösser sein als die Anzahl der in `x` enthaltenen Elemente. Wenn `size` und `length(x)` genau gleich gross sind, ist das Ergebnis von `sample()` eine zufällige Permutation der Elemente in `x`.

Tipp: Wenn man das Ergebnis einer Zufallsziehung reproduzierbar machen will, muss vor Ausführung von `sample()` mit der Funktion `set.seed()` ein beliebiger Startwert (z.B. `123` oder `5983467`) angegeben werden. Wird vor der nächsten Durchführung der (gleichen) Zufallsziehung derselbe Startwert wieder gesetzt, erhält man wieder dasselbe Ergebnis. Daran kann man erkennen, dass hinter der Funktion `sample()` kein wahrer Zufallsvorgang steckt, sondern damit ein komplizierter Algorithmus in Gang gesetzt wird, der einen Zufallsvorgang simuliert (und der - wenn man am selben Punkt "einsteigt" - immer zum selben, nichtzufälligen Ergebnis kommt).

```
sample(x, size, replace, prob)  Ziehen mit/ohne Zurücklegen.
prob: Vektor mit Wahrscheinlichkeiten
```

2.1.8 Weitere nützliche Funktionen

```
c()                   Combine: kreiert einen Vektor
seq(from, to, by)     Generiert eine Sequenz
:                     Colon Operator: generiert eine reguläre Sequenz
                      (d.h. Sequenz in Einerschritten)
rep(x, times, each)   Wiederholt x
```

```
                        times: die Sequenz wird n-mal wiederholt
                        each: jedes Element wird n-mal wiederholt

head(x, n = 6)          Zeigt die n ersten Elemente von x an
tail(x, n = 6)          Zeigt die n letzten Elemente von x an
```

2.1.9 Beispiele

```
c(1, 2, 3, 4, 5, 6)
```

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

```
mean(c(1, 2, 3, 4, 5, 6))
```

```
[1] 3.5
```

```
mean(c(1, NA, 3, 4, 5, 6), na.rm = TRUE)
```

```
[1] 3.8
```

```
mean(c(1, NA, 3, 4, 5, 6), na.rm = FALSE)
```

```
[1] NA
```

```
sd(c(1, 2, 3, 4, 5, 6))
```

```
[1] 1.870829
```

```
sum(c(1, 2, 3, 4, 5, 6))
```

```
[1] 21
```

```
min(c(1, 2, 3, 4, 5, 6))
```

```
[1] 1
```

```r
range(c(1, 2, 3, 4, 5, 6))
```

```
[1] 1 6
```

```r
scale(c(1, 2, 3, 4, 5, 6), center = TRUE, scale = FALSE)
```

```
     [,1]
[1,] -2.5
[2,] -1.5
[3,] -0.5
[4,]  0.5
[5,]  1.5
[6,]  2.5
attr(,"scaled:center")
[1] 3.5
```

```r
# Der Mittelwert des zentrierten Vektors wird zur Information mit ausgegeben!
```

```r
scale(c(1, 2, 3, 4, 5, 6), center = TRUE, scale = TRUE)
```

```
           [,1]
[1,] -1.3363062
[2,] -0.8017837
[3,] -0.2672612
[4,]  0.2672612
[5,]  0.8017837
[6,]  1.3363062
attr(,"scaled:center")
[1] 3.5
attr(,"scaled:scale")
[1] 1.870829
```

```r
# Mittelwert und Standardabweichung des z-standardisierten Vektors
# werden zur Information mit ausgegeben!
```

```r
# Ziehen mit Zurücklegen
sample(c(1, 2, 3, 4, 5, 6), size = 1, replace = TRUE)
```

```
[1] 2
```

```r
sample(c(1, 2, 3, 4, 5, 6), size = 12, replace = TRUE)
```

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

```r
# Ziehen mit Zurücklegen, ungleich gewichtet:
sample(c(1, 2, 3, 4, 5, 6), size = 1, replace = TRUE,
       prob = c(4/12, 1/12, 1/12, 2/12, 2/12, 2/12 ))
```

```
[1] 5
```

```r
sample(c(1, 2, 3, 4, 5, 6), size = 12, replace = TRUE,
       prob = c(4/12, 1/12, 1/12, 2/12, 2/12, 2/12 ))
```

```
 [1] 1 1 1 6 4 6 1 1 1 1 1 1
```

```r
c(1, 2, 3, 4, 5, 6)
```

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

```r
seq(from = 1, to = 6, by = 1)
```

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

```r
1:6
```

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

```r
rep(1:6, times = 2)
```

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

```r
rep(1:6, each = 2)
```

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

```r
rep(1:6, times = 2, each = 2)
```

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

Übung

1) Erzeugen Sie eine Sequenz von 0 bis 100 in 5-er Schritten.
2) Berechnen Sie den Mittelwert des Vektors [1, 3, 4, 7, 11, 2].
3) Lassen Sie sich die Spannweite $x_{max} - x_{min}$ dieses Vektors ausgeben.
4) Berechnen Sie die Summe dieses Vektors.
5) Zentrieren Sie diesen Vektor.
6) Simulieren Sie einen Münzwurf mit der Funktion `sample()`. Tipp: nehmen Sie für Kopf 1 und für Zahl 0. Simulieren Sie 100 Münzwürfe.
7) Simulieren Sie eine Trick-Münze mit $p \neq 0.5$
8) Generieren Sie einen Vektor, der aus 100 Wiederholungen der Zahl 3 besteht.

Lösung

```r
# 1)
seq(0, 100, by = 5)
```

```
 [1]   0   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90
[20]  95 100
```

```r
# 2)
c(1, 3, 4, 7, 11, 2)
```

```
[1]  1  3  4  7 11  2
```

```r
mean(c(1, 3, 4, 7, 11, 2))
```

```
[1] 4.666667
```

```r
# 3)
max(c(1, 3, 4, 7, 11, 2)) - min(c(1, 3, 4, 7, 11, 2))
```

```
[1] 10
```

```r
# 4)
sum(c(1, 3, 4, 7, 11, 2))
```

```
[1] 28
```

```
#  5) Zentrieren
scale(c(1, 3, 4, 7, 11, 2), scale = FALSE)
```

```
           [,1]
[1,] -3.6666667
[2,] -1.6666667
[3,] -0.6666667
[4,]  2.3333333
[5,]  6.3333333
[6,] -2.6666667
attr(,"scaled:center")
[1] 4.666667
```

```
# 6)
sample(c(1, 0), size = 1, replace = TRUE)
```

```
[1] 1
```

```
sample(c(1, 0), size = 100, replace = TRUE)
```

```
  [1] 0 1 1 1 1 0 0 1 1 0 0 1 1 1 1 0 1 1 0 1 1 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 1
 [38] 0 0 1 0 0 1 0 0 1 1 0 1 1 0 1 1 1 0 1 1 1 1 1 1 0 0 1 0 1 1 1 0 1 0 1 1 1
 [75] 0 1 0 0 1 1 1 1 0 0 0 1 1 1 0 0 0 1 0 0 0 0 1 1 1 0
```

```
# 7) Trick coin
sample(c(1, 0), size = 100, replace = TRUE, prob = c(5/6, 1/6))
```

```
  [1] 1 1 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1
 [38] 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 1 1 1 0 1
 [75] 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1
```

```
# 8)
rep(3, times = 100)
```

```
  [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
 [38] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
 [75] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

2.2 Variable definieren

Bisher haben wir die Resultate unserer Berechnungen nicht gespeichert. Selbstverständlich können wir in `R` Variablen definieren, und diesen einen oder mehrere Werte zuweisen. Allgemeiner sprechen wir von der Zuweisung bzw. Definition von "Objekten".

Variablen bzw. Objekte werden in R so definiert: `my_var <- 4`. `<-` ist hier ein spezieller Zuweisungspfeil, und besteht aus einem `<` Zeichen und einem `-`. Es gibt in RStudio dafür eine Tastenkombination `ALT` + `-` (auf dem Mac `option` + `-`). Hier weisen wir also der neuen Variablen `my_var` den Wert `4` zu.

> Vertiefung
>
> Man kann in `R` sowohl `<-` als auch `=` als Zuweisungsoperator verwenden. Auf den ersten Blick erscheint das verwirrend (es ist aus historischen Gründen so). Da `=` ausserdem das Symbol für die Zuweisung von Argumenten (in Funktionen) ist, empfehlen wir für die Zuweisung von Variablen/Objekten den Zuweisungspfeil `<-`: dann ist klar, dass man eine Variable definiert. Ausserdem haben `R` Puristen `<-` lieber.

2.2.1 Variablennamen

Eine Variable muss also einen Namen haben - dieser besteht aus `Buchstaben`, `Zahlen` und den Zeichen `_` und/oder `.` Ein Variablenname muss mit einem Buchstaben beginnen, und darf keine Leerschläge enthalten.

Es gibt ein paar Konventionen, an die man sich halten sollte, um `R` Code lesbar und verständlich zu machen - vor allem, wenn man diesen mit anderen teilt. Wir empfehlen, für die Namensgebung `snake_case` zu verwenden, d.h. wir trennen Wörter innerhalb eines Namens mit einem Unterstrich: `my_var`.

Hier zusammen mit weiteren Möglichkeiten:

```
snake_case_variable
camelCaseVariable
variable.with.periods
variable.With_noConventions
```

```
# gute Variablennamen
x_mean
x_sd

anzahl_personen
alter
```

```
# weniger gute Variablennamen
p
a

# unmögliche Variablennamen
x mittelwert
sd von x
```

In vielen Texten, vor allem in älteren, findet man Variablennamen mit `.` anstelle von `_`, moderne Style Guides empfehlen aber `_`.

Wenn wir eine Variable definiert haben:

```
my_var <- 4
```

können wir uns deren Wert in der Konsole ausgeben lassen:

```
print(my_var)
```

```
[1] 4
```

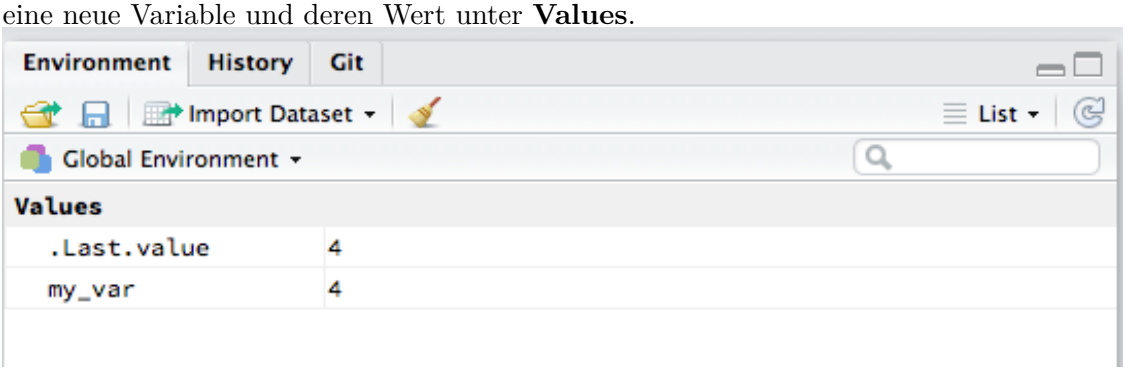
```
# oder einfach

my_var
```

```
[1] 4
```

> Vertiefung
>
> Man kann auch gleich bei der Definition der Variablen die Zuweisung in Klammern schreiben, und das Resultat wird gleichzeitig in der Konsole angezeigt:
>
> ```
> (my_var <- 4)
> ```
>
> ```
> [1] 4
> ```
>
> Das ist manchmal praktisch, im Allgemeinen definiert man aber zunächst eine Variable/ein Objekt und lässt es sich anschliessend ausgeben (oder auch nicht).

Hinweis

Wenn Sie die Variable `my_var` definieren, sehen Sie in RStudio im **Environment**-Bereich eine neue Variable und deren Wert unter **Values**.



Übung

Wiederholen Sie einige der Beispiele von oben, aber speichern Sie diesmal die Resultate in neuen Variablen ab. Sie können die Variablen dann für weitere Operationen wieder verwenden.

```
vektor <- c(1, 3, 4, 7, 11, 2)

mittelwert <- mean(vektor)
mittelwert
```

```
[1] 4.666667
```

```
gerundeter_mittelwert <- round(mittelwert, digits = 1)
gerundeter_mittelwert
```

```
[1] 4.7
```

Diese Variablen existieren nun im **Global Environment**, aber nur solange die aktuelle `R` Session bestehen bleibt (die Zuweisung/Erstellung einer Variablen *speichert* diese also nicht im Wortsinne). Wenn sie `R` neu starten, sind diese Variablen nicht mehr vorhanden. Deshalb sollte man immer in einem `R` Script oder `R` Notebook festhalten, was man gemacht hat.

Hinweis

Dass die Variablen nicht mehr vorhanden sind, wenn sie `R` neu starten, gilt zumindest dann, wenn Sie unserer Empfehlung in Kapitel 1 folgen und die Optionen `Restore`

`.RData into workspace at startup` und `Save workspace to .RData on exit` abwählen. Wenn Sie das nicht tun, wird der gesamte Workspace mit allen Objekten im Global Environment nach dem Neustart wieder geladen (wurden also am Ende der vorherigen Sitzung als Teil des Workspaces in einer Datei abgespeichert). Das führt mit der Zeit aber oft zu einem grossen Chaos von Objekten und Variablen und damit zu Problemen und Fehlern. Daher empfehlen wir, diese Optionen so zu setzen, dass man eine neue `R` Session immer mit einem leeren Global Environment beginnt und sich früher erstellte Variablen etc. durch die Befehle im abgespeicherten `R` Script File jedesmal wieder neu erstellt. Dieses Vorgehen muss aber nicht dazu führen, dass man sehr rechenintensive Schritte jedesmal wiederholen muss, da man auch einzelne Objekte wie z.B. grosse bearbeitete Datensätze separat abspeichern und dann wieder laden kann (was dann auch im `R` Script File festgehalten wird). Darauf werden wir später nochmals zurückkommen.

2.3 Funktionen aufrufen

Wir haben nun schon einige Funktionen verwendet. Hier schauen wir uns nun die Syntax eines Funktionsaufrufs (function call) an.

Die Funktion, welche unten angezeigt wird, besteht aus einem Namen `function_name` und hat zwei Argumente, `arg1` und `arg2`. Die Argumente können, aber müssen nicht, default (voreingestellte) Werte haben. In diesem Beispiel hat `arg1` keinen default Wert. `arg2` hat den default Wert `val2`. Argumente *ohne* default Werte müssen beim Aufruf einer Funktion zwingend angegeben werden. Wenn man Argumente *mit* default Werten nicht explizit angibt, wird automatisch der jeweilige default Wert für das Argument verwendet. Typische default Werte für Argumente sind `TRUE` oder `FALSE`, z.B. gibt es wie oben bereits beschrieben in vielen Funktionen das Argument `na.rm` mit der Voreinstellung (default Wert) `na.rm = FALSE`. Wenn dieses Argument also NICHT explizit angegeben wird, werden fehlende Werte *nicht* entfernt, bevor die jeweilige Funktion angewendet wird und die Funktion gibt im Fall vorhandener fehlender Werte dann als Ergebnis selbst `NA` aus.

```
function_name(arg1, arg2 = val2)
```

Eine Funktion kann beliebig viele Argumente haben.

Hinweis

Wir verwenden in `R` den `=` Operator, wenn wir einem Argument einen Wert zuweisen. Der Unterschied zwischen `<-` und `=` ist am Anfang nicht ganz leicht nachvollziehbar, und wird hier noch einmal illustriert.

```r
# Zuweisungspfeil:
# `<-` weist einem Objekt, z.B. einer Variablen, einen (oder mehrere) Wert(e) zu:
x <- c(23.192, 21.454, 24.677)

# Gleichheitszeichen:
# `=` weist bei einem Funktionsaufruf den Argumenten der Funktion einen Wert zu.
# Z.B. die round Funktion:
round(x, digits = 1)
```

```
[1] 23.2 21.5 24.7
```

Das Argument `digits` bestimmt, auf wieviele Nachkommastellen gerundet werden soll, und wird innerhalb der Funktion `round()` verwendet. Mit `=` weisen wir diesem Argument den Wert `1` zu.

Wenn man wissen will, welche Argumente eine Funktion hat, kann man dies am besten mit Tab-Completion herausfinden.

Übung

Tippen Sie in der Konsole `scale(` und drücken Sie `TAB`. Sie sehen, dass diese Funktion drei Argumente hat, `x`, `center` und `scale`. Schreiben Sie `scale(vektor, scale =` gefolgt von `TAB`. Sie sehen, dass das Argument `scale` (der Funktion `scale()`) den default Wert `TRUE` hat.
Was sind die Argumente der Funktion `round()`? Hat eines der Argumente einen default Wert?
Schauen Sie im **Help** Viewer nach, was die Funktion `rnorm()` macht. Was sind die Argumente? Was bedeuten die default Werte?
Schauen Sie im **Help** Viewer nach, welche Argumente die `seq()` Funktion hat.
Was machen folgende Funktionsaufrufe?

```r
seq()
seq(1, 10)
seq(1, 10, 2)
seq(1, 10, 2, 20)
seq(1, 10, length.out = 20)
```

Verschachtelung von Funktionen

Wir können beliebig viele Funktion ineinander verschachteln, d.h. wir können den Output einer Funktion einer weiteren Funktion als Input übergeben. Wir sehen uns diese Verschachtelung

an folgendem Beispiel an.

Wir definieren zuerst einen Vektor, berechnen davon den Mittelwert und runden diesen auf zwei Dezimalstellen:

```
# definiert einen Vektor:
c(34.444, 45.853, 21.912, 29.261, 31.558)
```

```
[1] 34.444 45.853 21.912 29.261 31.558
```

```
# berechnet den Mittelwert:
mean(c(34.444, 45.853, 21.912, 29.261, 31.558))
```

```
[1] 32.6056
```

```
# rundet auf 2 Dezimalstellen:
round(mean(c(34.444, 45.853, 21.912, 29.261, 31.558)),
      digits = 2)
```

```
[1] 32.61
```

Die Funktionen werden immer in der Reihenfolge *von innen nach aussen* ausgeführt, d.h. zuerst `mean()`, und dann `round()`.

Wenn wir mehrere Funktionen ineinander verschachteln, kann unser Code schnell unlesbar werden. Natürlich könnten wir die einzelnen Zwischenschritte speichern, wie im Beispiel weiter oben, aber dann definieren wir eine Menge Variablen, welche wir vielleicht gar nicht benötigen. Wir werden im Kapitel über Datentransformation einen neuen Operator kennenlernen, der eine elegante Lösung für dieses Problem bietet.

2.4 Datentypen

Wir haben bisher mit Vektoren gearbeitet. Diese stellen den fundamentalen Datentyp dar. Alle weiteren Datentypen bauen auf diesem auf. Vektoren selber können in folgende Typen unterschieden werden in:

- **numeric vectors**: Mit diesen hatten wir es bisher zu tun. Numerische Vektoren werden weiter in `integer` (ganze Zahlen) und `double` (reelle Zahlen) unterteilt.
- **character vectors**: Die Elemente dieses Typs bestehen aus Zeichen, welche von Anführungszeichen umgeben werden (entweder `'` oder `"`), z.B. `'Wort'` oder `"Wort"`.

- **logical vectors**: Die Elemente dieses Typs können nur 3 Werte annehmen: `TRUE`, `FALSE` oder `NA`.

Vektoren müssen aus denselben Elementen bestehen, d.h. wir können nicht `logical` und `character` Vektoren mischen. Vektoren haben 3 Eigenschaften:

- Typ: `typeof()`: Was ist es?
- Länge: `length()`: Wie viele Elemente?
- Attribute: `attributes()`: Zusätzliche Information (Metadaten)

Vektoren werden entweder mit der `c()` (Abkürzung für `combine`) Funktion erstellt, oder mit speziellen Funktion, wie z.B. `seq()` oder `rep()`. Wir schauen uns nun die verschiedenen Typen von Vektoren an.

2.4.1 Numeric vectors

Numerische Vektoren bestehen aus Zahlen. Und zwar entweder aus natürlichen Zahlen (`integer`) oder aus reellen Zahlen (`double`).

```
numbers <- c(1, 2.5, 4.5)
typeof(numbers)
```

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

```
length(numbers)
```

```
[1] 3
```

Wir können die einzelnen Elemente eines Vektor mit `[]` auswählen (dies nennt man im Fachjargon `subsetting`):

```
# Das erste Element:
numbers[1]
```

```
[1] 1
```

```
# Das zweite Element:
numbers[2]
```

```
[1] 2.5
```

```r
# Das letzte Element:
# numbers hat die Länge 3
length(numbers)
```

```
[1] 3
```

```r
# Wir können damit numbers indizieren
numbers[length(numbers)]
```

```
[1] 4.5
```

```r
# Mit - (Minus) können wir ein Element weglassen, z.B. alle Elemente ausser das Erste:
numbers[-1]
```

```
[1] 2.5 4.5
```

```r
# Wir können eine Sequenz verwenden, z.B. die ersten zwei Elemente:
numbers[1:2]
```

```
[1] 1.0 2.5
```

```r
# Wir können das erste und dritte Element weglassen:
numbers[-c(1, 3)]
```

```
[1] 2.5
```

Matrizen

Wie bereits erwähnt, ist eigentlich alles in `R` ein Vektor. Ein Skalar ist ein Vektor der Länge 1. Eine Matrix ist im Prinzip auch ein Vektor, aber einer mit einem `dim` (dimension) Attribut:

```r
# x ist ein Vektor
x <- 1:8

# Wir weisen nun x ein dim Attribut zu, d.h. wir machen aus
# x eine Matrix (hier mit 2 Zeilen und 4 Spalten)
dim(x) <- c(2, 4)
x
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

```
# Wir können Matrizen auch so erstellen:
m <- matrix(1:8, nrow = 2, ncol = 4, byrow = FALSE)
m
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

```
# Was sind die Dimensionen?
dim(m)
```

```
[1] 2 4
```

Beachten Sie das Argument `byrow`, welches den default Wert `FALSE` hat. Wenn wir das Argument `byrow = TRUE` setzen, dann erhalten wir:

```
m2 <- matrix(1:8, nrow = 2, ncol = 4, byrow = TRUE)
m2
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

Dies führt dazu, dass die Zeilen zuerst aufgefüllt werden, während oben `byrow = FALSE` die Spalten zuerst aufgefüllt werden.

Wir können Matrizen transponieren, d.h. die Zeilen werden zu Spalten, und die Spalten werden zu Zeilen:

```
m_transponiert <- t(m)
m_transponiert
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
```

Es gibt zwei weitere Funktionen, welche wir kennenlernen sollten: `cbind()` und `rbind()`. Diese dienen dazu, Vektoren oder Matrizen zusammenzufügen.

`cbind()` kombiniert die Spalten (column-bind) von Vektoren/Matrizen zu einem Objekt:

```
x1 <- 1:3
# x1 ist ein Vektor
x1
```

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

```
x2 <- 10:12
# x2 ist ein Vektor
x2
```

```
[1] 10 11 12
```

```
m1 <- cbind(x1, x2)
# m1 ist eine Matrix mit den Dimensionen [3, 2]
m1
```

```
     x1 x2
[1,]  1 10
[2,]  2 11
[3,]  3 12
```

Daraus entsteht eine Matrix `m1` mit den Ausgangsvektoren als Spalten.

`rbind()` kombiniert die Zeilen (row-bind) von Vektoren/Matrizen zu einem Objekt:

```
m2 <- rbind(x1, x2)
# m2 ist eine Matrix mit den Dimensionen [2, 3]
m2
```

```
   [,1] [,2] [,3]
x1    1    2    3
x2   10   11   12
```

Hier resultiert eine Matrix `m2` mit den Ausgangsvektoren als Zeilen.

Auch Matrizen können mit `[]` indiziert werden. Wir müssen hier aber angeben, welche Zeile(n) und Spalte(n) wir erhalten wollen, und zwar mit `[zeilennummer, spaltennummer]`.

Die Zeilennummer und Spaltennummer sind durch ein Komma getrennt. Wenn `zeilennummer` oder `spaltennummer` leer gelassen werden, heisst das: "Wähle alle Zeilen/Spalten aus."

```
# Die erste Zelle der Matrix: Zeile 1, Spalte 1
m1[1, 1]
```

```
x1
 1
```

```
# Zeile 1, Spalte 2
m1[1, 2]
```

```
x2
10
```

```
# Zeilen 2 bis 3, Spalte 1
m1[2:3, 1]
```

```
[1] 2 3
```

```
# Alle Zeilen, Spalte 1
m1[, 1]
```

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

```
# Zeile 2, alle Spalten
m1[2, ]
```

```
x1 x2
 2 11
```

Vektorisierung

Alle Operatoren in `R` sind vektorisiert, d.h. sie operieren elementweise auf Vektoren:

```r
x1 <- 1:10
x1 + 2
```

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

```r
x2 <- 11:20

x1 + x2
```

```
 [1] 12 14 16 18 20 22 24 26 28 30
```

```r
x1 * x2
```

```
 [1]  11  24  39  56  75  96 119 144 171 200
```

Dasselbe gilt für Funktionen:

```r
x1 <- 1:10

log(x1)
```

```
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
```

```r
exp(x1)
```

```
 [1]     2.718282     7.389056    20.085537    54.598150   148.413159
 [6]   403.428793  1096.633158  2980.957987  8103.083928 22026.465795
```

Recycling

Etwas Besonderes in R ist das Vektor-Recycling: dies bedeutet, dass ein kürzerer Vektor wiederholt wird, wenn wir z.B. zwei Vektoren addieren. Dies kann oft nützlich sein, birgt aber auch Gefahren, wenn man sich dessen nicht bewusst ist.

```r
# Der kürzere Vektor wird rezykliert:
1:10 + 1:2
```

```
 [1]  2  4  4  6  6  8  8 10 10 12
```

Was hier genau passiert, kann so dargestellt werden:

```
1  2  3  4  5  6  7  8  9 10
1  2  1  2  1  2  1  2  1  2
```

Der kürzere Vektor `1:2` wird so oft wie nötig wiederholt.

Was passiert, wenn der der längere Vektor nicht ein Vielfaches des kürzeren ist?

```
1:10 + 1:3
```

```
Warning in 1:10 + 1:3: longer object length is not a multiple of shorter object
length
```

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

`R` führt zwar den Befehl aus, gibt uns aber auch eine Warnung.

```
1  2  3  4  5  6  7  8  9 10
1  2  3  1  2  3  1  2  3  1
```

Missing Values

Fehlende Werte werden mit `NA` deklariert.

```
zahlen <- c(12, 13, 15, 11, NA, 10)
zahlen
```

```
[1] 12 13 15 11 NA 10
```

Mit der Funktion `is.na()` kann man testen, ob etwas tatsächlich ein fehlender Wert ist:

```
is.na(zahlen)
```

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

Hinweis

Fehlende Werte sind nicht zu verwechseln mit den Werten `Inf` (infinity) und `NaN` (not a number). Diese entstehen z.B. bei Division durch Null: `1/0` oder `0/0`. Es gibt ausserdem noch den Datentyp `NULL`: diesen benutzen wir, wenn etwas undefiniert bleiben soll.

```
1/0
```

```
[1] Inf
```

```
0/0
```

```
[1] NaN
```

2.4.2 Character vectors

Character vectors werden benutzt, um Textelemente zu speichern und darzustellen. Die einzelnen Elemente eines Character-Vektors werden auch als "Strings" (Zeichenfolgen) bezeichnet. Dabei muss ein Element nicht aus einem einzelnen Wort bestehen, d.h. auch Leerzeichen können Teil eines Strings sein.

```
text <- c("these are", "some strings")
text
```

```
[1] "these are"    "some strings"
```

```
typeof(text)
```

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

```
# text hat 2 Elemente:
length(text)
```

```
[1] 2
```

Wie numerische Vektoren können auch Character-Vektoren indiziert werden, d.h. wir können einzelne Elemente auswählen.

`letters` und `LETTERS` sind sogenannte `built-in constants`. Dies sind Vektoren mit allen Klein- bzw. Grossbuchstaben der englischen Sprache.

```
?letters
```

```
letters[1:3]
```

```
[1] "a" "b" "c"
```

```
letters[10:15]
```

```
[1] "j" "k" "l" "m" "n" "o"
```

```
LETTERS[24:26]
```

```
[1] "X" "Y" "Z"
```

Es gibt eine Funktion, mit der wir character vectors zusammenfügen können:

```
paste(LETTERS[1:3], letters[24:26], sep = "_")
```

```
[1] "A_x" "B_y" "C_z"
```

```
# Spezialfall mit sep = ""
paste0(1:3, letters[5:7])
```

```
[1] "1e" "2f" "3g"
```

```
vorname <- "Ronald Aylmer"
nachname <- "Fisher"
paste("Mein Name ist:", vorname, nachname, sep = " ")
```

```
[1] "Mein Name ist: Ronald Aylmer Fisher"
```

```
zahl <- 7
# zahl ist zwar eine ganze Zahl, wird aber durch paste() zu einem character
paste(zahl, "ist eine Zahl", sep = " ")
```

```
[1] "7 ist eine Zahl"
```

2.4.3 Logical vectors

Logische Vektoren können drei Werte annehmen: `TRUE`, `FALSE` oder `NA`.

```
log_var <- c(TRUE, FALSE, TRUE)
log_var
```

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

Logische Vektoren werden vor allem dazu benutzt, um numerische Vektoren zu indizieren, z.B. um alle positiven Elemente eines Vektors auszuwählen.

```
# x ist ein Vektor von standardnormalverteilten Zufallszahlen (d.h. wir ziehen
# hier eine Zufallsstichprobe der Grösse n = 24 aus einer normalverteilten
# Population mit dem Mittelwert 0 und der Standardabweichung 1). Die Funktion
# `rnorm()`, mit der wir die Zufallsstichprobe standardnormalverteilter Zahlen
# ziehen, werden wir zu einem späteren Zeitpunkt noch ausführlicher behandeln.
set.seed(5434) # macht das Beispiel reproduzierbar

x <- rnorm(24)
x
```

```
 [1]  1.06115528  0.87480990 -0.30032832  1.21965848  0.09860288  1.89862128
 [7] -1.54699798  0.96349219 -0.64968432 -1.09672125 -0.55326456 -0.29394388
[13]  0.58151046 -0.15135071  1.66997280 -0.10726874  0.51633289 -0.64741465
[19]  0.10489022 -0.95484078  0.22940461 -0.54106301 -0.76310004  1.22446844
```

```
# Wir brauchen nun einen logischen Vektor, der für jedes Element angibt,
# ob dieses Element die Bedingung erfüllt (x soll positiv sein):
x > 0
```

```
 [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
[13]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE
```

```
# Nun indizieren wir damit den Vektor x:
x[x > 0]
```

```
 [1] 1.06115528 0.87480990 1.21965848 0.09860288 1.89862128 0.96349219
 [7] 0.58151046 1.66997280 0.51633289 0.10489022 0.22940461 1.22446844
```

```
# Wir könnten den Index auch speichern
index <- x > 0

# und damit x indizieren:
x[index]
```

```
 [1] 1.06115528 0.87480990 1.21965848 0.09860288 1.89862128 0.96349219
 [7] 0.58151046 1.66997280 0.51633289 0.10489022 0.22940461 1.22446844
```

Wir können auch alle Elemente von `x` suchen, welche eine Standardabweichung über dem Mittelwert liegen:

```
x
```

```
 [1]  1.06115528  0.87480990 -0.30032832  1.21965848  0.09860288  1.89862128
 [7] -1.54699798  0.96349219 -0.64968432 -1.09672125 -0.55326456 -0.29394388
[13]  0.58151046 -0.15135071  1.66997280 -0.10726874  0.51633289 -0.64741465
[19]  0.10489022 -0.95484078  0.22940461 -0.54106301 -0.76310004  1.22446844
```

```
x_mean <- mean(x)
x_sd <- sd(x)
x_mean
```

```
[1] 0.1182059
```

```
x_sd
```

```
[1] 0.9156615
```

```
x > (x_mean + x_sd)
```

```
 [1]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
x[x > (x_mean + x_sd)]
```

```
[1] 1.061155 1.219658 1.898621 1.669973 1.224468
```

```
# oder in einem Befehl
x[x > (mean(x) + sd(x))]
```

```
[1] 1.061155 1.219658 1.898621 1.669973 1.224468
```

Vertiefung

Wenn wir wissen wollen, welche Elemente des Vektors die Bedingung erfüllen, können wir die Funktion `which()` benützen.

```
# dies zeigt an, welche Elemente von x die Bedingung erfüllen
which(x > (mean(x) + sd(x)))
```

```
[1]  1  4  6 15 24
```

In diesem Beispiel liegen die Elemente `1, 4, 6, 15, 24` eine Standardabweichung oder mehr über dem Mittelwert. Wir können auch damit den Vektor indizieren; anstatt einen logischen Vektor zu verwenden, sagen wir einfach explizit, welche Elemente wir haben wollen.

```
x[which(x > (mean(x) + sd(x)))]
```

```
[1] 1.061155 1.219658 1.898621 1.669973 1.224468
```

```
# x[which(x > 0)] und x[x > 0] sind äquivalent
x[which(x > (mean(x) + sd(x)))] == x[x > (mean(x) + sd(x))]
```

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

2.4.4 Factors

Bisher haben wir `numeric`, `logical` und `character` Vektoren kennengelernt. Diese werden in `R` auch `atomic vectors` genannt, weil sie die fundamentalen Datentypen darstellen. Ein weiterer Objekttyp wird benötigt, um kategoriale Daten oder Gruppierungsvariablen darzustellen. Dieser Objekttyp wird `factor` genannt. Ein `factor` ist ein Vektor von natürlichen Zahlen (integer vector), der mit zusätzlicher Information (Metadaten) versehen ist. Diese `attributes` sind die Objektklasse `factor` und die Faktorstufen `levels`. Am besten wir illustrieren dies an einem Beispiel.

```r
# gender ist ein character vector
gender <- c("male", "female", "male", "nonbinary", "male", "female")
gender
```

```
[1] "male"      "female"    "male"      "nonbinary" "male"      "female"
```

```r
typeof(gender)
```

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

```r
attributes(gender)
```

```
NULL
```

Nun können wir mit der Funktion `factor()` einen Faktor definieren:

```r
gender <- factor(gender, levels = c("female", "male", "nonbinary"))
gender
```

```
[1] male      female    male      nonbinary male      female
Levels: female male nonbinary
```

```r
# gender hat nun den Datentyp integer
typeof(gender)
```

```
[1] "integer"
```

```r
# aber die `class` factor
class(gender)
```

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

```r
# und die Attribute levels und class
attributes(gender)
```

```
$levels
[1] "female"    "male"      "nonbinary"

$class
[1] "factor"
```

Wir haben bei der Definition die `levels` explizit angegeben. Dies hätten wir nicht unbedingt tun müssen, da `R` automatisch die vorhandenen Strings in Faktorstufen konvertiert. Standardmässig werden die Faktorstufen alphabetisch geordnet, d.h. in der internen Definition der Faktorvariablen wird in diesem Fall `female` automatisch zur ersten Faktorstufe, `male` zur zweiten und `nonbinary` zur dritten Faktorstufe. Diese interne Definition der Faktorstufenreihenfolge ist unabhängig von der Reihenfolge des Auftretens bestimmter Ausprägungen/Faktorstufen in den Daten.

> Vertiefung
>
> Dass ein `factor` ein `integer vector` ist, erkennen wir, wenn wir das `class` Attribut entfernen, mit `unclass(gender)`. Dies wird auch im **Environment**-Bereich von RStudio angezeigt.
>
> ```
> gender
> ```
>
> ```
> [1] male      female    male      nonbinary male      female
> Levels: female male nonbinary
> ```
>
> ```
> unclass(gender)
> ```
>
> ```
> [1] 2 1 2 3 2 1
> attr(,"levels")
> [1] "female"    "male"      "nonbinary"
> ```

Faktoren werden wir später oft benötigen, wenn wir Regressionsmodelle mit Codiervariablen erstellen möchten. Die **erste Stufe eines Faktors** wird von `R` für solche Modelle automatisch als "Referenzkategorie" bestimmt. Manchmal wollen wir jedoch eine andere Faktorstufe als Referenzkategorie. In diesem Fall kann man die Reihenfolge der Faktorstufen ändern. Es gibt zwei Möglichkeiten: mit `relevel()` oder wie oben bereits gezeigt mit `factor()`.

Mit `relevel()` kann die Referenzkategorie (und nur diese) direkt bestimmt werden. Die anderen Faktorstufen bleiben in ihrer ursprünglichen Reihenfolge.

```
levels(gender)
```

```
[1] "female"    "male"      "nonbinary"
```

```
# Wir müssen das Resultat der Variable wieder zuweisen
gender <- relevel(gender, ref = "male")
levels(gender)
```

```
[1] "male"      "female"    "nonbinary"
```

```
# Nur die Definition der Faktorstufenreihenfolge hat sich geändert hat, die
# Daten selbst sind unverändert
gender
```

```
[1] male      female    male      nonbinary male      female
Levels: male female nonbinary
```

Mit `factor()` kann man die Reihenfolge genau bestimmen - es müssen aber alle Stufen explizit angegeben werden.

```
gender
```

```
[1] male      female    male      nonbinary male      female
Levels: male female nonbinary
```

```
gender <- factor(gender, levels = c("male", "nonbinary", "female"))
gender
```

```
[1] male      female    male      nonbinary male      female
Levels: male nonbinary female
```

Diese Änderungen der Faktorstufenreihenfolge erscheinen womöglich nicht sonderlich relevant, da sich ja an den Daten selbst nichts ändert, d.h. die Reihenfolge des Auftretens bestimmter Ausprägungen (Faktorstufen) in den Daten bleibt gleich, z.B. hat sich die vierte von sechs Personen als nonbinary bezeichnet, unabhängig davon, welche Faktorstufe als erste, zweite und dritte definiert ist. Wie oben schon gesagt, spielt die interne Definition der Faktorstufenreihenfolge später eine wichtige Rolle bei der Datenanalyse, daher ist es hilfreich, sich jetzt schon damit vertraut zu machen.

Vertiefung

Das `levels`-Attribut kann auch direkt angepasst werden, indem man diesem einen Vektor mit Faktorstufen zuweist. Aber Achtung, neben der Faktorstufenreihenfolge werden in diesem Fall auch die Daten angepasst!

```
# Kopie der Variablen erstellen, um später nicht durcheinander zu kommen
gender_veraendert <- gender

levels(gender_veraendert) <- c("nonbinary", "male", "female")

# Vektor mit veränderter Faktorstufendefinition nochmal ausgeben
gender_veraendert
```

```
[1] nonbinary female    nonbinary male      nonbinary female
Levels: nonbinary male female
```

Personen, die vorher die erste Faktorstufe `male` aufwiesen, bekommen jetzt den Wert der neuen ersten Faktorstufe `nonbinary` zugewiesen; Personen, die vorher die zweite Faktorstufe `nonbinary` aufwiesen, bekommen jetzt den Wert der neuen zweiten Faktorstufe `male` zugewiesen; nur Personen mit Geschlecht `female` bekommen keinen anderen Wert zugewiesen, da es sich bei `female` um die neue *und* alte dritte Faktorstufe handelt.
Wegen des Verwirrungspotentials bei der Verwendung dieser Direktzuweisung von `levels` sollte diese Variante nur mit grösster Vorsicht verwendet werden (wir benötigen sie in diesem Kurs nicht).

Eine weitere nützliche Funktion für Faktoren ist `table()`. Damit können wir eine Häufigkeitstabelle erstellen. Wie oft kommt jede Faktorstufe in den Daten vor?

```
table(gender)
```

```
gender
     male nonbinary    female
        3         1         2
```

2.4.5 Lists

Ein weiterer Datentyp ist `list`. Während Vektoren aus Elementen desselben Typs bestehen, können Listen aus heterogenen Elementen zusammengesetzt werden.

Listen werden mit der Funktion `list()` definiert:

```r
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
x
```

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

[[2]]
[1] "a"

[[3]]
[1]  TRUE FALSE  TRUE

[[4]]
[1] 2.3 5.9
```

Hier haben wir eine Liste `x` definiert, welche als Elemente einen `numeric` Vektor, einen `character`, einen `logical` Vektor und einen weiteren `numeric` Vektor enthält.

Listen können wie Vektoren indiziert werden:

```r
x[1]
```

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

```r
x[2]
```

```
[[1]]
[1] "a"
```

```r
x[3]
```

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

```r
x[4]
```

```
[[1]]
[1] 2.3 5.9
```

Vertiefung

Listen können auch mit `[[` indiziert werden. Damit werden die Elemente noch weiter "entpackt". Dies wird sehr gut in *R for Data Science* (Wickham, Çetinkaya-Rundel, et al., 2023) erklärt.

Wir werden in dieser Vorlesung selten selber Listen erstellen. Listen sind aber äusserst wichtig in `R`, und die statistischen Funktionen haben üblicherweise als Output eine Liste. Diese sind aber **`named lists`**, was bedeutet, dass die Elemente der Liste einen Namen haben. Erstellt wird eine **`named list`** so:

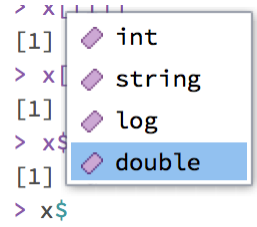
```
x <- list(int = 1:3,
          string = "a",
          log = c(TRUE, FALSE, TRUE),
          double = c(2.3, 5.9))
x
```

```
$int
[1] 1 2 3

$string
[1] "a"

$log
[1]  TRUE FALSE  TRUE

$double
[1] 2.3 5.9
```

```
# Der Typ einer Liste ist "list"
typeof(x)
```

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

Die Elemente von `x` haben nun Namen, und können somit viel einfacher direkt ausgewählt werden. Dafür gibt es den speziellen `$` Operator. Wenn Sie wissen, dass `x` eine Liste ist, Sie aber die Namen der Elemente nicht kennen, können Sie in der Konsole oder im Editor `x$` schreiben, und dann `TAB` drücken. RStudio zeigt alle Elemente dieser Liste an.



Diese Namen müssen nicht in Anführungszeichen geschrieben werden.

```
x$string
```

```
[1] "a"
```

```
x$double
```

```
[1] 2.3 5.9
```

2.4.6 Data Frames

Nun kommen wir zu dem für uns wichtigsten Objekt in `R`, dem Data Frame. Datensätze werden in `R` durch Data Frames repräsentiert. Ein Data Frame besteht aus Zeilen (rows) und Spalten (columns). Technisch gesehen ist ein Data Frame eine Liste, deren Elemente gleich lange (equal-length) Vektoren sind. Die Vektoren selber können `numeric`, `logical` oder `character` Vektoren sein, oder natürlich Faktoren. Numerische Variablen in einem Datensatz sollten demzufolge `numeric` Vektoren und kategoriale Variablen/Gruppierungsvariablen sollten `factors` sein. Ein Data Frame ist eine 2-dimensionale Struktur, und kann einerseits wie ein Vektor indiziert werden (genauer: wie eine Matrix), andererseits wie eine Liste.

Traditionell werden Data Frames in `R` mit der Funktion `data.frame()` definiert. In RStudio bzw. dem `tidyverse`-Package werden Data Frames neuerdings auch `tibbles` oder `tbl` genannt. `tibbles` werden mit der Funktion `tibble()` definiert, und stellen lediglich eine moderne Variante eines Data Frames dar. Sie erleichtern das Arbeiten mit Datensätzen.

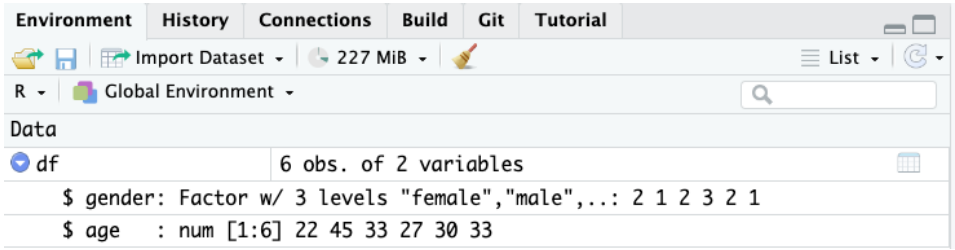
> Vertiefung
>
> Genauer gesagt: wenn man `tibble()` beim Importieren von Datensätzen benützt, muss man weniger vorsichtig sein. RStudio macht das automatisch. Wir verwenden die Begriffe **Data Frame** und **Tibble** als Synonyme, auch wenn sie sich in einigen Merkmalen unterscheiden (**Tibble** hört sich an wie 'table', wenn man es mit einem neuseeländischen Akzent ausspricht).

Ein Data Frame wird so definiert:

```
library(dplyr)
df <- tibble(gender = factor(c("male", "female", "male",
                               "nonbinary", "male", "female")),
             age = c(22, 45, 33, 27, 30, 33))
df
```

```
# A tibble: 6 x 2
  gender      age
  <fct>     <dbl>
1 male         22
2 female       45
3 male         33
4 nonbinary    27
5 male         30
6 female       33
```

`df` ist nun ein Data Frame mit zwei Variablen, `gender` und `age`. Im **Environment**-Bereich von RStudio erscheinen Data Frames unter **Data**:



Ein Data Frame hat die Attribute `names()`, `colnames()` und `rownames()`, wobei `names()` und `colnames()` dasselbe bedeuten.

```
attributes(df)
```

```
$class
[1] "tbl_df"     "tbl"        "data.frame"

$row.names
[1] 1 2 3 4 5 6

$names
[1] "gender" "age"
```

Die Länge eines Data Frames ist die Länge der Liste, d.h. sie entspricht der Anzahl Spalten. Diese kann mit `ncol()` abgefragt werden, wogegen man mit `nrow()` die Anzahl Zeilen des Data Frames erhält.

```
ncol(df)
```

```
[1] 2
```

```
nrow(df)
```

```
[1] 6
```

Data Frame Subsetting

Wie oben erwähnt, kann ein Data Frame wie eine Liste indiziert werden, oder wie eine Matrix.

- Wie eine Liste: die einzelnen Spalten können mit `$` ausgewählt werden.
- Wie eine Matrix: die Elemente können mit `[` ausgewählt werden.

```
# Spaltennamen (Variablen) auswählen
df$gender
```

```
[1] male      female    male      nonbinary male      female
Levels: female male nonbinary
```

```
df$age
```

```
[1] 22 45 33 27 30 33
```

```
df["gender"]
```

```
# A tibble: 6 x 1
  gender
  <fct>
1 male
2 female
3 male
4 nonbinary
5 male
6 female
```

```
df["age"]
```

```
# A tibble: 6 x 1
    age
  <dbl>
1    22
2    45
3    33
4    27
5    30
6    33
```

```
# Nach Position auswählen
df[1]
```

```
# A tibble: 6 x 1
  gender
  <fct>
1 male
2 female
3 male
4 nonbinary
5 male
6 female
```

```
df[2]
```

```
# A tibble: 6 x 1
    age
  <dbl>
1    22
2    45
3    33
4    27
5    30
6    33
```

Ähnlich wie bei Matrizen können Zeilen und Spalten ausgewählt werden, auch hier mit `[zeilennummer, spaltennummer]`.

```r
# Erste Zeile, erste Spalte
df[1, 1]
```

```
# A tibble: 1 x 1
  gender
  <fct>
1 male
```

```r
# Erste Zeile, alle Spalten
df[1, ]
```

```
# A tibble: 1 x 2
  gender   age
  <fct>  <dbl>
1 male      22
```

```r
# ALle Zeilen, erste Spalte
df[, 1]
```

```
# A tibble: 6 x 1
  gender
  <fct>
1 male
2 female
3 male
4 nonbinary
5 male
6 female
```

```r
# Alle Zeilen, alle Spalten
df[ , ]
```

```
# A tibble: 6 x 2
  gender      age
  <fct>     <dbl>
1 male         22
2 female       45
3 male         33
4 nonbinary    27
5 male         30
6 female       33
```

```r
# Wir können auch Sequenzen verwenden
# Die ersten drei Zeilen, alle Spalten
df[1:3, ]
```

```
# A tibble: 3 x 2
  gender   age
  <fct>  <dbl>
1 male      22
2 female    45
3 male      33
```

Da die Spalten Vektoren sind, können wir diese auch indizieren:

```r
df$gender[1]
```

```
[1] male
Levels: female male nonbinary
```

```r
# oder

df$age[2:3]
```

```
[1] 45 33
```

2.5 Übungsaufgaben

Zahlen runden

```r
x <- rnorm(10, mean = 1, sd = 0.5)
x
```

```
 [1]  0.66851021  1.60952322  0.62064523  0.62563620  1.31433575  1.10078969
 [7]  0.48313164 -0.02969281  0.71076782  0.85972382
```

1) Runden Sie den Vektor `x` auf 0 Dezimalstellen.

Lösung

```
round(x = x, digits = 0)
```

```
 [1] 1 2 1 1 1 1 0 0 1 1
```

2) Runden Sie den Vektor `x` auf 3 Dezimalstellen.

Lösung

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

```
 [1]  0.669  1.610  0.621  0.626  1.314  1.101  0.483 -0.030  0.711  0.860
```

3) Runden Sie `zahl` auf die nächste natürliche Zahl auf/ab.

```
(zahl <- 3.45263)
```

```
[1] 3.45263
```

Lösung

```
ceiling(zahl)
```

```
[1] 4
```

```
floor(zahl)
```

```
[1] 3
```

Mittelwert berechnen

1) Berechnen Sie den Mittelwert der Variablen `age` in folgendem Data Frame:

```
df <- tibble(gender = sample(c("male", "female"),
                             size = 24,
                             replace = TRUE),
             age = runif(24, min = 19, max = 45))
df
```

```
# A tibble: 24 x 2
   gender   age
   <chr>  <dbl>
 1 male    22.1
 2 female  22.6
 3 male    24.4
 4 female  40.7
 5 female  22.0
 6 female  42.1
 7 female  22.1
 8 female  41.7
 9 male    38.2
10 male    44.1
# i 14 more rows
```

Lösung

```
mean(df$age)
```

```
[1] 31.17342
```

2) Lassen Sie sich deskriptive Statistiken ausgeben (`summary()`).

Lösung

```
summary(df)
```

```
   gender              age
 Length:24          Min.   :20.27
 Class :character   1st Qu.:24.29
 Mode  :character   Median :30.15
                    Mean   :31.17
                    3rd Qu.:37.94
                    Max.   :44.08
```

Matrizen

1) Kombinieren Sie die beiden Matrizen `m1` und `m2` (beide mit den Dimensionen [12, 4]), so dass eine neue Matrix `m3` entsteht, mit den Dimensionen [24, 4].

```r
m1 <- matrix(rnorm(48, mean = 110, sd = 5), ncol = 4)
m2 <- matrix(rnorm(48, mean = 100, sd = 10), ncol = 4)
m1
```

```
          [,1]     [,2]     [,3]     [,4]
 [1,] 112.6577 110.3748 114.1209 104.0900
 [2,] 116.7024 120.9622 111.6725 113.6405
 [3,] 114.4603 109.6374 107.9318 102.1479
 [4,] 109.8077 109.1152 108.8644 114.3867
 [5,] 113.5108 110.4932 100.2663 112.1465
 [6,] 119.0698 110.9733 113.7455 111.6653
 [7,] 109.4984 114.9592 106.6855 106.4838
 [8,] 110.6606 112.5256 111.0061 114.9715
 [9,] 108.3435 101.4709 107.8447 107.2151
[10,] 108.1350 110.0513 112.4929 101.0572
[11,] 110.8037 113.1775 113.9552 102.4492
[12,] 104.0074 103.4657 110.5090 110.2850
```

```r
m2
```

```
           [,1]      [,2]      [,3]      [,4]
 [1,] 107.06283  97.57565 100.09972 106.05653
 [2,]  94.31888 121.33598  89.61459 120.11837
 [3,]  94.44192 102.61623 113.93752  89.38039
 [4,]  82.87758 108.59534 101.22006  94.78172
 [5,]  97.84665 114.72505 104.53691  97.47717
 [6,] 108.32954  89.14035 107.05284  88.84799
 [7,] 112.39876 120.85322  93.22207 116.04059
 [8,] 111.42864  98.94615 113.08818  99.33227
 [9,] 111.44163  99.61579 102.23889  94.39533
[10,] 110.92366  92.24573  89.61547 115.96388
[11,]  95.51570 114.81185  91.85183 102.81788
[12,] 113.37321  88.37608 109.54577 100.70044
```

Lösung

```r
m3 <- rbind(m1, m2)
m3
```

```
           [,1]      [,2]      [,3]      [,4]
 [1,] 112.65772 110.37480 114.12094 104.09003
```

```
 [2,] 116.70244 120.96217 111.67248 113.64050
 [3,] 114.46026 109.63743 107.93183 102.14786
 [4,] 109.80768 109.11518 108.86436 114.38675
 [5,] 113.51080 110.49324 100.26626 112.14648
 [6,] 119.06984 110.97326 113.74550 111.66534
 [7,] 109.49837 114.95920 106.68548 106.48376
 [8,] 110.66056 112.52559 111.00613 114.97145
 [9,] 108.34351 101.47088 107.84474 107.21505
[10,] 108.13497 110.05134 112.49292 101.05717
[11,] 110.80370 113.17754 113.95516 102.44920
[12,] 104.00743 103.46569 110.50899 110.28505
[13,] 107.06283  97.57565 100.09972 106.05653
[14,]  94.31888 121.33598  89.61459 120.11837
[15,]  94.44192 102.61623 113.93752  89.38039
[16,]  82.87758 108.59534 101.22006  94.78172
[17,]  97.84665 114.72505 104.53691  97.47717
[18,] 108.32954  89.14035 107.05284  88.84799
[19,] 112.39876 120.85322  93.22207 116.04059
[20,] 111.42864  98.94615 113.08818  99.33227
[21,] 111.44163  99.61579 102.23889  94.39533
[22,] 110.92366  92.24573  89.61547 115.96388
[23,]  95.51570 114.81185  91.85183 102.81788
[24,] 113.37321  88.37608 109.54577 100.70044
```

2) Wählen Sie aus `m3` die Elemente so aus (mit Matrix subsetting), dass Sie die ursprüngliche Matrix `m1` erhalten.

Lösung

```
m3[1:12,]
```

```
          [,1]     [,2]     [,3]     [,4]
 [1,] 112.6577 110.3748 114.1209 104.0900
 [2,] 116.7024 120.9622 111.6725 113.6405
 [3,] 114.4603 109.6374 107.9318 102.1479
 [4,] 109.8077 109.1152 108.8644 114.3867
 [5,] 113.5108 110.4932 100.2663 112.1465
 [6,] 119.0698 110.9733 113.7455 111.6653
 [7,] 109.4984 114.9592 106.6855 106.4838
 [8,] 110.6606 112.5256 111.0061 114.9715
 [9,] 108.3435 101.4709 107.8447 107.2151
[10,] 108.1350 110.0513 112.4929 101.0572
```

```
[11,] 110.8037 113.1775 113.9552 102.4492
[12,] 104.0074 103.4657 110.5090 110.2850
```

```
m1
```

```
          [,1]     [,2]     [,3]     [,4]
 [1,] 112.6577 110.3748 114.1209 104.0900
 [2,] 116.7024 120.9622 111.6725 113.6405
 [3,] 114.4603 109.6374 107.9318 102.1479
 [4,] 109.8077 109.1152 108.8644 114.3867
 [5,] 113.5108 110.4932 100.2663 112.1465
 [6,] 119.0698 110.9733 113.7455 111.6653
 [7,] 109.4984 114.9592 106.6855 106.4838
 [8,] 110.6606 112.5256 111.0061 114.9715
 [9,] 108.3435 101.4709 107.8447 107.2151
[10,] 108.1350 110.0513 112.4929 101.0572
[11,] 110.8037 113.1775 113.9552 102.4492
[12,] 104.0074 103.4657 110.5090 110.2850
```

```
m3[1:12,] == m1
```

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

Character vectors

Generieren Sie aus den Variablen `ID`, `Initialen` und `Alter` eine neue Variable, welche so aussieht:

```
"1-RS-44" "2-MM-78" "3-PD-22" "4-PG-34" "5-DK-67" "1-RS-59"
```

Lösung

```
ID <- c(1, 2, 3, 4, 5)
Initialen <- c("RS", "MM", "PD", "PG", "DK")
Alter <- c(44, 78, 22, 34, 67, 59)
```

```
personen <- paste(ID, Initialen, Alter, sep = "-")
personen
```

```
[1] "1-RS-44" "2-MM-78" "3-PD-22" "4-PG-34" "5-DK-67" "1-RS-59"
```

Data Frame

Ändern Sie die Reihenfolge der Faktorstufen des Datensatzes `alk_aggr`, so dass `placebo` die Referenzkategorie bildet.

```
library(dplyr)
library(tidyr)

kein_alkohol <- c(64, 58, 64)
placebo <- c(74, 79, 72)
anti_placebo <- c(71, 69, 67)
alkohol <- c(69, 73, 74)

alk_aggr <- tibble(kein_alkohol = kein_alkohol,
                   placebo = placebo,
                   anti_placebo = anti_placebo,
                   alkohol = alkohol)

alk_aggr <- alk_aggr |>
  pivot_longer(everything(),
               names_to = "alkoholbedingung",
               values_to = "aggressivitaet") |>
  mutate(alkoholbedingung = factor(alkoholbedingung))

alk_aggr
```

```
# A tibble: 12 x 2
```

```
   alkoholbedingung aggressivitaet
   <fct>                      <dbl>
 1 kein_alkohol                  64
 2 placebo                       74
 3 anti_placebo                  71
 4 alkohol                       69
 5 kein_alkohol                  58
 6 placebo                       79
 7 anti_placebo                  69
 8 alkohol                       73
 9 kein_alkohol                  64
10 placebo                       72
11 anti_placebo                  67
12 alkohol                       74
```

Lösung

```
levels(alk_aggr$alkoholbedingung)
```

```
[1] "alkohol"      "anti_placebo" "kein_alkohol" "placebo"
```

```
alk_aggr$alkoholbedingung <- factor(alk_aggr$alkoholbedingung,
                                    levels = c("placebo",
                                               "anti_placebo",
                                               "kein_alkohol",
                                               "alkohol"))
levels(alk_aggr$alkoholbedingung)
```

```
[1] "placebo"      "anti_placebo" "kein_alkohol" "alkohol"
```

```
# alternative Lösung
alk_aggr$alkoholbedingung <- relevel(alk_aggr$alkoholbedingung, ref = "placebo")
levels(alk_aggr$alkoholbedingung)
```

```
[1] "placebo"      "anti_placebo" "kein_alkohol" "alkohol"
```

Fortgeschrittene Aufgabe

1) Wählen Sie aus einem numerischen Vektor nur die geraden Zahlen:

```r
x <- seq(1, 20, by = 1)
x
```

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

Tipp: Sie brauchen dafür den modulo operator `%%`. Geraden Zahlen sind durch 2 teilbar, d.h. es gibt bei der ganzzahligen Teilung keinen Rest.

Lösung

```r
# Gerade Zahlen sind durch 2 teilbar. Wir wollen nun für jedes Element in `x` den Rest, wen
x %% 2
```

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

```r
# Dies gibt uns einen logischen Vektor; wir machen daraus eine Indexvariable:
index <- x %% 2 == 0
index
```

```
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[13] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

```r
# Und nun müssen wir damit `x` indizieren:
gerade_zahlen <- x[index]
gerade_zahlen
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

2) Machen Sie dasselbe für ungerade Zahlen.

```r
x <- seq(1, 20, by = 1)
x
```

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

Lösung

```
# wir wählen alle Zahlen aus, deren Rest ungleich Null ist
index <- x %% 2 != 0

# und indizieren damit x
ungerade_zahlen <- x[index]

ungerade_zahlen
```

```
 [1]  1  3  5  7  9 11 13 15 17 19
```

```
# dasselbe in einer Zeile:
x[x %% 2 != 0]
```

```
 [1]  1  3  5  7  9 11 13 15 17 19
```

3 Datensätze

Wir werden nun anfangen, mit Datensätzen als Data Frames zu arbeiten. Zunächst werden wir diese selber kreieren, und dann Datensätze aus verschiedenen Datenformaten importieren.

3.1 Datensätze selber erstellen

3.1.1 Ohne Messwiederholung

Wir erstellen nun einen Datensatz mit einem `between-subjects` Faktor.

Übung

Erstellen Sie einen Data Frame. Dieser soll zwei Variablen beinhalten:
`Aggressivität`: das Ausmass des aggressiven Verhaltens, gemessen über die Stärke eines elektrischen Schocks, den die Versuchspersonen einer anderen (fiktiven) Person applizieren sollten. Dies könnte die abhängige Variable in einer ANOVA sein.
`Alkoholkonsum`: Vier Experimentalgruppen (Faktorstufen), welche die Alkoholbedingung repräsentieren. Die Stufen sind `kein Alkohol`, `Placebo`, `Anti-Placebo` und `Alkohol`.
Gegeben sind folgende Daten (als Vektoren):

```
kein_alkohol <- c(64, 58, 64)
placebo <- c(74, 79, 72)
anti_placebo <- c(71, 69, 67)
alkohol <- c(69, 73, 74)
```

Das Ziel ist ein solcher Data Frame:

```
alk_aggr
```

```
# A tibble: 12 x 2
   alkoholbedingung alk_aggr
   <fct>               <dbl>
 1 kein_alkohol           64
 2 kein_alkohol           58
```

```
 3 kein_alkohol           64
 4 placebo                74
 5 placebo                79
 6 placebo                72
 7 anti_placebo           71
 8 anti_placebo           69
 9 anti_placebo           67
10 alkohol                69
11 alkohol                73
12 alkohol                74
```

Wie können wir aus vier einzelnen Vektoren einen solchen Data Frame erstellen? Eine mögliche Lösung ist, zunächst aus jedem der vier Vektoren einen Data Frame zu erstellen und die Alkoholbedingung jeweils als weitere Variable hinzuzufügen. Anschliessend können wir die vier Data Frames zu *einem* zusammenfügen, und die Alkoholbedingung zu einem `factor` konvertieren.

Wir wiederholen hier die Erstellung der einzelnen Vektoren, damit es noch klarer wird:

```
# Zuerst dplyr package laden:
library(dplyr)
```

```
kein_alkohol <- c(64, 58, 64)
kein_alkohol <- tibble(
  aggressivitaet = kein_alkohol,
  alkoholbedingung = "kein_alkohol"
)
kein_alkohol
```

```
# A tibble: 3 x 2
  aggressivitaet alkoholbedingung
           <dbl> <chr>
1             64 kein_alkohol
2             58 kein_alkohol
3             64 kein_alkohol
```

```
placebo <- c(74, 79, 72)
placebo <- tibble(
  aggressivitaet = placebo,
  alkoholbedingung = "placebo"
)
placebo
```

```
# A tibble: 3 x 2
  aggressivitaet alkoholbedingung
           <dbl> <chr>
1             74 placebo
2             79 placebo
3             72 placebo
```

```
anti_placebo <- c(71, 69, 67)
anti_placebo <- tibble(
  aggressivitaet = anti_placebo,
  alkoholbedingung = "anti_placebo"
)
anti_placebo
```

```
# A tibble: 3 x 2
  aggressivitaet alkoholbedingung
           <dbl> <chr>
1             71 anti_placebo
2             69 anti_placebo
3             67 anti_placebo
```

```
alkohol <- c(69, 73, 74)
alkohol <- tibble(
  aggressivitaet = alkohol,
  alkoholbedingung = "alkohol"
)
alkohol
```

```
# A tibble: 3 x 2
  aggressivitaet alkoholbedingung
           <dbl> <chr>
1             69 alkohol
2             73 alkohol
3             74 alkohol
```

Nun können wir diese vier Data Frames mit `rbind()` oder `bind_rows()` zusammenfügen. Die Funktion `bind_rows()` ist im `dplyr` package, und hat einige Vorteile (in diesem Beispiel spielt es keine Rolle, welche Sie benützen).

```
alk_aggr <- bind_rows(
  kein_alkohol,
  placebo,
  anti_placebo,
  alkohol
)
```

```
alk_aggr
```

```
# A tibble: 12 x 2
   aggressivitaet alkoholbedingung
            <dbl> <chr>
 1             64 kein_alkohol
 2             58 kein_alkohol
 3             64 kein_alkohol
 4             74 placebo
 5             79 placebo
 6             72 placebo
 7             71 anti_placebo
 8             69 anti_placebo
 9             67 anti_placebo
10             69 alkohol
11             73 alkohol
12             74 alkohol
```

Nun müssen wir noch die Variable `alkoholbedingung` zu einem `factor` konvertieren:

```
alk_aggr$alkoholbedingung <- factor(alk_aggr$alkoholbedingung)
```

```
alk_aggr
```

```
# A tibble: 12 x 2
   aggressivitaet alkoholbedingung
            <dbl> <fct>
 1             64 kein_alkohol
 2             58 kein_alkohol
 3             64 kein_alkohol
 4             74 placebo
 5             79 placebo
 6             72 placebo
```

```
 7             71 anti_placebo
 8             69 anti_placebo
 9             67 anti_placebo
10             69 alkohol
11             73 alkohol
12             74 alkohol
```

> Hinweis
>
> Oft wird auch die Funktion `as.factor()` verwendet. Diese konvertiert ein bestehendes Objekt zu einem `factor`, und würde hier zu demselben Ergebnis führen.

Der Datensatz ist nun komplett, wir wollen aber noch die Variablenreihenfolge umkehren:

```
alk_aggr <- alk_aggr[2:1]
alk_aggr
```

```
# A tibble: 12 x 2
   alkoholbedingung aggressivitaet
   <fct>                     <dbl>
 1 kein_alkohol                 64
 2 kein_alkohol                 58
 3 kein_alkohol                 64
 4 placebo                      74
 5 placebo                      79
 6 placebo                      72
 7 anti_placebo                 71
 8 anti_placebo                 69
 9 anti_placebo                 67
10 alkohol                      69
11 alkohol                      73
12 alkohol                      74
```

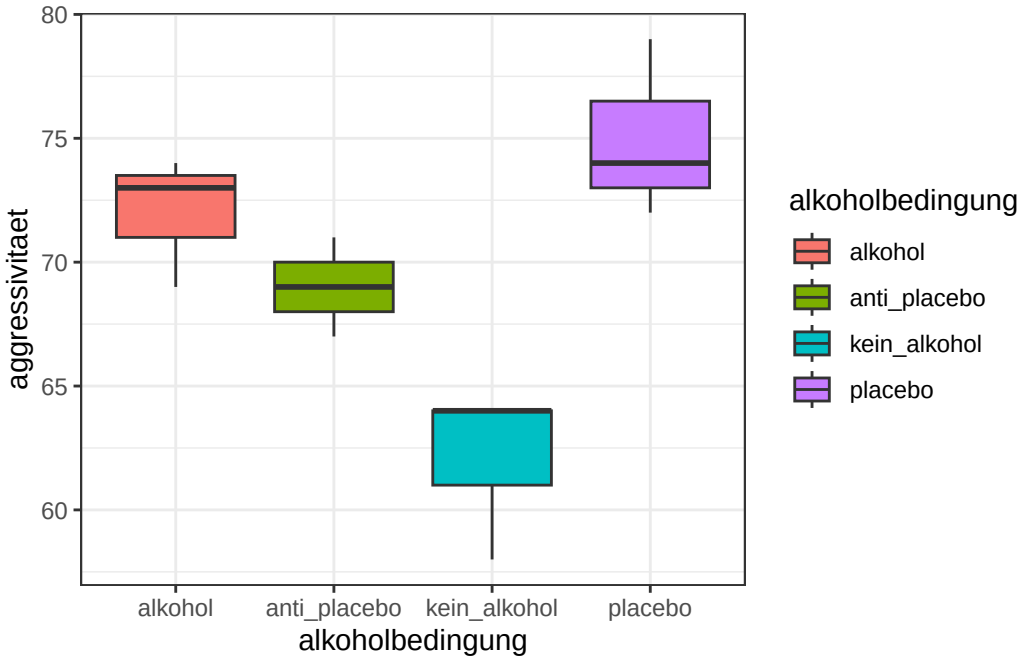
In diesem Datensatz haben wir zwei Variablen - einen Gruppierungsfaktor `alkoholbedingung` und eine Messvariable `aggressivitaet`, und die Stufen des Faktors sind nicht messwiederholt. Jede Beobachtung steht auf einer Zeile, und jede Variable steht in einer Spalte - ein solcher Datensatz ist im *long* Format dargestellt.

> Übung
>
> Welche Faktorstufe ist die Referenzkategorie?

Hinweis

Als kleine Vorschau: wir können diesen Datensatz nun einfach grafisch darstellen, z.B. mit einem Boxplot-Diagramm.

```
library(ggplot2)
alk_aggr |>
  ggplot(aes(
    x = alkoholbedingung,
    y = aggressivitaet,
    fill = alkoholbedingung
  )) +
  geom_boxplot() +
  theme_bw()
```



Die Methode, welche wir hier angewandt haben, um einen Datensatz zu erstellen, ist keine sehr gute Lösung. Wir haben zu viel manuell gemacht, obwohl der Computer eigentlich diese Aufgaben für uns übernehmen sollte. Noch wichtiger: einige Arbeitsschritte haben wir mehrmals ausgeführt, und vielleicht haben wir Copy + Paste benützt. Dies kann gefährlich sein, da sich leicht Fehler einschleichen können.
Wir werden in Kürze eine elegantere Methode kennenlernen, um Data Frames zu erstellen und zu transformieren. Dabei werden wir so vorgehen, dass wir die vier Vektoren spaltenweise zusammenfügen und danach den Data Frame konvertieren (Reshaping).

3.1.2 Mit Messwiederholung

Bei einem Faktor mit Messwiederholung sind wir oft von anderen Statistikprogrammen wie SPSS oder jamovi ein anderes Format gewöhnt. Wenn die Stufen des *repeated measures* Faktors nicht in einer Spalte stehen, sondern in separaten Spalten, ist der Datensatz nicht im *long*, sondern im *wide* Format. Wir veranschaulichen das mit einem weiteren Beispiel.

> Übung
>
> Erstellen Sie einen Data Frame mit den folgenden Variablen:
> `vpn`: Die Versuchspersonennummer.
> `zufriedenheit`: Ein Zufriedenheitsrating auf einer Skala von 0-100 Punkten.
> `messzeitpunkt`: Ein Faktor mit zwei Stufen, welche zwei unterschiedliche Messzeitpunkte repräsentieren.

Wir konstruieren nun diesen Datensatz im *wide* Format.

```
vpn <- c("vp_1", "vp_2", "vp_3", "vp_4")
vpn <- factor(vpn)
# n ist die Anzahl vpn
n <- length(vpn)
```

```
# Daten werden simuliert - Statistiker:innen machen so etwas oft, um etwas zu
# illustrieren oder um statistische Verfahren zu testen.

set.seed(1234)

zufriedenheit_t1 <- round(rnorm(n, mean = 60, sd = 10), digits = 2)
zufriedenheit_t1
```

```
[1] 47.93 62.77 70.84 36.54
```

```
zufriedenheit_t2 <- round(rnorm(n, mean = 75, sd = 10), digits = 2)
zufriedenheit_t2
```

```
[1] 79.29 80.06 69.25 69.53
```

Nun können wir die Variablen mit `tibble()` zu einem Datensatz zusammenfügen:

```r
zufriedenheit <- tibble(
  vpn = vpn,
  zufriedenheit_t1 = zufriedenheit_t1,
  zufriedenheit_t2 = zufriedenheit_t2
)
zufriedenheit
```

```
# A tibble: 4 x 3
  vpn   zufriedenheit_t1 zufriedenheit_t2
  <fct>            <dbl>            <dbl>
1 vp_1              47.9             79.3
2 vp_2              62.8             80.1
3 vp_3              70.8             69.2
4 vp_4              36.5             69.5
```

Mit diesem Datensatz können wir in `R` z.B. einen t-Test durchführen, das *wide* Format ist aber für viele Anwendungen nicht optimal, denn viele statistische Verfahren verlangen einen Datensatz im *long* Format. Um die Daten mit `ggplot2` grafisch darzustellen, ist es ebenfalls sinnvoll, den Datensatz in ein long Format zu konvertieren. Datenkonvertierung wird oft als Reshaping bezeichnet. Wir werden im nächsten Kapitel genauer darauf eingehen, wenn wir die `tidyverse` Packages kennenlernen. Ein Begriff, welchen wir immer häufiger antreffen, ist *tidy* Data. Dies bezieht sich eben auf das Datenformat: das *long* Format wird bevorzugt (ist *tidier* als das *wide* Format). Wenn wir uns daran halten, erleichtern wir uns die Arbeit mit `R` erheblich, und wir können von den `tidyverse` Packages profitieren, welche alle sehr gut aufeinander abgestimmt sind.

Bevor wir das Data Reshaping automatisieren, ist es aber sinnvoll, einmal einen Datensatz manuell von *wide* zu *long* zu konvertieren.

3.1.3 Manuelle Konversion von *wide* zu *long*

Jede Person gibt zu zwei Messzeitpunkten ein Zufriedenheits-Rating - diese Ratings stellen die Beobachtungen dar. In einem *long* Datensatz stehen die **Beobachtungen** in den Zeilen, daher erhält jede Beobachtung eine Zeile, und nicht mehr jede Person (da jede Person jetzt zwei Beobachtungen hat). Die Werte in der Personen/ID-Variable (hier: `vpn`) werden gemäss der Anzahl Beobachtungen pro Person dupliziert.

```r
# Wir nennen den Datensatz nun zufriedenheit_wide
zufriedenheit_wide <- zufriedenheit
```

```r
vpn <- rep(zufriedenheit_wide$vpn, each = 2)
vpn
```

```
[1] vp_1 vp_1 vp_2 vp_2 vp_3 vp_3 vp_4 vp_4
Levels: vp_1 vp_2 vp_3 vp_4
```

Nun müssen wir aus den beiden Variablen `zufriedenheit_t1` und `zufriedenheit_t2` des wide Datensatzes einen Faktor `messzeitpunkt` und eine Variable `zufriedenheit` erstellen:

```r
messzeitpunkt <- rep(c("t1", "t2"), times = n)
messzeitpunkt <- as.factor(messzeitpunkt)

messzeitpunkt
```

```
[1] t1 t2 t1 t2 t1 t2 t1 t2
Levels: t1 t2
```

```r
zufriedenheit <- c(rbind(
  zufriedenheit_wide$zufriedenheit_t1,
  zufriedenheit_wide$zufriedenheit_t2
))

# oder

zufriedenheit <- as.vector(rbind(
  zufriedenheit_wide$zufriedenheit_t1,
  zufriedenheit_wide$zufriedenheit_t2
))

zufriedenheit
```

```
[1] 47.93 79.29 62.77 80.06 70.84 69.25 36.54 69.53
```

> Vertiefung
>
> Dieser Schritt war etwas schwierig nachzuvollziehen. Das macht nichts, Sie müssen das in Zukunft auch nicht selber machen. Wir haben hier lediglich die beiden Spalten `zufriedenheit_t1` und `zufriedenheit_t2` als Zeilenvektoren zu einer Matrix zusammengefügt, und dann mit `c()` oder `as.vector()` zu einem Vektor konvertiert. Dabei

haben wir uns die Tatsache zunutze gemacht, dass `R` Operationen zuerst auf die Spalten einer Matrix angewendet werden *(column-major)*.

```
rbind(
  zufriedenheit_wide$zufriedenheit_t1,
  zufriedenheit_wide$zufriedenheit_t2
)
```

```
      [,1]  [,2]  [,3]  [,4]
[1,] 47.93 62.77 70.84 36.54
[2,] 79.29 80.06 69.25 69.53
```

Wir haben nun alle drei Variablen, die wir für den Datensatz im long Format brauchen, und können diese nun zusammenfügen:

```
zufriedenheit_long <- tibble(
  vpn = vpn,
  messzeitpunkt = messzeitpunkt,
  zufriedenheit = zufriedenheit
)
```

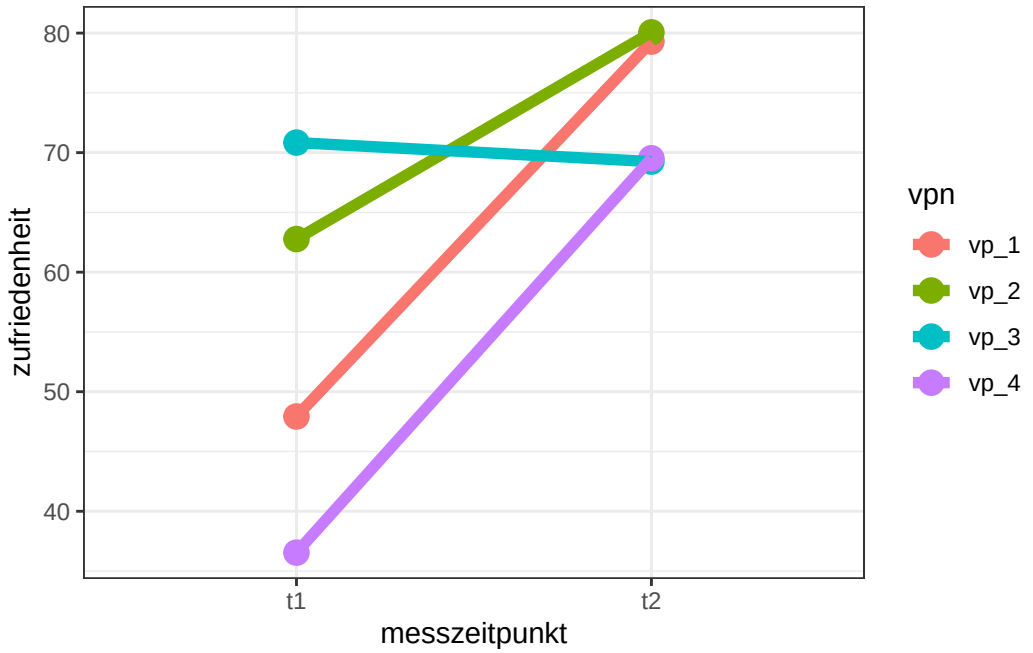
```
zufriedenheit_long
```

```
# A tibble: 8 x 3
  vpn   messzeitpunkt zufriedenheit
  <fct> <fct>                <dbl>
1 vp_1  t1                    47.9
2 vp_1  t2                    79.3
3 vp_2  t1                    62.8
4 vp_2  t2                    80.1
5 vp_3  t1                    70.8
6 vp_3  t2                    69.2
7 vp_4  t1                    36.5
8 vp_4  t2                    69.5
```

Der Datensatz mit Messwiederholung kann jetzt ähnlich wie derjenige ohne Messwiederholung mit `ggplot2` grafisch dargestellt werden.

```
library(ggplot2)
zufriedenheit_long |>
  ggplot(aes(
```

```
    x = messzeitpunkt,
    y = zufriedenheit,
    group = vpn, colour = vpn
  )) +
  geom_point(size = 4) +
  geom_line(size = 2) +
  theme_bw() # macht den Hintergrund weiss
```



Manuelles Reshaping ist eher mühsam, und sollte vermieden werden. Das obige Beispiel soll nur zur Illustration dienen. Wir werden in Zukunft dafür eine Funktion aus dem `tidyr` Package verwenden: `pivot_longer()`.

Als kleine Vorschau: die Konvertierung von *wide* zu *long* würden wir so machen:

```
library(tidyr)
library(stringr)

zufriedenheit_wide
```

```
# A tibble: 4 x 3
  vpn   zufriedenheit_t1 zufriedenheit_t2
  <fct>            <dbl>            <dbl>
1 vp_1              47.9             79.3
```

```
2 vp_2              62.8              80.1
3 vp_3              70.8              69.2
4 vp_4              36.5              69.5
```

```
zufriedenheit_long <- zufriedenheit_wide |>
  pivot_longer(-vpn, names_to = "messzeitpunkt", values_to = "zufriedenheit")

zufriedenheit_long
```

```
# A tibble: 8 x 3
  vpn   messzeitpunkt    zufriedenheit
  <fct> <chr>                    <dbl>
1 vp_1  zufriedenheit_t1          47.9
2 vp_1  zufriedenheit_t2          79.3
3 vp_2  zufriedenheit_t1          62.8
4 vp_2  zufriedenheit_t2          80.1
5 vp_3  zufriedenheit_t1          70.8
6 vp_3  zufriedenheit_t2          69.2
7 vp_4  zufriedenheit_t1          36.5
8 vp_4  zufriedenheit_t2          69.5
```

Jetzt wollen wir noch `zufriedenheit_` von den Faktorstufen entfernen, so dass die Stufen `t1` und `t2` heissen. Dafür verwenden wir die Funktion `str_replace()`.

```
zufriedenheit_long$messzeitpunkt <- zufriedenheit_long$messzeitpunkt |>
  str_replace(".*_", "") |>
  as.factor()

zufriedenheit_long
```

```
# A tibble: 8 x 3
  vpn   messzeitpunkt zufriedenheit
  <fct> <fct>                 <dbl>
1 vp_1  t1                     47.9
2 vp_1  t2                     79.3
3 vp_2  t1                     62.8
4 vp_2  t2                     80.1
5 vp_3  t1                     70.8
6 vp_3  t2                     69.2
7 vp_4  t1                     36.5
8 vp_4  t2                     69.5
```

Die obige Schreibweise benützt den *Pipe* Operator `|>`: wir werden bald mehr darüber erfahren. Äquivalent dazu können wir die Funktionen auch verschachtelt aufrufen:

```
zufriedenheit_long <- pivot_longer(-vpn,
  names_to = "messzeitpunkt",
  values_to = "zufriedenheit"
)

zufriedenheit_long$messzeitpunkt <-
  as.factor(str_replace(
    zufriedenheit_long$messzeitpunkt,
    ".*_", ""
  ))
```

3.1.4 Long vs. wide

Um den Unterschied zwischen einem *long* und einem *wide* Datensatz zu verdeutlichen, zeigen wir hier nochmals beide Varianten:

```
# wide
zufriedenheit_wide
```

```
# A tibble: 4 x 3
  vpn   zufriedenheit_t1 zufriedenheit_t2
  <fct>            <dbl>            <dbl>
1 vp_1              47.9             79.3
2 vp_2              62.8             80.1
3 vp_3              70.8             69.2
4 vp_4              36.5             69.5
```

```
# long
zufriedenheit_long
```

```
# A tibble: 8 x 3
  vpn   messzeitpunkt zufriedenheit
  <fct> <fct>                 <dbl>
1 vp_1  t1                   47.9
2 vp_1  t2                   79.3
3 vp_2  t1                   62.8
4 vp_2  t2                   80.1
```

```
5 vp_3  t1                    70.8
6 vp_3  t2                    69.2
7 vp_4  t1                    36.5
8 vp_4  t2                    69.5
```

Übung

1) Versuchen Sie, aus dem *long* Datensatz den ersten Messzeitpunkt für alle Versuchspersonen zu extrahieren. Sie müssen dazu die Zeilen des Data Frames mit einem logischen Vektor indizieren, und alle Spalten auswählen.

2) Extrahieren Sie die Zufriedenheitsratings zu beiden Messzeitpunkten für die erste Versuchsperson.

Lösung

Aufgabe 1)

```
zufriedenheit_long[zufriedenheit_long$messzeitpunkt == "t1", ]
```

```
# A tibble: 4 x 3
  vpn   messzeitpunkt zufriedenheit
  <fct> <fct>                <dbl>
1 vp_1  t1                    47.9
2 vp_2  t1                    62.8
3 vp_3  t1                    70.8
4 vp_4  t1                    36.5
```

Aufgabe 2)

```
zufriedenheit_long[zufriedenheit_long$vpn == "vp_1", 3]
```

```
# A tibble: 2 x 1
  zufriedenheit
          <dbl>
1          47.9
2          79.3
```

oder

```
zufriedenheit_long[zufriedenheit_long$vpn == "vp_1", "zufriedenheit"]
```

```
# A tibble: 2 x 1
  zufriedenheit
          <dbl>
1          47.9
2          79.3
```

3.2 Daten importieren

Meistens arbeiten wir jedoch nicht mit selber generierten Datensätzen, sondern wollen diese aus Textdateien, Excel-Spreadsheets oder SPSS-Dateien importieren.

Laden Sie bitte folgende Datensätze von GitHub herunter, indem Sie Rechtsklick auf die folgenden Datensätze machen und dann "Ziel speichern unter" wählen:

1) zufriedenheit.csv
2) zufriedenheit-semicolon.csv
3) zufriedenheit.sav
4) zufriedenheit.xlsx

Es handelt sich um den Datensatz zur Zufriedenheit, den wir oben erstellt haben. `zufriedenheit.csv` ist eine Textdatei, in dem die Spalten durch Kommata getrennt sind (csv = comma-separated values). `zufriedenheit-semicolon.csv` ist ebenfalls eine Textdatei, die Spalten sind hier aber durch Strichpunkte (Semikolons) getrennt. `zufriedenheit.sav` und `zufriedenheit.xls` sind SPSS- bzw. Excel-Dateien.

Kreieren Sie in Ihrem Projektordner in RStudio einen neuen Ordner und nennen Sie diesen **data**. Speichern Sie nun die heruntergeladenen Datenfiles in diesem Ordner. Wir werden diese Dateien nun der Reihe nach importieren.

Es gibt in RStudio zwei Möglichkeiten, Datensätze zu importieren:

1) Mit Funktionsaufrufen: `read_csv()` bzw. `read_delim()` (für ';'), `read_sav()` und `read_excel()`.
2) Mit dem GUI: Das Menu kann entweder via 'File > Import Dataset' oder im Environment-Bereich aufgerufen werden.

Die zweite Option ist am Anfang einfacher und hat zwei Vorteile. Erstens werden alle Optionen für den Import angezeigt (diese sind auch als Funktionsargumente verfügbar), und zweitens generiert RStudio den entsprechenden `R` Code. Wenn man sich nicht sicher ist, kann man beim ersten Mal das GUI benützen, und in Zukunft dann den generierten Code. Dies ist oft schneller und hat wiederum den Vorteil, dass man alle Schritte für die Datenanalyse in einem `R` Script/Notebook festhalten kann.

3.2.1 CSV-Dateien

Die Funktion, welche wir benötigen, um CSV-Dateien zu importieren, befindet sich im `readr` Package. Dieses müssen wir zuerst laden:

```
library(readr)
```

Da dieses Package zu dem Metapackage `tidyverse` gehört, können wir einfach dieses laden:

```
library(tidyverse)
```

Zuerst importieren wir den Datensatz via GUI. Klicken Sie in *Environment* auf *Import Dataset > From Text (readr)* (oder '*File > Import Dataset > From Text (readr)*'). Im Dialogfenster sehen Sie rechts unten einen *Code Preview*. Hier steht am Anfang:

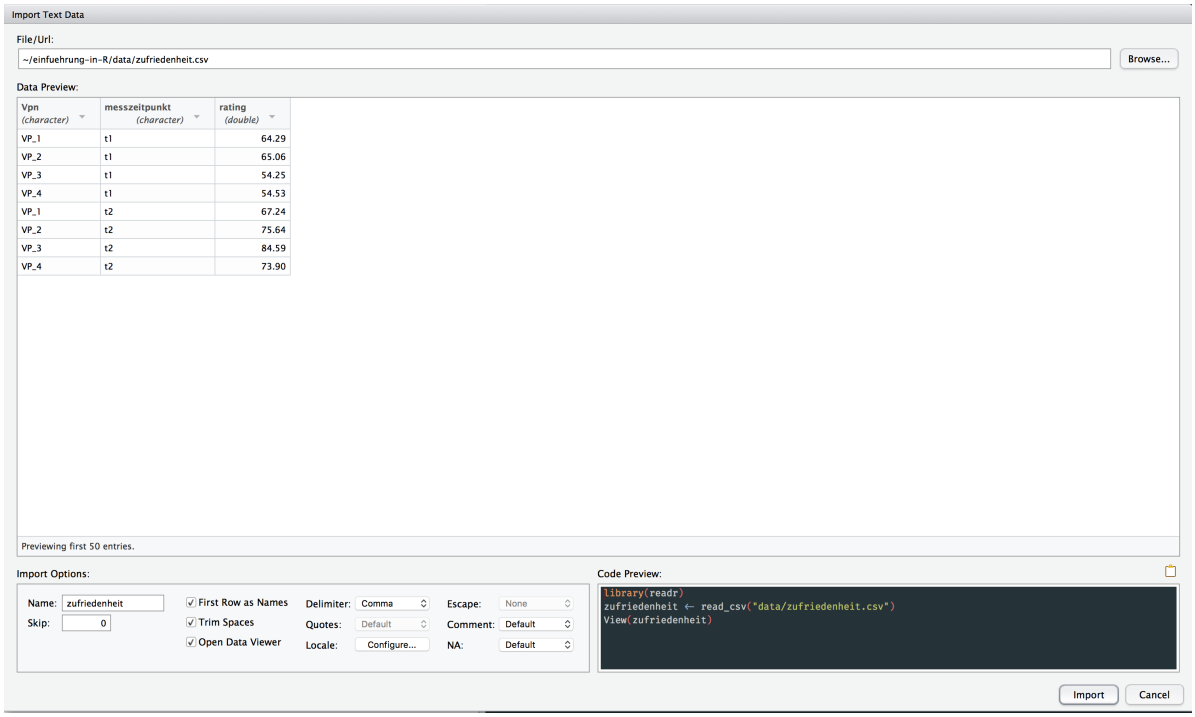
```
library(readr)
dataset <- read_csv(NULL)
View(dataset)
```

Links unten sehen Sie alle Optionen für den Import. Diese Optionen sind Argumente der `read_csv()` oder der allgemeineren `read_delim()` Funktion:

```
args(read_csv)
```

```
function (file, col_names = TRUE, col_types = NULL, col_select = NULL,
    id = NULL, locale = default_locale(), na = c("", "NA"), quoted_na = TRUE,
    quote = "\"", comment = "", trim_ws = TRUE, skip = 0, n_max = Inf,
    guess_max = min(1000, n_max), name_repair = "unique", num_threads = readr_threads(),
    progress = show_progress(), show_col_types = should_show_types(),
    skip_empty_rows = TRUE, lazy = should_read_lazy())
NULL
```

Wählen Sie über *File / Browse* die Datei `zufriedenheit.csv` aus. Sie sehen nun den Pfad dieser Datei sowie eine *Data Preview*. Hier werden die Daten gezeigt, mit Angaben zum Datentyp.



Sie sollten sehen, dass sowohl die Variable `vpn` als auch die Variable `messzeitpunkt` als `character vectors` importiert wurden. Diese werden wir nachher zu Faktoren konvertieren.

Bei *Import Options* wird automatisch ein Name für den Data Frame generiert; dieser entspricht dem Dateinamen (ohne das Suffix '.csv.').

Sie probieren am besten selbst aus, welche Auswirkung es hat, wenn Sie die Optionen ändern. Wenn Sie z.B. "First Row as Names" nicht auswählen, wird `R` die Variablennamen, die in der ersten Zeile der CSV-Datei stehen, nicht verwenden. Eine weitere wichtige Option ist "NA": hier können Sie angeben, welche Werte in der Datei als fehlende Werte behandelt werden sollen.

Nun klicken Sie bitte auf *Import*. Im *Environment*-Bereich erscheint ein Data Frame (`zufriedenheit`), und der Code, der verwendet wurde, erscheint in der Konsole.

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

```
Rows: 8 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): vpn, messzeitpunkt
```

```
dbl (1): zufriedenheit

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

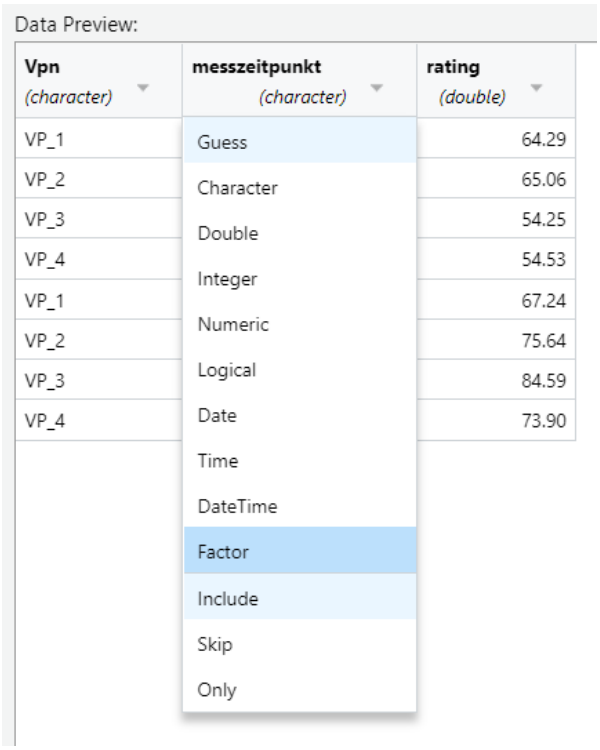
Nun müssen wir die beiden Gruppierungsvariablen zu Faktoren konvertieren:

```
zufriedenheit$vpn <- as.factor(zufriedenheit$vpn)
zufriedenheit$messzeitpunkt <- as.factor(zufriedenheit$messzeitpunkt)
```

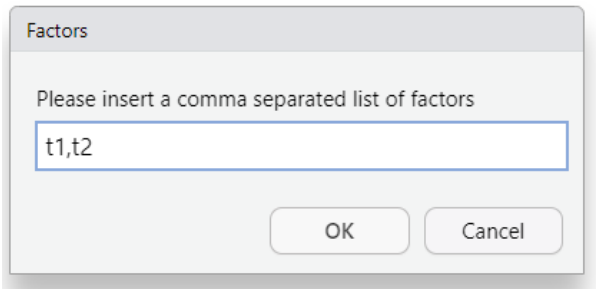
```
zufriedenheit
```

```
# A tibble: 8 x 3
  vpn   messzeitpunkt zufriedenheit
  <fct> <fct>                <dbl>
1 vp_1  t1                    64.3
2 vp_2  t1                    65.1
3 vp_3  t1                    54.2
4 vp_4  t1                    54.5
5 vp_1  t2                    67.2
6 vp_2  t2                    75.6
7 vp_3  t2                    84.6
8 vp_4  t2                    73.9
```

Alternativ können Variablen auch bereits beim Einlesen via GUI formatiert werden. Öffnen sie dazu nochmals via *Import Dataset > From Text (readr) > Browse* den Datensatz `zufriedenheit.csv`. Durch das Klicken auf einen der Spaltennamen im *Data Preview* öffnet sich ein Dropdown:



Dieses Dropdown hat zwei Teile: im ersten Teil (von *Guess* bis *Factor*) können sie wählen, welches Format die Spalte haben soll (standardmässig auf *Guess*. Im zweiten Teil (*Include* bis *Only*) kann gesteuert werden, ob die Spalte eingelesen werden soll oder nicht (standardmässig auf *Include*). Wir wählen hier *Factor*, um die Spalte als Faktor einzulesen. Dies öffnet ein weiteres Fenster. Hier geben wir die gewünschten Faktorstufen ein, jeweils getrennt durch ein Komma:



Im *Code Preview* werden diese Änderungen nun ebenfalls angezeigt. Wenn Sie den Datensatz das nächste Mal einlesen, werden diese Einstellungen also direkt übernommen.

Code Preview:

```
library(readr)
zufriedenheit <- read_csv("data/zufriedenheit.csv",
    col_types = cols(vpn = col_factor(levels = c("vp_1",
        "vp_2", "vp_3", "vp_4")), messzeitpunkt = col_factor(levels = c("t1",
        "t2"))))
View(zufriedenheit)
```

Diese Methode der Faktor-Konvertierung ist am Anfang deutlich einfacher als die manuelle Methode mit `as.factor()` oder `factor()`. Es ist jedoch wichtig, sich auch mit letzteren vertraut zu machen, da auch nach Einlesen eines Datensatzes häufig damit gearbeitet werden muss, z.B. wenn eine (weitere) Faktorvariable aus einer bestehenden Variable erstellt werden soll usw..

Übung

Machen Sie nun dasselbe mit der Datei `zufriedenheit-semicolon.csv` . Hier müssen Sie bei den Optionen für *Delimiter* unbedingt *Semicolon* auswählen, da `R` normalerweise ein Komma als Trennzeichen erwartet.
Sie erhalten nun im *Code Preview* den Code:

```
library(readr)
zufriedenheit_semicolon <- read_delim("data/zufriedenheit-semicolon.csv",
  ";",
  escape_double = FALSE,
  trim_ws = TRUE
)
```

Versuchen Sie, die Dateien ohne GUI, d.h. mit dem generierten `read_delim()`-Code zu importieren.

Wenn wir einen Data Frame als CSV-Datei speichern wollen, verwenden wir dafür die Funktion `write_csv()`:

```
write_csv(x = zufriedenheit, file = "data/zufriedenheit.csv")
```

3.2.2 SPSS-Dateien

Nun importieren wir denselben Datensatz, aber dieses Mal von einer SPSS-Datei (`zufriedenheit.sav`). Dafür benötigen wir die Funktion `read_sav()`. Diese befindet sich im Package `haven`, welches wir zuerst noch installieren müssen:

```
install.packages("haven")
```

Sobald `haven` installiert ist, können Sie auch im *Environment* auf *Import Dataset* > *From SPSS* klicken und die Datei auswählen. Sie sehen dann den *Code Preview*:

```
library(haven)
zufriedenheit_spss <- read_sav("data/zufriedenheit.sav")
```

Im Gegensatz zum Importieren von CSV-Dateien gibt es hier keine Optionen - mit Ausnahme des Namens, den der Data Frame nach dem Import erhält. Diesen ändern wir vom vorgeschlagenen `zufriedenheit` zu `zufriedenheit_spss` (um den vorher importierten Data Frame nicht zu überschreiben). Klicken Sie auf *Import*, der Data Frame `zufriedenheit_spss` erscheint anschliessend im *Global Environment*.

Im Unterschied zu dem von einer CSV-Datei importierten Data Frame haben die Variablen `zufriedenheit_spss` weitere Attribute. Das wichtigste dabei ist das `Labels` Attribut:

```
zufriedenheit_spss$vpn
```

```
<labelled<double>[8]>
[1] 1 2 3 4 1 2 3 4

Labels:
 value label
     1  vp_1
     2  vp_2
     3  vp_3
     4  vp_4
```

Dieses beinhaltet die Wertelabels aus einem SPSS-Datensatz. In SPSS, einem weit verbreiteten kommerziellen Statistikprogramm, werden - anders als in `R` - auch Faktorstufen (also Gruppen- oder Kategorienbezeichnungen) grundsätzlich numerisch kodiert (z.B. für Geschlecht: `0` für `maennlich`, `1` für `weiblich` und `2` für `divers` - oder beliebige andere Zahlen). Die eigentlichen Bezeichnungen der Faktorstufen werden dann zusätzlich als Attribute der Werte hinterlegt (sogenannte Wertelabels).

Die so aus einem SPSS-Datensatz eingelesene Variable ist zunächst ein Hybrid aus einer numerischen und einer Faktor-Variable, eine so genannte *labelled double* Variable. In unserem Beispiel werden die Werte der Versuchspersonen-Variable `vpn` als numerische Werte eingelesen, die aber jeweils ein Label besitzen, das die eigentliche Bezeichnung der Faktorstufe (hier des Personen-Faktors) darstellt.

Um die als *labelled double* eingelesenen Variablen zu Faktoren konvertieren, muss nach dem Einlesen der Daten die Funktion `as_factor()` aus `haven` verwendet werden. Diese Funktion erlaubt es, entweder die numerischen Werte oder die Wertelabels (oder beide zusammengefügt) als Stufen des zu erstellenden Faktors zu definieren. Die Einstellung erfolgt mit dem Argument `levels`. Die für uns sinnvollste Einstellung ist in den meisten Fällen `levels = "default"`. Damit werden die *Wertelabels* als Faktorstufen benützt, sofern solche vorhanden sind, andernfalls wird auf die Werte selbst zurückgegriffen. `"default"` ist zudem auch die Standardeinstellung, sodass beim Weglassen des `levels`-Arguments standardmässig die Wertelabels als Faktorstufen eingelesen werden. Die anderen `levels`-Optionen sind `"both"` (Werte und Wertelabels werden zusammengefügt), `"labels"` (nur Labels, `NA` falls keine vorhanden) und `"values"` (nur Werte).

Argumente der Funktion `as_factor()`

```
levels
How to create the levels of the generated factor:
"default": uses labels where available, otherwise the values. Labels are sorted by value.
"both": like "default", but pastes together the level and value
"labels": use only the labels; unlabelled values become NA
"values": use only the values

ordered
If TRUE create an ordered (ordinal) factor, if FALSE (the default) create a regular (nominal)
```

Mit dem Argument `ordered`, welches entweder `TRUE` oder `FALSE` (Standardeinstellung) ist, kann ein ordinaler Faktor erstellt werden, d.h. ein Faktor, dessen Stufen in einer Rangfolge stehen. In den meisten Fällen wird diese Einstellung nicht benötigt, das Argument kann daher normalerweise weggelassen werden (default: `ordered = FALSE`).

```
zufriedenheit_spss$vpn <- as_factor(zufriedenheit_spss$vpn,
  levels = "default"
)
zufriedenheit_spss$messzeitpunkt <- as_factor(zufriedenheit_spss$messzeitpunkt,
  levels = "default"
)
```

So können `R` Data Frames als SPSS-Datendatei gespeichert werden (Dateiendung: `.sav`):

```
write_sav(data = zufriedenheit_spss, path = "data/zufriedenheit.sav")
```

Übung

Importieren Sie den Datensatz `beispieldaten.sav` (ebenfalls auf GitHub). Versuchen Sie herauszufinden, welche Wertelabels die kategorialen Variablen haben.

3.2.3 Excel-Dateien

Auch Excel-Dateien können importiert werden. Wählen Sie *Import Dataset* > *From Excel*, und wählen Sie dann die heruntergeladene Excel-Datei `zufriedenheit.xlsx` aus. Nennen Sie den Data Frame `zufriedenheit_xl` (um ihn von den bereits importierten Data Frames zu unterscheiden). Unter dem *Name*-Feld sehen sie ein Dropdown-Menu mit dem Namen *Sheet*. Hier können Sie angeben, welches Worksheet der Excel-Datei Sie importieren möchten. Wenn Sie hier `zufriedenheit` auswählen, erscheint folgender `R` Code im *Code Preview*:

```
library(readxl)
zufriedenheit_xl <- read_excel("data/zufriedenheit.xlsx",
  sheet = "zufriedenheit"
)
```

Die benötigte Funktion heisst `read_excel()` und befindet sich im `readxl` package. Dieses muss zuvor explizit geladen werden, d.h. es wird nicht automatisch mit dem Befehl `library(tidyverse)` geladen.

Auch hier müssen kategoriale Variablen anschliessend zu Faktoren konvertiert werden:

```
zufriedenheit_xl$vpn <- as.factor(zufriedenheit_xl$vpn)
zufriedenheit_xl$messzeitpunkt <- as.factor(zufriedenheit_xl$messzeitpunkt)
```

3.2.4 RData-Dateien

Unsere letzte Option ist eine `RData`-Datei. Dies ist ein *binary*- Datei, d.h. es handelt sich in diesem Fall um *keine* Textdatei. Wir können mehrere Objekte aus dem `R` Workspace in eine `RData`-Datei speichern, und wieder daraus laden. Der grosse Vorteil daran ist, dass Attribute, wie z.B. die Objektklasse `factor` und die Faktorstufen `levels` mitgespeichert werden, und nicht wie beim Speichern z.B. als CSV- oder Excel-Datei verloren gehen. Ein weiterer Vorteil ist, dass die Datei komprimiert werden kann. Demgegenüber steht die Tatsache, dass eine `RData` Datei nicht ohne Weiteres von anderen Statistikprogrammen gelesen werden kann. Deshalb sollte man besonders dann Textdateien (.csv) verwenden, wenn man einen Datensatz anderen Personen zur Verfügung stellen will.

Ausserdem ist es empfehlenswert, alle Schritte der Datenanalyse (inkl. Einlesen der Daten) in einem `R` Script File festzuhalten. Der Import von Dateien und die Umwandlung von Variablen kann so nachvollzogen und das Einlesen der Daten bei späteren `R` Sitzungen automatisiert werden.

Mit der Funktion `save()` können wir Objekte, die sich im Global Environment befinden, direkt als `.RData` (oder `.Rda`) Dateien speichern. Z.B. können wir so den Data Frame `zufriedenheit` als `RData`-Datendatei auf unserem Computer speichern:

```
save(zufriedenheit, file = "data/zufriedenheit.Rda")
```

Wir können dieser Funktion auch eine Liste von Objekten als Argument übergeben:

```
save(zufriedenheit, zufriedenheit_spss, zufriedenheit_xl,
  file = "data/zufriedenheit_alle.Rda"
)
```

`zufriedenheit_alle.Rda` enthält nun alle drei Data Frames.

Mit `load()` können wir diese auch wieder laden:

```
load(file = "data/zufriedenheit_alle.Rda")
```

4 Daten transformieren

4.1 Tidy data

Wenn in der Statistik/Data Science über Datensätze gesprochen wird, werden oft die Begriffe *wide* und *long* verwendet. Diese beziehen sich auf die *Form* eines Datensatzes: ein **wide** Datensatz hat mehr Spalten als ein entsprechender **long** Datensatz und umgekehrt hat ein **long** Datensatz mehr Zeilen als ein entsprechender **wide** Datensatz.

Genauer gesagt hat in einem *long* Datensatz jede Beobachtung eine eigene Zeile und jede Variable ist eine Spalte. In einem *wide* Datensatz können (insbesondere messwiederholte) Variablen über mehrere Spalten verteilt werden, und somit jede Zeile eine bestimmte *Person* repräsentieren.

Betrachten wir folgendes Beispiel: Wir haben zwei Beobachtungen (irgendeiner psychologischen Variable) für zwei Personen, zu drei verschiedenen Zeitpunkten. Nennen wir die Zeitpunkte `t1`, `t2` und `t3`. Die folgende Darstellung eines solchen Datensatzes `df` gilt als *wide format*:

```
# A tibble: 2 x 4
  name        t1    t2    t3
  <fct>    <dbl> <dbl> <dbl>
1 Marshall     4     5     7
2 Skye         3     6     7
```

Hier steht jede Zeile für eine Person und die Beobachtungen sind auf die drei Spalten `t1`, `t2` und `t3` verteilt. Diese Spalten repräsentieren aber eigentlich die drei Stufen eines (messwiederholten) Faktors `zeit`. Wir können genau die gleichen Daten in einem *längeren* Format darstellen, in dem jede Zeile einer *Beobachtung* entspricht und jede Spalte eine *Variable* darstellt. Insgesamt gibt es dann drei Variablen: `name`, `zeit` und `punkte`, wobei `punkte` hier als Platzhalter für die gemessenene (psychologische) Variable steht, deren Ausprägungen in dieser Spalte aufgeführt werden.

Um den obigen *wide* Datensatz in einen *long* Datensatz zu transformieren, verwenden wir `pivot_longer()` aus dem package `tidyr`. Die Details zu dieser Funktion behandeln wir weiter unten im Abschnitt Reshaping: `tidyr`.

```r
df |>
  pivot_longer(!name, names_to = "zeit", values_to = "punkte")
```

```
# A tibble: 6 x 3
  name     zeit  punkte
  <fct>    <chr>  <dbl>
1 Marshall t1         4
2 Marshall t2         5
3 Marshall t3         7
4 Skye     t1         3
5 Skye     t2         6
6 Skye     t3         7
```

Wir haben jetzt nicht mehr für jede Person eine Zeile. Die Werte von Variablen, die sich nicht für jede Beobachtung (= jede Zeile) ändern, werden jetzt über die Zeilen hinweg wiederholt (gilt für `name` und `zeit`).

Ein Datensatz, der auf diese Weise organisiert ist, wird oft auch als *tidy* bezeichnet.

Wie wir später noch sehen werden, erfordern viele Arten von statistischen Analysen und insbesondere Grafik-Funktionen einen *long* Datensatz, und daher muss oft erstaunlich viel Zeit für die Organisation von Daten für die weitere Analyse aufgewendet werden (diese Art von Arbeit wird oft als "data wrangling" bezeichnet).

In diesem Kurs werden wir mit den `tidyverse` Packages zur Manipulation und Transformation von Daten arbeiten. Natürlich gibt es auch andere Möglichkeiten der Datenbearbeitung, aber unserer Ansicht nach bietet die `tidyverse`-Methode die konsistenteste Art der Arbeit mit Daten und reduziert daneben auch die kognitive Belastung der Benutzer.

Die Pakete, die für Datenmanipulationen verwendet werden, sind `tidyr` für die Transformation/Umformung von Datensätzen und `dplyr` für die Manipulation/Bearbeitung von Datensätzen und den dort enthaltenen Variablen. Für Faktoren benutzen wir zudem eine Funktion aus dem package `forcats`. Die Funktionen, die wir uns anschauen werden, sind:

Package	Funktion	Verwendung
`tidyr`	`pivot_longer()`	erhöht die Anzahl der Zeilen, verringert die Anzahl der Spalten
`tidyr`	`pivot_wider()`	verringert die Anzahl der Zeilen, erhöht die Anzahl der Spalten
`tidyr`	`drop_na()`	löscht alle Zeilen eines Datensatzes, die missing values (NA) enthalten
`dplyr`	`rename()`	zum Umbenennen von Variablen
`dplyr`	`select()`	wählt Variablen (Spalten) aus

Package	Funktion	Verwendung
`dplyr`	`relocate()`	verändert die Reihenfolge von Variablen (Spalten)
`dplyr`	`filter()`	wählt Beobachtungen (Zeilen) aus
`dplyr`	`arrange()`	sortiert einen Datensatz nach einer bestimmten Variablen
`dplyr`	`mutate()`	erstellt neue Variablen und ändert bereits vorhandene Variablen
`dplyr`	`case_when()`	zum Rekodieren von vorhandenen Variablen
`forcats`	`fct_recode()`	zum Rekodieren/Umbenennen von Faktorstufen
`dplyr`	`group_by()`	ermöglicht Operationen an Teilmengen der Daten
`dplyr`	`summarize()` / `summarise()`	fasst Daten zusammen

Mit diesen Funktionen können sehr komplexe Datenmanipulationen durchgeführt werden und trotzdem bleibt der `R` Code relativ übersichtlich. Neben den oben genannten enthält `dplyr` noch viele weitere Funktionen. Dazu gehören z.B. Funktionen, die es ermöglichen, verschiedene Datensätze miteinander zu verbinden (zu "mergen").

4.2 Der Pipe Operator

Wir haben schon festgestellt, dass Code schnell unübersichtlich werden kann, wenn wir eine Sequenz von Operationen ausführen. Dies führt zu verschachtelten Funktionsaufrufen.

Beispiel: Wir haben einen numerischen Vektor von $n = 10$ Messwerten (hier zu Übungszwecken mit `rnorm()` aus normalverteilten Zufallszahlen generiert) und wollen diese zuerst zentrieren, dann die Standardabweichung berechnen, und anschliessend noch auf zwei Nachkommastellen runden.

```
set.seed(1283)
stichprobe <- rnorm(10, 24, 5)
stichprobe
```

```
 [1] 24.74984 21.91726 23.98551 19.63019 23.96428 22.83092 18.86240 19.08125
 [9] 23.76589 21.88846
```

Die gewünschte Berechnung der gerundeten Standardabweichung der zentrierten Werte können wir als verschachtelte Funktionsaufrufe durchführen:

```
round(sd(scale(stichprobe,
               center = TRUE,
               scale = FALSE)),
      digits = 2)
```

```
[1] 2.19
```

Die Funktion `scale()`, `sd()` und `round()` werden nun der Reihe nach ausgeführt (von innen nach aussen), und zwar so, dass der Output einer Funktion der nächsten Funktion als Input übergeben wird.

Die Funktionen `scale()` und `round()` haben zusätzlich noch Argumente: `center = TRUE`, `scale = FALSE`, bzw. `digits = 2`. Dies ist zwar effizient, aber führt zu Code, der schwierig zu lesen ist.

Eine Alternative dazu wäre, die Zwischenschritte als Variablen zu speichern:

```
stichprobe_z <- scale(stichprobe, center = TRUE,
                      scale = FALSE)

sd_stichprobe_z <- sd(stichprobe_z)
sd_stichprobe_z_gerundet <- round(sd_stichprobe_z,
                                  digits = 2)

sd_stichprobe_z_gerundet
```

```
[1] 2.19
```

So steht jeder der Teilschritte in einer eigenen Zeile und wir verstehen den Code ohne Probleme. Diese Methode erfordert jedoch, dass wir zwei Variablen definieren, die wir eigentlich gar nicht brauchen.

Es gibt nun aber eine sehr elegante Methode, um Funktionen nacheinander aufzurufen, ohne diese Funktionen ineinander verschachtelt schreiben zu müssen: wir benützen dafür den `pipe` Operator. Er sieht so aus:

```
|>
```

und ist als *Infix*-Operator definiert. Das bedeutet, dass er *zwischen* zwei Objekten steht, ähnlich wie ein mathematischer Operator. Der Name `pipe` ist so zu verstehen, dass wir ein Objekt an eine Funktionen “weiterleiten” oder “übergeben”.

Dieser `pipe` Operator wird so oft verwendet, dass er schon eine eigene Tastenkombination hat: `Cmd`+`Shift`+`M` (MacOS) oder `Ctrl`+`Shift`+`M` (Windows und Linux).

> **Hinweis**
>
> Probieren Sie diese Tastenkombination aus, um den `pipe` Operator einzufügen.
> Falls bei Ihnen statt `|>` das Symbol `%>%` erscheint, müssen Sie in den RStudio Optionen unter `Code` bei "Use native pipe operator, `|>` (requires R 4.1+)" ein Häkchen setzen (siehe Kapitel 1.1). `%>%` ist der ursprüngliche `pipe` Operator aus dem `tidyverse`-Package `magrittr`, der immer noch häufig verwendet wird. Die geringfügigen Funktionsunterschiede zwischen den beiden `pipe` Operatoren sind für unsere Zwecke vernachlässigbar.

Unser Beispiel von oben:

```
round(sd(scale(stichprobe,
               center = TRUE,
               scale = FALSE)),
      digits = 2)
```

```
[1] 2.19
```

wird mit dem `|>` Operator zu:

```
stichprobe |>
  scale(center = TRUE, scale = FALSE) |>
  sd() |>
  round(digits = 2)
```

```
[1] 2.19
```

Dieser Code ist so zu lesen:

1) Wir beginnen mit dem Objekt `stichprobe` und übergeben es mit `|>` als Argument an die Funktion `scale()`
2) Wir wenden `scale()`, mit den zusätzlichen Argumenten `center = TRUE, scale = FALSE` darauf an, und übergeben den Output als Argument an die Funktion `sd()`
3) Wir wenden `sd()` an (ohne weitere Argumente) und reichen den Output als Argument weiter an `round()`
4) `round()`, mit dem weiteren Argument `digits = 2`, wird ausgeführt. Da kein weiterer `pipe` folgt, wird der Output in die Konsole geschrieben.

Somit ist klar: wenn wir das Resultat weiterverwenden möchten, müssen wir es einer Variablen zuweisen:

```
sd_stichprobe_z_gerundet <- stichprobe |>
  scale(center = TRUE, scale = FALSE) |>
  sd() |>
  round(digits = 2)

sd_stichprobe_z_gerundet
```

```
[1] 2.19
```

Wir übergeben also mit `|>` ein Objekt an eine Funktion. Wenn wir nichts weiter spezifizieren, ist dieses Objekt das erste Argument der Funktion. Die grossen Vorteile sind:

- Unser Code ist lesbarer.
- Wir mussten keine unnötigen Variablen definieren.

Syntax des Pipe Operators

Der `|>` Operator wird im Allgemeinen wie folgt verwendet. Nehmen wir an `f()`, `g()` und `h()` seien Funktionen, dann gilt:

```
x |> f()

# ist äquivalent zu

f(x)
```

Wenn `y` ein weiteres Argument von `f()` ist, dann gilt:

```
x |> f(y)

# ist äquivalent zu

f(x, y)
```

Wenn wir der Reihe nach `f()`, `g()` und `h()` anwenden, dann gilt:

```r
x |> f() |> g() |> h()

# oder

x |>
  f() |>
  g() |>
  h()

# ist äquivalent zu

h(g(f(x)))
```

Wir müssen das Objekt `x` nicht als erstes Argument weitergeben, sondern können es an einer beliebigen Stelle der Funktion verwenden, an die `x` weitergegeben wird. Dafür brauchen wir den Argument-Platzhalter `_`. Dabei muss der Name des Arguments, bei dem der Platzhalter eingesetzt wird, immer explizit angegeben werden:

```r
x |> f(y, argument = _)

# ist äquivalent zu

f(y, argument = x)
```

Zum Beispiel könnten wir den oben in Kapitel 3.2 eingelesenen Datensatz `zufriedenheit` als letztes Argument `data` an die Funktion `aggregate()` pipen, um die Mittelwerte (Argument: `FUN = mean`) des Zufriedenheits-Ratings für die beiden Messzeitpunkten zu berechnen:

```r
load(file = "data/zufriedenheit.Rda")

zufriedenheit
```

```
# A tibble: 8 x 3
  vpn   messzeitpunkt zufriedenheit
  <fct> <fct>                 <dbl>
1 vp_1  t1                     64.3
2 vp_2  t1                     65.1
3 vp_3  t1                     54.2
4 vp_4  t1                     54.5
5 vp_1  t2                     67.2
6 vp_2  t2                     75.6
```

```
7 vp_3  t2                   84.6
8 vp_4  t2                   73.9
```

```
zufriedenheit |>
  aggregate(zufriedenheit ~ messzeitpunkt, FUN = mean, data = _)
```

```
  messzeitpunkt zufriedenheit
1            t1       59.5325
2            t2       75.3425
```

In den meisten Fällen ist jedoch das Objekt, welches übergeben wird, gleichzeitig auch das erste Argument der nächsten Funktion (vor allem für die `tidyverse` Funktionen), so dass wir diesen Platzhalter selten brauchen.

Wenn wir mit den Funktionen der `tidyr` und `dplyr` Packages arbeiten, werden wir diesen Operator sehr häufig benutzen. Ein weiterer Grund, sich damit anzufreunden, ist, dass dieser immer häufiger Verwendung findet und sehr viele Beispiele im Internet (z.B. auf Stackoverflow) den `|>` Operator benutzen.

4.3 Reshaping: `tidyr`

Um einen Datensatz zu transformieren (reshaping) brauchen wir zwei Funktionen: `pivot_longer()` und `pivot_wider()`, beide befinden sich im Package `tidyr`.

```
library(tidyr)
```

4.3.1 `pivot_longer()`

Wir benutzen `pivot_longer()`, wenn wir einen *wide* Datensatz zu einem *long* Datensatz konvertieren möchten. `pivot_longer()` wird also dazu verwendet, mehrere Spalten, die die Stufen eines Faktors repräsentieren, zu einer Spalte zusammenzufügen, welche den Faktor selbst repräsentiert. Die Werte in den ursprünglichen Variablen werden in einer Werte-Variable zusammengefasst.

> Hinweis
>
> `pivot_longer()` nimmt mehrere Spalten als Eingabe und erzeugt eine Variable/Spalte, deren Datenwerte aus den Spaltennamen bestehen `(names_to)`. Ausserdem wird eine Variable/Spalte erstellt, die die in den ursprünglichen Spalten enthaltenen Datenwerte beinhaltet `(values_to)`.

Die Syntax von `pivot_longer()` sieht so aus:

```
pivot_longer(data, cols, names_to, values_to)
```

oder mit `|>`

```
data |>
  pivot_longer(cols, names_to, values_to)
```

Die Argumente haben die folgende Bedeutung:

```
data:       ein data frame
cols:       die Spalten, deren Werte in einer Spalte zusammengeführt werden sollen
names_to:   "Name" (string) der neuen Spalte, die aus den ausgewählten Spaltennamen
erstellt wird
values_to   "Name" (string) der Spalte, die aus den Beobachtungen in den Zellen
erstellt wird
```

Sehen wir uns noch einmal das oben verwendete Beispiel an. Wir haben einen *wide* Datensatz (mit Namen `df`) mit zwei Personen, die zu drei Zeitpunkten gemessen wurden.

```
df
```

```
# A tibble: 2 x 4
  name        t1    t2    t3
  <fct>    <dbl> <dbl> <dbl>
1 Marshall     4     5     7
2 Skye         3     6     7
```

Wir nehmen an, dass `t1`, `t2` und `t3` keine wirklich getrennten Variablen sind, sondern dass sie als Stufen eines Zeitfaktors betrachtet werden können (z.B. als drei Zeitpunkte in einem Messwiederholungsdesign). Die Werte in den Spalten `t1`, `t2` und `t3` beziehen sich auf die gleiche Art von Messung (z.B. Punktzahl) und sollten daher tatsächlich durch *eine* Variable in einem *long* Datensatz repräsentiert werden. Das sind also die im Argument `cols` zu bestimmenden Variablen/Spalten.

Wir möchten, dass unser neuer Faktor den Namen `zeit` erhält und die gemessene Variable `punkte` genannt wird.

Die `name`-Variable sollte ignoriert werden, d.h. sie sollte bei der Umstrukturierung nicht mit benutzt werden (also von der Funktion nicht als weiterer Messzeitpunkt betrachtet werden).

Hinweis

Um beim Subsetting eine Variable wegzulassen (und damit alle anderen auszuwählen), haben wir weiter oben den Minus-Operator benutzt (z.B. `-name`). Wenn wir mit den `tidyverse`-Funktionen arbeiten, benutzen wir an der Stelle von `-` den logischen Operator `!` (NOT). Um im Folgenden das `cols`-Argument zu definieren, benutzen wir daher `!name` statt `-name` (obwohl letzteres auch funktionieren würde, aber - zumindest für `tidyverse`-Funktionen - als veraltet gilt).

```
library(tidyr)
df_long <- df |>
  pivot_longer(!name, names_to = "zeit", values_to = "punkte")
```

```
df_long
```

```
# A tibble: 6 x 3
  name     zeit  punkte
  <fct>    <chr>  <dbl>
1 Marshall t1         4
2 Marshall t2         5
3 Marshall t3         7
4 Skye     t1         3
5 Skye     t2         6
6 Skye     t3         7
```

`zeit` sollte im nächsten Schritt noch als Faktor definiert werden:

```
df_long$zeit <- as.factor(df_long$zeit)
```

```
df_long
```

```
# A tibble: 6 x 3
  name     zeit  punkte
  <fct>    <fct>  <dbl>
1 Marshall t1         4
2 Marshall t2         5
3 Marshall t3         7
4 Skye     t1         3
5 Skye     t2         6
6 Skye     t3         7
```

Eine weitere Eigenschaft von `pivot_longer()` ist, dass die Werte derjenigen Variablen, die **nicht** Teil des Reshapings sind (in unserem Fall also einzig die Variable `name`) wiederholt werden. Die Zeilen 1-3 beinhalten jetzt also Beobachtungen, die zu `Marshall` gehören und die Zeilen 4-6 Beobachtungen, die zu `Skye` gehören.

Der Aufruf der Funktion `pivot_longer()` hätte auch expliziter geschrieben werden können:

```
df |>
  pivot_longer(cols = !name,
               names_to = "zeit",
               values_to = "punkte")
```

Im `cols`-Argument haben wir `!name` verwendet: so werden alle Spalten ausser `name` für die wide-to-long-Transformation verwendet. Wir hätten stattdessen auch direkt die drei Zeit-Spalten auswählen können:

```
df |>
  pivot_longer(cols = c(t1, t2, t3),
               names_to = "zeit",
               values_to = "punkte")
```

```
# A tibble: 6 x 3
  name     zeit  punkte
  <fct>    <chr>  <dbl>
1 Marshall t1         4
2 Marshall t2         5
3 Marshall t3         7
4 Skye     t1         3
5 Skye     t2         6
6 Skye     t3         7
```

Wir werden weitere und allgemeinere Möglichkeiten für die Auswahl von Spalten/Variablen eines Datensatzes kennenlernen, wenn wir uns das `dplyr` Package ansehen.

Beispiel

Schauen wir uns ein weiteres Beispiel an, diesmal unter Verwendung des `therapie`-Datensatzes, den wir zunächst von Github herunterladen und anschliessend in unserem `data`-Ordner abspeichern: `therapie.sav`.

Jetzt lesen wir den Datensatz lokal aus unserem `data`-Ordner ein und konvertieren die dort vorhandenen Faktoren:

```
library(haven)
therapie <- read_sav("data/therapie.sav")
therapie$id <- as_factor(therapie$id)
therapie$gruppe <- as_factor(therapie$gruppe)
```

```
therapie
```

```
# A tibble: 100 x 4
   id    gruppe         pretest posttest
   <fct> <fct>            <dbl>    <dbl>
 1 1     Kontrollgruppe    4.29     3.21
 2 2     Kontrollgruppe    6.18     5.99
 3 3     Kontrollgruppe    3.93     4.17
 4 4     Kontrollgruppe    5.06     4.76
 5 5     Kontrollgruppe    6.45     5.64
 6 6     Kontrollgruppe    4.49     4.67
 7 7     Kontrollgruppe    4.60     4.24
 8 8     Kontrollgruppe    4.46     3.34
 9 9     Kontrollgruppe    4.76     4.11
10 10    Kontrollgruppe    5.12     5.29
# i 90 more rows
```

Die Struktur dieses Datensatzes ähnelt der des obigen Beispiels. Wir wollen die Spalten `pretest` und `posttest` kombinieren, da sie Stufen eines gemeinsamen Faktors `zeit` sind. Die Daten in den Zellen repräsentieren also wiederholte Messungen. Da die Variablen ausser den Messzeitpunkten `pretest` und `posttest` keine weiteren Informationen aufweisen, können wir aus dem Datensatz allein nicht erkennen, was eigentlich die gemessene (psychologische) Variable ist. Im obigen Einführungsbeispiel haben wir die beim Pivotieren gebildete Variable einfach `punkte` genannt. Jetzt wollen wir der Variable aber einen Namen geben, der dem Inhalt der Messungen entspricht. Dafür benötigen wir die Zusatzinformation, dass es sich bei den Werten unter `pretest` und `posttest` um eine Skala psychischer Stresssymptome handelt, die auf einer Skala von 0 bis 7 gemessen wurden. Wir wollen die Variable daher `symptome` nennen.

Ausserdem sollen die Variablen `id` und `gruppe` für das Pivotieren ignoriert werden, aber im Datensatz verbleiben.

Führen wir die wide-to-long-Transformation durch:

```
therapie_long <- therapie |>
  pivot_longer(c("pretest", "posttest"),
               names_to = "zeit",
               values_to = "symptome")
```

```r
# zeit sollte ein Faktor sein
therapie_long$zeit <- factor(therapie_long$zeit,
                             levels = c("pretest", "posttest"))
# Hier definieren wir explizit 'pretest' als erste und 'posttest' als zweite
# Faktorstufe, weil es inhaltlich und für spätere Datenanalysen Sinn macht, den
# zeitlich früher gelegenen Messzeitpunkt als erste Faktorstufe zu bestimmen.
# Ohne explizite Definition über das levels-Argument wäre hier 'posttest' die
# erste Faktorstufe (weil alphabetisch vor 'pretest').
```

```r
therapie_long
```

```
# A tibble: 200 x 4
   id    gruppe          zeit     symptome
   <fct> <fct>           <fct>       <dbl>
 1 1     Kontrollgruppe pretest      4.29
 2 1     Kontrollgruppe posttest     3.21
 3 2     Kontrollgruppe pretest      6.18
 4 2     Kontrollgruppe posttest     5.99
 5 3     Kontrollgruppe pretest      3.93
 6 3     Kontrollgruppe posttest     4.17
 7 4     Kontrollgruppe pretest      5.06
 8 4     Kontrollgruppe posttest     4.76
 9 5     Kontrollgruppe pretest      6.45
10 5     Kontrollgruppe posttest     5.64
# i 190 more rows
```

4.3.2 `pivot_wider()`

`pivot_wider()` ist das Gegenteil von `pivot_longer()`. Diese Funktion nimmt einen Faktor und eine Messvariable und "verteilt" die Werte der Messvariable über neue Spalten, welche die Stufen des Faktors repräsentieren. Dies bedeutet, dass wir `pivot_wider()` verwenden, wenn wir aus einem *long* Datensatz einen *wide* Datensatz machen wollen.

Die `pivot_wider()` Syntax sieht so aus:

```r
pivot_wider(data, names_from, values_from)

# oder

data |>
  pivot_wider(names_from, values_from)
```

An unserem einfachen Beispiel illustriert bedeutet dies, dass wir neue Variablen für jede Stufe des Faktors `zeit` erstellen wollen, uns zwar unter Verwendung der Werte der Variable `punkte`.

```
df_long
```

```
# A tibble: 6 x 3
  name     zeit  punkte
  <fct>    <fct>  <dbl>
1 Marshall t1         4
2 Marshall t2         5
3 Marshall t3         7
4 Skye     t1         3
5 Skye     t2         6
6 Skye     t3         7
```

```
df_wide <- df_long |>
  pivot_wider(names_from = zeit, values_from = punkte)
```

```
df_wide
```

```
# A tibble: 2 x 4
  name        t1    t2    t3
  <fct>    <dbl> <dbl> <dbl>
1 Marshall     4     5     7
2 Skye         3     6     7
```

`df_wide` sieht genau so aus wie der ursprüngliche Datensatz `df`. Wir können noch prüfen, ob beide wirklich äquivalent sind:

```
all.equal(df, df_wide)
```

```
[1] TRUE
```

Beispiel

Nun konvertieren wir den `therapie`-Datensatz von *long* zurück zu *wide*:

```
therapie_long
```

```
# A tibble: 200 x 4
   id    gruppe         zeit     symptome
   <fct> <fct>          <fct>       <dbl>
 1 1     Kontrollgruppe pretest      4.29
 2 1     Kontrollgruppe posttest     3.21
 3 2     Kontrollgruppe pretest      6.18
 4 2     Kontrollgruppe posttest     5.99
 5 3     Kontrollgruppe pretest      3.93
 6 3     Kontrollgruppe posttest     4.17
 7 4     Kontrollgruppe pretest      5.06
 8 4     Kontrollgruppe posttest     4.76
 9 5     Kontrollgruppe pretest      6.45
10 5     Kontrollgruppe posttest     5.64
# i 190 more rows
```

```
therapie_wide <- therapie_long |>
  pivot_wider(names_from = zeit, values_from = symptome)
```

```
therapie_wide
```

```
# A tibble: 100 x 4
   id    gruppe         pretest posttest
   <fct> <fct>            <dbl>    <dbl>
 1 1     Kontrollgruppe    4.29     3.21
 2 2     Kontrollgruppe    6.18     5.99
 3 3     Kontrollgruppe    3.93     4.17
 4 4     Kontrollgruppe    5.06     4.76
 5 5     Kontrollgruppe    6.45     5.64
 6 6     Kontrollgruppe    4.49     4.67
 7 7     Kontrollgruppe    4.60     4.24
 8 8     Kontrollgruppe    4.46     3.34
 9 9     Kontrollgruppe    4.76     4.11
10 10    Kontrollgruppe    5.12     5.29
# i 90 more rows
```

4.3.3 Fehlende Werte auschliessen: `drop_na()`

Eine ganz wichtige Funktion im `tidyr` Package ist `drop_na()`. Damit können wir alle Zeilen löschen, welche fehlende Werte haben.

Als Illustration dient dieses Beispiel:

```
df_1 <- tibble(var1 = factor(c("a", NA, "b", "b")), var2 = c(1, NA, 2, 3), var3 = c(NA, 21, 2
df_1
```

```
# A tibble: 4 x 3
  var1   var2  var3
  <fct> <dbl> <dbl>
1 a         1    NA
2 <NA>     NA    21
3 b         2    24
4 b         3    NA
```

```
df_1 |>
  drop_na()
```

```
# A tibble: 1 x 3
  var1   var2  var3
  <fct> <dbl> <dbl>
1 b         2    24
```

Wir haben einen Faktor (`var1`) und zwei numerische Variablen (`var2` und `var3`). Für viele statistische Analysen benötigen wir vollständige Datensätze, d.h. Datensätze ohne fehlende Werte/`NA`s. In diesem Fall bleibt von den vier Beobachtungen nur noch eine übrig, da nur Zeile 3 des Datensatzes kein einziges `NA` aufweist.

Angenommen, die für eine bestimmte Analyse interessierenden Variablen sind `var1` und `var2`, während `var3` (könnte z.B. die Altersvariable sein), keine Rolle spielt.

Dann wäre es sinnvoll, zunächst einen Subdatensatz zu bilden, der nur die ersten beiden Variablen enthält, und erst dann `drop_na()` auszuführen:

```
df_1a <- df_1[1:2]

df_1a |>
  drop_na()
```

```
# A tibble: 3 x 2
  var1   var2
  <fct> <dbl>
1 a         1
2 b         2
3 b         3
```

So verbleiben drei Beobachtungen für die nachfolgende Datenanalyse (alle ausser die zweite Zeile/Person aus `df_1`, die sowohl in `var1` als auch in `var2` ein `NA` aufweist).

Diese Funktion ist sehr nützlich, sollte aber wie im Beispiel gezeigt vorsichtig verwendet werden, denn sonst löscht man ggf. Zeilen, die `NA`s in Variablen besitzen, die für die vorliegende Fragestellung nicht relevant sind.

4.4 Data Wrangling: `dplyr`

Nun sind wir in der Lage, Datensätze von *wide* zu *long* und umgekehrt zu transformieren, aber damit ist unsere Arbeit noch nicht getan. Die meisten Datensätze müssen bearbeitet werden, bevor sie analysiert werden können: Wir müssen z.B. Fälle und/oder Variablen auswählen, Werte recodieren, Variablen umbenennen, nach bestimmten Variablen sortieren, neue Variablen bilden, Datensätze nach Gruppierungsvariablen aufteilen oder Variablen zusammenfassen. Im Englischen wird diese Art von Arbeit an und mit Daten oft als *data wrangling* bezeichnet.

Das `dplyr` Package stellt Funktionen für alle diese Aufgaben zur Verfügung (und noch viele mehr, wir betrachten hier nur eine Auswahl). `dplyr` besteht sozusagen aus Verben (Funktionen) für all diese Operationen, und diese Funktionen können - je nach Bedarf - auf sehr elegante Weise zusammengesetzt werden.

> Hinweis
>
> Wir werden in dieser Vorlesung nur einen kleinen Teil der Funktionalität von `dplyr` kennenlernen. Wer mehr wissen will, kann dies in den Help Pages nachschauen: `help(package = "dplyr")`.

Für den Rest dieses Kapitels arbeiten wir mit *long* Datensätzen. Falls ein Datensatz *wide* ist, sollte er zu *long* konvertiert werden.

Wir laden zuerst das `dplyr` Package:

```
library(dplyr)
```

Wir sehen uns nun der Reihe nach die verschiedenen Funktionen und deren Verwendung an. Wir verwenden immer den `|>` Operator. Der Input Dataframe ist dabei immer als erstes Argument der Funktion zu verstehen. Für die Beispiele verwenden wir die Datensätze `df_long.Rda`, `therapie_long.Rda`, `therapie_wide.Rda` sowie `alkohol_aggression.csv` und `beispieldaten.sav`. Speichern Sie die heruntergeladenen Datenfiles in Ihrem `data`-Ordner ab.

Während die ersten drei bereits weiter oben geladen und verwendet wurden, lesen wir die letzten beiden beim jeweiligen Beispiel ein.

Bei allen unten stehenden Beispielen gilt: wenn wir das Resultat weiterverwenden möchten, müssen wir den Output einer neuen Variablen zuweisen.

4.4.1 Variablen umbenennen mit `rename()`

Variablen können mit `rename()` umbenannt werden. Die nicht umbenannten Variablen verbleiben im Datensatz.

Syntax:

```
df |> rename(neuer_name = alter_name)
```

Beispiel

```
# "zeit" umbennen in "messzeitpunkt"
therapie_long |>
  rename(messzeitpunkt = zeit)
```

```
# A tibble: 200 x 4
   id    gruppe         messzeitpunkt symptome
   <fct> <fct>          <fct>            <dbl>
 1 1     Kontrollgruppe pretest           4.29
 2 1     Kontrollgruppe posttest          3.21
 3 2     Kontrollgruppe pretest           6.18
 4 2     Kontrollgruppe posttest          5.99
 5 3     Kontrollgruppe pretest           3.93
 6 3     Kontrollgruppe posttest          4.17
 7 4     Kontrollgruppe pretest           5.06
 8 4     Kontrollgruppe posttest          4.76
 9 5     Kontrollgruppe pretest           6.45
10 5     Kontrollgruppe posttest          5.64
# i 190 more rows
```

4.4.2 Variablen auswählen mit `select()`

Mit der Funktion `select()` wählen wir Variablen aus einem Datensatz aus.

Syntax:

```r
# df steht für data frame
select(df, variable1, variable2, !variable3)

# oder

df |> select(variable1, variable2, !variable3)
```

`select()` wählt hier aus dem Dataframe `df` die Variablen `variable1` und `variable2` aus, `variable3` wird weggelassen.

Beispiele

```r
# nur ID
df_long |> select(name)
```

```
# A tibble: 6 x 1
  name
  <fct>
1 Marshall
2 Marshall
3 Marshall
4 Skye
5 Skye
6 Skye
```

```r
# Messzeitpunkt und Score
df_long |> select(zeit, punkte)
```

```
# A tibble: 6 x 2
  zeit  punkte
  <fct>  <dbl>
1 t1         4
2 t2         5
3 t3         7
4 t1         3
5 t2         6
6 t3         7
```

```r
# oder

df_long |> select(!name)
```

```
# A tibble: 6 x 2
  zeit  punkte
  <fct>  <dbl>
1 t1         4
2 t2         5
3 t3         7
4 t1         3
5 t2         6
6 t3         7
```

```r
therapie_long |>
  select(id, gruppe, symptome)
```

```
# A tibble: 200 x 3
   id    gruppe         symptome
   <fct> <fct>             <dbl>
 1 1     Kontrollgruppe     4.29
 2 1     Kontrollgruppe     3.21
 3 2     Kontrollgruppe     6.18
 4 2     Kontrollgruppe     5.99
 5 3     Kontrollgruppe     3.93
 6 3     Kontrollgruppe     4.17
 7 4     Kontrollgruppe     5.06
 8 4     Kontrollgruppe     4.76
 9 5     Kontrollgruppe     6.45
10 5     Kontrollgruppe     5.64
# i 190 more rows
```

```r
# wählt alle Variablen von gruppe bis symptome aus
# (also auch die Variablen dazwischen)
therapie_long |>
  select(gruppe:symptome)
```

```
# A tibble: 200 x 3
   gruppe         zeit     symptome
   <fct>          <fct>       <dbl>
```

```
 1 Kontrollgruppe pretest      4.29
 2 Kontrollgruppe posttest     3.21
 3 Kontrollgruppe pretest      6.18
 4 Kontrollgruppe posttest     5.99
 5 Kontrollgruppe pretest      3.93
 6 Kontrollgruppe posttest     4.17
 7 Kontrollgruppe pretest      5.06
 8 Kontrollgruppe posttest     4.76
 9 Kontrollgruppe pretest      6.45
10 Kontrollgruppe posttest     5.64
# i 190 more rows
```

```
# Für die gleichzeitige Auswahl und Umbenennung
# einer Variable verwendet man =. Der neue Name
# steht auf der linken Seite des =, die alte
# Variable auf der rechten Seite:
therapie_long |>
  select(messzeitpunkt = zeit)
```

```
# A tibble: 200 x 1
   messzeitpunkt
   <fct>
 1 pretest
 2 posttest
 3 pretest
 4 posttest
 5 pretest
 6 posttest
 7 pretest
 8 posttest
 9 pretest
10 posttest
# i 190 more rows
```

Hilfsfunktionen für `select()`

Für die Auswahl von Variablen gibt es eine Reihe von sogenannten “helper functions”:

```
# einschliessen
therapie_long |> select(starts_with("gr"))
```

```
# A tibble: 200 x 1
   gruppe
   <fct>
 1 Kontrollgruppe
 2 Kontrollgruppe
 3 Kontrollgruppe
 4 Kontrollgruppe
 5 Kontrollgruppe
 6 Kontrollgruppe
 7 Kontrollgruppe
 8 Kontrollgruppe
 9 Kontrollgruppe
10 Kontrollgruppe
# i 190 more rows
```

```
therapie_long |> select(ends_with("me"))
```

```
# A tibble: 200 x 1
   symptome
      <dbl>
 1     4.29
 2     3.21
 3     6.18
 4     5.99
 5     3.93
 6     4.17
 7     5.06
 8     4.76
 9     6.45
10     5.64
# i 190 more rows
```

```
therapie_long |> select(contains("p"))
```

```
# A tibble: 200 x 2
   gruppe         symptome
   <fct>             <dbl>
 1 Kontrollgruppe     4.29
 2 Kontrollgruppe     3.21
 3 Kontrollgruppe     6.18
 4 Kontrollgruppe     5.99
```

```
 5 Kontrollgruppe     3.93
 6 Kontrollgruppe     4.17
 7 Kontrollgruppe     5.06
 8 Kontrollgruppe     4.76
 9 Kontrollgruppe     6.45
10 Kontrollgruppe     5.64
# i 190 more rows
```

```
vars <- c("gruppe", "symptome")
# einschliessendes ODER mit one_of()
therapie_long |> select(one_of(vars))
```

```
# A tibble: 200 x 2
   gruppe         symptome
   <fct>             <dbl>
 1 Kontrollgruppe     4.29
 2 Kontrollgruppe     3.21
 3 Kontrollgruppe     6.18
 4 Kontrollgruppe     5.99
 5 Kontrollgruppe     3.93
 6 Kontrollgruppe     4.17
 7 Kontrollgruppe     5.06
 8 Kontrollgruppe     4.76
 9 Kontrollgruppe     6.45
10 Kontrollgruppe     5.64
# i 190 more rows
```

```
# ausschliessen
therapie_long |> select(!starts_with("gr"))
```

```
# A tibble: 200 x 3
   id    zeit     symptome
   <fct> <fct>       <dbl>
 1 1     pretest      4.29
 2 1     posttest     3.21
 3 2     pretest      6.18
 4 2     posttest     5.99
 5 3     pretest      3.93
 6 3     posttest     4.17
 7 4     pretest      5.06
 8 4     posttest     4.76
```

```
 9 5     pretest      6.45
10 5     posttest     5.64
# i 190 more rows
```

```
therapie_long |> select(!ends_with("me"))
```

```
# A tibble: 200 x 3
   id    gruppe         zeit
   <fct> <fct>          <fct>
 1 1     Kontrollgruppe pretest
 2 1     Kontrollgruppe posttest
 3 2     Kontrollgruppe pretest
 4 2     Kontrollgruppe posttest
 5 3     Kontrollgruppe pretest
 6 3     Kontrollgruppe posttest
 7 4     Kontrollgruppe pretest
 8 4     Kontrollgruppe posttest
 9 5     Kontrollgruppe pretest
10 5     Kontrollgruppe posttest
# i 190 more rows
```

```
therapie_long |> select(!contains("p"))
```

```
# A tibble: 200 x 2
   id    zeit
   <fct> <fct>
 1 1     pretest
 2 1     posttest
 3 2     pretest
 4 2     posttest
 5 3     pretest
 6 3     posttest
 7 4     pretest
 8 4     posttest
 9 5     pretest
10 5     posttest
# i 190 more rows
```

```
vars <- c("gruppe", "symptome")
therapie_long |> select(!one_of(vars))
```

```
# A tibble: 200 x 2
   id    zeit
   <fct> <fct>
 1 1     pretest
 2 1     posttest
 3 2     pretest
 4 2     posttest
 5 3     pretest
 6 3     posttest
 7 4     pretest
 8 4     posttest
 9 5     pretest
10 5     posttest
# i 190 more rows
```

```
# Datensatz 'beispieldaten.sav' importieren und alle Items von swk1 bis swk12 auswählen
beispieldaten <- read_sav("data/beispieldaten.sav")
beispieldaten |>
  select(num_range("swk", 1:12))
```

```
# A tibble: 286 x 12
    swk1  swk2  swk3  swk4  swk5  swk6  swk7  swk8  swk9 swk10 swk11 swk12
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1     2     5     6     4     7     5     5     6     6     4     4     5
 2     4     6     4     4     6     5     5     6     5     4     4     4
 3     5     4     5     6     5     7     7     6     4     5     3     3
 4     3     6     6     5     7     7     6     6     7     7     3     4
 5     5     6     6     5     7     6     5     5     7     6     4     4
 6     3     5     5     4     5     7     5     6     6     5     3     5
 7     4     3     5     4     6     7     1     6     5     7     3     6
 8     5     4     7     7     7     7    NA     7     7     6     4     3
 9     6     6     5     5     7     6     5     6     6     6     6     6
10     5     3     4     5     5     5     5     4     5     5     5     6
# i 276 more rows
```

Damit können Variablen anhand von Suchkriterien ausgewählt oder ausgeschlossen werden. Wir werden in späteren Kapiteln weitere Beispiele dafür sehen.

```
# alle numerischen Variablen auswählen
therapie_wide |>
  select(where(is.numeric))
```

```
# A tibble: 100 x 2
   pretest posttest
     <dbl>    <dbl>
 1    4.29     3.21
 2    6.18     5.99
 3    3.93     4.17
 4    5.06     4.76
 5    6.45     5.64
 6    4.49     4.67
 7    4.60     4.24
 8    4.46     3.34
 9    4.76     4.11
10    5.12     5.29
# i 90 more rows
```

Hinweis

Die Funktion `where()` ist eine besondere Hilfsfunktion, weil in ihr wie hier mit `is.numeric()` eine weitere Funktion definiert wird. Diese Funktion prüft, ob die Variablen im Datensatz eine bestimmte Bedingung erfüllen (hier: ob sie vom Typ `numeric` sind). Die Bedingungsfunktion muss in diesem Fall *ohne* Funktionsklammern in die Funktion `where()` eingefügt werden (also als `where(is.numeric)`, nicht als `where(is.numeric())`).
Weitere Bedingungsfunktionen sind z.B. `is.factor()`, `is.character()` oder `is.labelled()`. Die Kombination von `where()` mit einer Bedingungsfunktion wird auch noch weiter unten in Verbindung mit `mutate()` eine wichtige Rolle spielen

4.4.3 Reihenfolge der Variablen verändern mit `relocate()`

Mit der Funktion `relocate()` verändern wir die Reihenfolge der Variablen. Per default werden die Variablen an den Anfang des Datensatzes verschoben. Mit den Argumenten `.after` und `.before` kann spezifiziert werden, wohin die Variablen verschoben werden sollen.

Beispiele

```
# pretest und posttest an den Anfang verschieben
therapie_wide |>
  relocate(pretest, posttest)
```

```
# A tibble: 100 x 4
   pretest posttest id    gruppe
     <dbl>    <dbl> <fct> <fct>
 1    4.29     3.21 1     Kontrollgruppe
 2    6.18     5.99 2     Kontrollgruppe
 3    3.93     4.17 3     Kontrollgruppe
 4    5.06     4.76 4     Kontrollgruppe
 5    6.45     5.64 5     Kontrollgruppe
 6    4.49     4.67 6     Kontrollgruppe
 7    4.60     4.24 7     Kontrollgruppe
 8    4.46     3.34 8     Kontrollgruppe
 9    4.76     4.11 9     Kontrollgruppe
10    5.12     5.29 10    Kontrollgruppe
# i 90 more rows
```

```
# gruppe vor posttest verschieben
therapie_wide |>
  relocate(gruppe, .before = posttest)
```

```
# A tibble: 100 x 4
   id    pretest gruppe         posttest
   <fct>   <dbl> <fct>             <dbl>
 1 1        4.29 Kontrollgruppe     3.21
 2 2        6.18 Kontrollgruppe     5.99
 3 3        3.93 Kontrollgruppe     4.17
 4 4        5.06 Kontrollgruppe     4.76
 5 5        6.45 Kontrollgruppe     5.64
 6 6        4.49 Kontrollgruppe     4.67
 7 7        4.60 Kontrollgruppe     4.24
 8 8        4.46 Kontrollgruppe     3.34
 9 9        4.76 Kontrollgruppe     4.11
10 10       5.12 Kontrollgruppe     5.29
# i 90 more rows
```

```
# id nach gruppe verschieben
therapie_wide |>
  relocate(id, .after = gruppe)
```

```
# A tibble: 100 x 4
   gruppe         id    pretest posttest
   <fct>          <fct>   <dbl>    <dbl>
```

```
 1 Kontrollgruppe 1        4.29     3.21
 2 Kontrollgruppe 2        6.18     5.99
 3 Kontrollgruppe 3        3.93     4.17
 4 Kontrollgruppe 4        5.06     4.76
 5 Kontrollgruppe 5        6.45     5.64
 6 Kontrollgruppe 6        4.49     4.67
 7 Kontrollgruppe 7        4.60     4.24
 8 Kontrollgruppe 8        4.46     3.34
 9 Kontrollgruppe 9        4.76     4.11
10 Kontrollgruppe 10       5.12     5.29
# i 90 more rows
```

4.4.4 Beobachtungen (Fälle) auswählen mit `filter()`

Beobachtungen oder Fälle (Zeilen) werden mit `filter()` ausgewählt, d.h. wir können damit Fälle auswählen, welche gewisse Bedingungen erfüllen.

Es können auch mehrere Bedingungen mit logischen Operatoren verknüpft werden:

Syntax:

```
df |> filter(variable1 < WERT1 & variable2 == WERT2)
```

Beispiele

```
# nur Kontrollgruppe auswählen
therapie_long |>
  filter(gruppe == "Kontrollgruppe")
```

```
# A tibble: 100 x 4
   id    gruppe         zeit     symptome
   <fct> <fct>          <fct>       <dbl>
 1 1     Kontrollgruppe pretest      4.29
 2 1     Kontrollgruppe posttest     3.21
 3 2     Kontrollgruppe pretest      6.18
 4 2     Kontrollgruppe posttest     5.99
 5 3     Kontrollgruppe pretest      3.93
 6 3     Kontrollgruppe posttest     4.17
 7 4     Kontrollgruppe pretest      5.06
 8 4     Kontrollgruppe posttest     4.76
```

```
 9 5     Kontrollgruppe pretest      6.45
10 5     Kontrollgruppe posttest     5.64
# i 90 more rows
```

```
# nur posttest
therapie_long |>
  filter(zeit == "posttest")
```

```
# A tibble: 100 x 4
   id    gruppe         zeit     symptome
   <fct> <fct>          <fct>       <dbl>
 1 1     Kontrollgruppe posttest     3.21
 2 2     Kontrollgruppe posttest     5.99
 3 3     Kontrollgruppe posttest     4.17
 4 4     Kontrollgruppe posttest     4.76
 5 5     Kontrollgruppe posttest     5.64
 6 6     Kontrollgruppe posttest     4.67
 7 7     Kontrollgruppe posttest     4.24
 8 8     Kontrollgruppe posttest     3.34
 9 9     Kontrollgruppe posttest     4.11
10 10    Kontrollgruppe posttest     5.29
# i 90 more rows
```

```
# nur pretest
therapie_long |>
  filter(zeit != "posttest")
```

```
# A tibble: 100 x 4
   id    gruppe         zeit    symptome
   <fct> <fct>          <fct>      <dbl>
 1 1     Kontrollgruppe pretest     4.29
 2 2     Kontrollgruppe pretest     6.18
 3 3     Kontrollgruppe pretest     3.93
 4 4     Kontrollgruppe pretest     5.06
 5 5     Kontrollgruppe pretest     6.45
 6 6     Kontrollgruppe pretest     4.49
 7 7     Kontrollgruppe pretest     4.60
 8 8     Kontrollgruppe pretest     4.46
 9 9     Kontrollgruppe pretest     4.76
10 10    Kontrollgruppe pretest     5.12
# i 90 more rows
```

```r
# nur symptome >= 5
therapie_long |>
  filter(symptome >= 5)
```

```
# A tibble: 66 x 4
   id    gruppe          zeit     symptome
   <fct> <fct>           <fct>       <dbl>
 1 2     Kontrollgruppe pretest      6.18
 2 2     Kontrollgruppe posttest     5.99
 3 4     Kontrollgruppe pretest      5.06
 4 5     Kontrollgruppe pretest      6.45
 5 5     Kontrollgruppe posttest     5.64
 6 10    Kontrollgruppe pretest      5.12
 7 10    Kontrollgruppe posttest     5.29
 8 11    Kontrollgruppe pretest      6.04
 9 13    Kontrollgruppe posttest     5.16
10 19    Kontrollgruppe pretest      5.20
# i 56 more rows
```

```r
# nur symptome mit Werten zwischen 3 und 5
therapie_long |>
  filter(symptome >= 3 & symptome <= 5)
```

```
# A tibble: 130 x 4
   id    gruppe          zeit     symptome
   <fct> <fct>           <fct>       <dbl>
 1 1     Kontrollgruppe pretest      4.29
 2 1     Kontrollgruppe posttest     3.21
 3 3     Kontrollgruppe pretest      3.93
 4 3     Kontrollgruppe posttest     4.17
 5 4     Kontrollgruppe posttest     4.76
 6 6     Kontrollgruppe pretest      4.49
 7 6     Kontrollgruppe posttest     4.67
 8 7     Kontrollgruppe pretest      4.60
 9 7     Kontrollgruppe posttest     4.24
10 8     Kontrollgruppe pretest      4.46
# i 120 more rows
```

```r
# nur Personen mit id 3 und 5
therapie_long |>
  filter(id == 3 | id == 5)
```

```
# A tibble: 4 x 4
  id    gruppe          zeit     symptome
  <fct> <fct>           <fct>       <dbl>
1 3     Kontrollgruppe pretest      3.93
2 3     Kontrollgruppe posttest     4.17
3 5     Kontrollgruppe pretest      6.45
4 5     Kontrollgruppe posttest     5.64
```

```
# Alternative dazu (mit dem %in% Operator)
therapie_long |>
  filter(id %in% c(3, 5))
```

```
# A tibble: 4 x 4
  id    gruppe          zeit     symptome
  <fct> <fct>           <fct>       <dbl>
1 3     Kontrollgruppe pretest      3.93
2 3     Kontrollgruppe posttest     4.17
3 5     Kontrollgruppe pretest      6.45
4 5     Kontrollgruppe posttest     5.64
```

4.4.5 Beobachtungen (Fälle) sortieren mit `arrange()`

Mit der `arrange()` Funktion können wir Beobachtungen sortieren, entweder in aufsteigender oder in absteigender Reihenfolge.

```
# aufsteigend
therapie_long |>
  arrange(id)
```

```
# A tibble: 200 x 4
   id    gruppe          zeit     symptome
   <fct> <fct>           <fct>       <dbl>
 1 1     Kontrollgruppe pretest      4.29
 2 1     Kontrollgruppe posttest     3.21
 3 2     Kontrollgruppe pretest      6.18
 4 2     Kontrollgruppe posttest     5.99
 5 3     Kontrollgruppe pretest      3.93
 6 3     Kontrollgruppe posttest     4.17
 7 4     Kontrollgruppe pretest      5.06
 8 4     Kontrollgruppe posttest     4.76
 9 5     Kontrollgruppe pretest      6.45
```

```
10 5     Kontrollgruppe posttest     5.64
# i 190 more rows
```

```r
# absteigend
therapie_long |>
  arrange(desc(id))
```

```
# A tibble: 200 x 4
   id    gruppe         zeit     symptome
   <fct> <fct>          <fct>       <dbl>
 1 100   Therapiegruppe pretest      4.77
 2 100   Therapiegruppe posttest     4.50
 3 99    Therapiegruppe pretest      4.66
 4 99    Therapiegruppe posttest     3.80
 5 98    Therapiegruppe pretest      4.36
 6 98    Therapiegruppe posttest     3.91
 7 97    Therapiegruppe pretest      4.53
 8 97    Therapiegruppe posttest     4.36
 9 96    Therapiegruppe pretest      5.15
10 96    Therapiegruppe posttest     5.05
# i 190 more rows
```

4.4.6 Neue Variablen erstellen mit `mutate()`

Neue Variablen können mit `mutate()` aus schon bestehenden Variablen gebildet werden.

Syntax:

```r
df |> mutate(neue_variable_1 = FORMEL_1,
             neue_variable_2 = FORMEL_2)
```

Beispiele

In Fragebogenstudien kommt es häufig vor, dass man für jede Person aus mehreren Variablen/Items einen Mittelwert berechnen will, z.B. um einige der Selbstwirksamkeitsitems aus dem Beispieldatensatz zu einer Skala zusammenzufassen:

```r
# Neue Variable mit dem Mittelwert der Items swk4 bis swk7 erstellen
beispieldaten |>
  select(num_range("swk", 4:7)) |>
  mutate(swk_mean = (swk4 + swk5 + swk6 + swk7) / 4)
```

```
# A tibble: 286 x 5
    swk4  swk5  swk6  swk7 swk_mean
   <dbl> <dbl> <dbl> <dbl>    <dbl>
 1     4     7     5     5     5.25
 2     4     6     5     5     5
 3     6     5     7     7     6.25
 4     5     7     7     6     6.25
 5     5     7     6     5     5.75
 6     4     5     7     5     5.25
 7     4     6     7     1     4.5
 8     7     7     7    NA    NA
 9     5     7     6     5     5.75
10     5     5     5     5     5
# i 276 more rows
```

An diesem Beispiel wird ersichtlich, dass die Berechnung des Mittelwerts über eine Formel nicht funktioniert, wenn `NA`s (fehlende Werte) vorhanden sind. Da in Zeile 8 kein Wert für `swk7` vorhanden ist, kann in dieser Zeile kein Mittelwert berechnet werden.

Die Verwendung von `mean()` mit dem Argument `na.rm = TRUE` ist auch nicht möglich, da `mean()` dann den Mittelwert über alle Personen UND alle Variablen berechnet (und nicht pro Person den Mittelwert über die Variablen).

```
beispieldaten |>
  select(num_range("swk", 4:7)) |>
  mutate(swk_mean = mean(c(swk4, swk5, swk6, swk7), na.rm = TRUE))
```

```
# A tibble: 286 x 5
    swk4  swk5  swk6  swk7 swk_mean
   <dbl> <dbl> <dbl> <dbl>    <dbl>
 1     4     7     5     5     5.58
 2     4     6     5     5     5.58
 3     6     5     7     7     5.58
 4     5     7     7     6     5.58
 5     5     7     6     5     5.58
 6     4     5     7     5     5.58
 7     4     6     7     1     5.58
 8     7     7     7    NA     5.58
 9     5     7     6     5     5.58
10     5     5     5     5     5.58
# i 276 more rows
```

Die Funktion `rowwise()` aus dem `dplyr`-Package kann uns hier weiterhelfen.

```
beispieldaten |>
  select(num_range("swk", 4:7)) |>
  rowwise() |>
  mutate(swk_mean = mean(c(swk4, swk5, swk6, swk7), na.rm = TRUE))
```

```
# A tibble: 286 x 5
# Rowwise:
    swk4  swk5  swk6  swk7 swk_mean
   <dbl> <dbl> <dbl> <dbl>    <dbl>
 1     4     7     5     5     5.25
 2     4     6     5     5     5
 3     6     5     7     7     6.25
 4     5     7     7     6     6.25
 5     5     7     6     5     5.75
 6     4     5     7     5     5.25
 7     4     6     7     1     4.5
 8     7     7     7    NA     7
 9     5     7     6     5     5.75
10     5     5     5     5     5
# i 276 more rows
```

Die Verwendung von `rowwise()` führt dazu, dass `mean(na.rm = TRUE)` den Mittelwert für jede Zeile berechnet. In Zeile 8 wurde nun also der Mittelwert über die verbliebenen 3 Items berechnet.

Mit `mutate()` können wir auch eine bestehende Variable umberechnen. In folgendem Beispiel rechnen wir die Symptom-Werte (Skala von 0 bis 7) in Prozentwerte um.

```
therapie_long |>
  mutate(symptome_p = round(symptome / 7 * 100, digits = 1))
```

```
# A tibble: 200 x 5
   id    gruppe         zeit     symptome symptome_p
   <fct> <fct>          <fct>       <dbl>      <dbl>
 1 1     Kontrollgruppe pretest      4.29       61.2
 2 1     Kontrollgruppe posttest     3.21       45.8
 3 2     Kontrollgruppe pretest      6.18       88.2
 4 2     Kontrollgruppe posttest     5.99       85.5
 5 3     Kontrollgruppe pretest      3.93       56.2
 6 3     Kontrollgruppe posttest     4.17       59.6
```

```
 7 4     Kontrollgruppe pretest      5.06       72.3
 8 4     Kontrollgruppe posttest     4.76       68.1
 9 5     Kontrollgruppe pretest      6.45       92.2
10 5     Kontrollgruppe posttest     5.64       80.6
# i 190 more rows
```

Mit dem Argument `.keep = c("all", "used", "unused", "none")` kann kontrolliert werden, welche Variablen im Output erscheinen. `.keep = "used"` führt z.B. dazu, dass neben der neu gebildeten Variable nur diejenigen Variablen, die für die Erstellung der neuen Variable verwendet wurden, im Output erscheinen.

> Hinweis
>
> Im obigen Beispiel mit `rowwise()` haben wir zunächst mit `select()` nur die Variablen ausgewählt, aus denen wir anschliessend die neue Variable gebildet haben. Das `mutate()`-Argument `.keep = "used"` erfüllt dieselbe Funktion und macht den Code kürzer.

```
therapie_long |>
  mutate(symptome_p = round(symptome / 7 * 100, digits = 1),
         .keep = "used")
```

```
# A tibble: 200 x 2
   symptome symptome_p
      <dbl>      <dbl>
 1     4.29       61.2
 2     3.21       45.8
 3     6.18       88.2
 4     5.99       85.5
 5     3.93       56.2
 6     4.17       59.6
 7     5.06       72.3
 8     4.76       68.1
 9     6.45       92.2
10     5.64       80.6
# i 190 more rows
```

`.keep = "none"` führt dazu, dass nur die neue Variable im Output erscheint.

```
therapie_long |>
  mutate(symptome_p = round(symptome / 7 * 100, digits = 1),
         .keep = "none")
```

```
# A tibble: 200 x 1
   symptome_p
        <dbl>
 1       61.2
 2       45.8
 3       88.2
 4       85.5
 5       56.2
 6       59.6
 7       72.3
 8       68.1
 9       92.2
10       80.6
# i 190 more rows
```

`across()`: Mehrere Variablen auswählen und verändern

Möchten wir Transformationen auf mehrere Variablen gleichzeitig anwenden, können wir `mutate()` mit `across()` verwenden.

Syntax:

```
across(.cols, .fns, .names)
```

Das erste Argument `.cols` wählt die Variablen aus, mit denen gearbeitet wird. Das zweite Argument `.fns` bestimmt eine oder mehrere Funktionen, die auf die ausgewählten Variablen angewendet wird/werden. Mit dem dritten Argument `.names` kann spezifiziert werden, wie die Variablen im Output benannt werden sollen.

Als Beispiel wollen wir die Symptom-Werte im Datensatz `therapie_wide` in Prozentwerte umrechnen. Diese Werte befinden sich in den Spalten `pretest` and `posttest`. Da diese die einzigen numerischen Variablen im Datensatz sind, können wir zur Auswahl dieser Variablen die Bedingung `where(is.numeric)` benutzen, die alle numerischen Variablen eines Data Frames auswählt.

Oben haben wir der in `mutate()` zu bildenden Variable einfach einen Namen gegeben und anschliessend die Formel bzw. Funktion angegeben, mit der die neue Variable erstellt werden soll.

Bei der Bildung/Transformation von mehr als einer Variablen muss zur Definition der Formel/Funktion eine *anonymous function* erstellt werden, d.h. es muss explizit eine Funktion definiert werden, die auf die ausgewählten Variablen angewendet wird.

> Hinweis
>
> Das Erstellen eigener Funktionen in `R` ist über diese Anwendung hinaus nicht Gegenstand dieses Kurses. Sie sehen aber hier schon, dass es nicht allzu schwierig ist, eigene Funktionen in `R` zu definieren.

Eine *anonymous function* startet mit `function(x)` gefolgt von der Definition der Funktion (in diesem Fall eine Kombination von `round()` und der Formel zur Umrechnung in Prozentwerte), wobei `x` (das Argument der zu bildenden Funktion) für die jeweils einzufügende Variable steht:

```
therapie_wide |>
  mutate(across(.cols = where(is.numeric),
                .fns = function(x) round(x / 7 * 100, digits = 1)))
```

```
# A tibble: 100 x 4
   id    gruppe         pretest posttest
   <fct> <fct>            <dbl>    <dbl>
 1 1     Kontrollgruppe    61.2     45.8
 2 2     Kontrollgruppe    88.2     85.5
 3 3     Kontrollgruppe    56.2     59.6
 4 4     Kontrollgruppe    72.3     68.1
 5 5     Kontrollgruppe    92.2     80.6
 6 6     Kontrollgruppe    64.1     66.7
 7 7     Kontrollgruppe    65.7     60.5
 8 8     Kontrollgruppe    63.7     47.8
 9 9     Kontrollgruppe    68       58.7
10 10    Kontrollgruppe    73.1     75.6
# i 90 more rows
```

Hier werden die Variablen `pretest` und `posttest` direkt verändert/überschrieben.

Mit dem `.names`-Argument können wir dafür sorgen, dass die transformierten Variablen unter einem neuen Namen dem Datensatz hinzugefügt werden (und die Ausgangsvariablen unverändert im Datensatz verbleiben).

```
therapie_wide |>
  mutate(across(.cols = where(is.numeric),
                .fns = function(x) round(x / 7 * 100, digits = 1),
                .names = "{.col}_percent"))
```

```
# A tibble: 100 x 6
```

```
   id    gruppe         pretest posttest pretest_percent posttest_percent
   <fct> <fct>            <dbl>    <dbl>           <dbl>            <dbl>
 1 1     Kontrollgruppe    4.29     3.21            61.2             45.8
 2 2     Kontrollgruppe    6.18     5.99            88.2             85.5
 3 3     Kontrollgruppe    3.93     4.17            56.2             59.6
 4 4     Kontrollgruppe    5.06     4.76            72.3             68.1
 5 5     Kontrollgruppe    6.45     5.64            92.2             80.6
 6 6     Kontrollgruppe    4.49     4.67            64.1             66.7
 7 7     Kontrollgruppe    4.60     4.24            65.7             60.5
 8 8     Kontrollgruppe    4.46     3.34            63.7             47.8
 9 9     Kontrollgruppe    4.76     4.11            68               58.7
10 10    Kontrollgruppe    5.12     5.29            73.1             75.6
# i 90 more rows
```

In einer Kurzvariante zur Definition einer *anonymous function* wird das Wort `function` durch das Zeichen \ (Backslash) ersetzt. Wir benutzen im Folgenden diese Variante.

Ausserdem können wir die Variablen auch mithilfe eines Vektors von Variablennamen (hier `c(pretest, posttest)`) auswählen, ohne eine Bedingung wie `where(is.numeric)` zu verwenden.

Zuletzt müssen die ersten beiden Argumente von `across()` nicht unbedingt benannt werden (wenn sie in der richtigen Reihenfolge verwendet werden), was den Code noch etwas verkürzt:

```
therapie_wide |>
  mutate(across(c(pretest, posttest), \(x) round(x / 7 * 100, digits = 1),
                .names = "{.col}_percent"))
```

```
# A tibble: 100 x 6
   id    gruppe         pretest posttest pretest_percent posttest_percent
   <fct> <fct>            <dbl>    <dbl>           <dbl>            <dbl>
 1 1     Kontrollgruppe    4.29     3.21            61.2             45.8
 2 2     Kontrollgruppe    6.18     5.99            88.2             85.5
 3 3     Kontrollgruppe    3.93     4.17            56.2             59.6
 4 4     Kontrollgruppe    5.06     4.76            72.3             68.1
 5 5     Kontrollgruppe    6.45     5.64            92.2             80.6
 6 6     Kontrollgruppe    4.49     4.67            64.1             66.7
 7 7     Kontrollgruppe    4.60     4.24            65.7             60.5
 8 8     Kontrollgruppe    4.46     3.34            63.7             47.8
 9 9     Kontrollgruppe    4.76     4.11            68               58.7
10 10    Kontrollgruppe    5.12     5.29            73.1             75.6
# i 90 more rows
```

Falls die Funktion, die wir auf mehrere Variablen/Spalten anwenden möchten, kein weiteres Argument hat, kann Sie auch ohne Definition einer *anonymous function* direkt verwendet werden, dann allerdings ohne Funktionsklammern. Z.B. können wir alle als *labelled double* eingelesenen Gruppierungsvariablen im Beispieldatensatz mit der Funktion `haven::as_factor()` zu Faktoren konvertieren und gleichzeitig die eingelesenen Labels zu Faktorstufen machen (vgl. Kap. 3.2).

Für diesen Zweck wurde im Package `haven` extra eine Funktion namens `is.labelled()` definiert, mit der wir (zusammen mit `across(where())`) die als *labelled double* eingelesenen Variablen identifizieren und auswählen können:

```
beispieldaten |>
  mutate(across(where(is.labelled), haven::as_factor),
         .keep = "used")
```

```
# A tibble: 286 x 6
   westost geschlecht bildung_vater          bildung_mutter bildung_vater_binaer
   <fct>   <fct>      <fct>                  <fct>          <fct>
 1 West    weiblich   Fachhochschulabschlus~ Fachhochschul~ hoch
 2 West    männlich   Fachabitur, Abitur     Realschulabsc~ hoch
 3 West    weiblich   Realschulabschluss (m~ Hauptschulabs~ niedrig
 4 West    weiblich   Realschulabschluss (m~ Fachhochschul~ niedrig
 5 West    weiblich   Fachhochschulabschlus~ Realschulabsc~ hoch
 6 West    männlich   Realschulabschluss (m~ Realschulabsc~ niedrig
 7 West    männlich   Realschulabschluss (m~ Realschulabsc~ niedrig
 8 West    weiblich   Fachhochschulabschlus~ Fachhochschul~ hoch
 9 West    männlich   Realschulabschluss (m~ Realschulabsc~ niedrig
10 West    männlich   Fachabitur, Abitur     Realschulabsc~ hoch
# i 276 more rows
# i 1 more variable: bildung_mutter_binaer <fct>
```

4.4.7 Variablen rekodieren mit `case_when()`

Manchmal möchte man Variablen nach bestimmten Bedingungen umkodieren, die neue Variable soll also bestimmte Werte aufweisen, je nachdem welche Eigenschaften die Werte der alten Variable haben.

Syntax:

```
case_when(bedingung_1 ~ "neuer_wert_1",
          bedingung_2 ~ "neuer_wert_2")
```

Da man hier Variablen nach bestimmten Bedingungen verändert bzw. neu bildet, wird die Funktion innerhalb von `mutate()` verwendet, es handelt sich also um eine Art Hilfsfunktion für `mutate()`.

Syntax mit `mutate()`:

```
mutate(neue_variable = case_when(alte_variable_bedingung_1 ~ "neuer_wert_1",
                                 alte_variable_bedingung_2 ~ "neuer_wert_2")
```

Beispiel

Wir wollen das `alter` der Jugendlichen im Beispieldatensatz so rekodieren, dass diejenigen, die älter als der Mittelwert sind, den Wert "älter" bekommen, und diejenigen, die jünger als der Mittelwert sind, den Wert "jünger" bekommen.

```
beispieldaten |>
  mutate(alter_dichotom = case_when(alter > mean(alter) ~ "älter",
                                    alter < mean(alter) ~ "jünger"),
         .keep = "used")
```

```
# A tibble: 286 x 2
   alter alter_dichotom
   <dbl> <chr>
 1    13 jünger
 2    14 jünger
 3    14 jünger
 4    14 jünger
 5    14 jünger
 6    14 jünger
 7    14 jünger
 8    15 älter
 9    15 älter
10    14 jünger
# i 276 more rows
```

Diese Variable können wir z.B. anschliessend nutzen, um herauszufinden, wie die Altersverteilung in dieser Hinsicht nach Geschlecht aussieht. Dazu müssen wir die Daten aber zuerst wieder zuweisen (hier zu `dat_table`):

```r
dat_table <- beispieldaten |>
  select(alter, geschlecht) |>
  drop_na() |>
  mutate(geschlecht = haven::as_factor(geschlecht),
         alter_dichotom = case_when(alter > mean(alter) ~ "älter",
                                    alter < mean(alter) ~ "jünger"))

table(dat_table$geschlecht, dat_table$alter_dichotom)
```

```
           älter jünger
  männlich    93     59
  weiblich    70     64
```

```r
# als Anteile pro Geschlecht (Zeilen der Kreuztabelle)
table(dat_table$geschlecht, dat_table$alter_dichotom) |>
  proportions(1) |>
  round(2)
```

```
           älter jünger
  männlich  0.61   0.39
  weiblich  0.52   0.48
```

4.4.8 Faktorstufen rekodieren mit `fct_recode()`

Um die Stufen eines Faktors umzukodieren, verwenden wir `fct_recode()` aus dem `forcats`-Package.

Syntax:

```r
fct_recode(variable,
           neuer_wert_1 = "alter_wert_1",
           neuer_wert_2 = "alter_wert_2")
```

Auch diese Funktion wird normalerweise innerhalb von `mutate()` verwendet.

Beispiel

```r
# Kontrollgruppe in "control" und die Therapiegruppe in "treatment" umbenennen
therapie_long |>
  mutate(gruppe = fct_recode(gruppe,
                             control = "Kontrollgruppe",
                             treatment = "Therapiegruppe"))
```

```
# A tibble: 200 x 4
   id    gruppe  zeit     symptome
   <fct> <fct>   <fct>       <dbl>
 1 1     control pretest      4.29
 2 1     control posttest     3.21
 3 2     control pretest      6.18
 4 2     control posttest     5.99
 5 3     control pretest      3.93
 6 3     control posttest     4.17
 7 4     control pretest      5.06
 8 4     control posttest     4.76
 9 5     control pretest      6.45
10 5     control posttest     5.64
# i 190 more rows
```

Hier wird mit `mutate()` keine neue Variable gebildet, sondern es wird eine Variable verändert, indem die durch `fct_recode()` veränderte Variable `gruppe` der ursprünglichen Variable wiederzugewiesen wird.

Wir können Faktorstufen nicht nur umbenennen, sondern sie auch "zusammenlegen", um einen Faktor mit weniger Faktorstufen zu erhalten. Das macht man manchmal, wenn manche Kategorien nur sehr geringe Häufigkeiten haben. Z.B. können wir die Altersvariable zunächst als Faktor definieren (es gibt dort nur ganzzahlige Altersangaben):

```r
# Alter als Faktorvariable definieren
beispieldaten <- beispieldaten |>
  mutate(alter_f = factor(alter))

table(beispieldaten$alter_f)
```

```
 13  14  15  16  17
  6 117 121  40   2
```

Es gibt nur sehr wenige 13-Jährige sowie 17-Jährige im Datensatz. Man könnte daher eine dreistufige Altersvariable bilden, bei der diese Alterskategorien mit der jeweils benachbarten zusammengelegt werden:

```
beispieldaten <- beispieldaten |>
  mutate(alter_dreistufig = fct_recode(alter_f,
                               alt = "17",
                               alt = "16",
                               mittel = "15",
                               jung = "14",
                               jung = "13"))

table(beispieldaten$alter_dreistufig)
```

```
  jung mittel    alt
   123    121     42
```

4.4.9 Daten gruppieren mit `group_by()`

Oft wollen wir bestimmte Operationen nicht auf den ganzen Datensatz anwenden, sondern nur auf Subgruppen, welche durch Faktorstufen definiert sind. Dafür gibt es die Funktion `group_by()` - diese teilt den Datensatz anhand einer Gruppierungsvariable auf, wendet eine Funktion auf jeden Teil/jede Gruppe an, und setzt den Datensatz danach wieder zusammen (split-apply-combine). `group_by()` wird deshalb meistens in Kombination mit anderen Funktionen verwendet.

> Hinweis
>
> Wollen wir die Gruppierung nach einer abgeschlossenen gruppierten Operation wieder rückgängig machen (um weitere ungruppierte Operationen durchzuführen), so verwenden wir `ungroup()`. Falls man das Ergebnis einem Objekt zuweist, ist es ratsam, am Schluss der Operation immer `ungroup()` zu verwenden, um zu verhindern, dass die Gruppierungsinformation in dem Objekt mit abgespeichert wird.

Syntax:

```
df <- group_by(gruppierung_1, gruppierung_2, gruppierung_3)
```

Beispiele

Lassen Sie uns einen neuen Datensatz importieren. In diesem Beispiel wurden 12 Probanden vier Versuchsbedingungen zugeordnet, und als abhängige Variable wurde das Ausmass des aggressiven Verhaltens gemessen. Nehmen wir nun an, dass wir dem Datensatz eine gruppenzentrierte Aggressionsvariable hinzufügen wollen (die Gruppen sind die Bedingungen). Solche gruppenzentrierten Variablen (jede Person hat als Wert die Abweichung vom Gruppenmittelwert der Gruppe, der sie zugehört) werden für bestimmte statistische Verfahren benötigt, die wir in den nächsten Semestern kennenlernen werden.

```
library(readr)
alk_aggr <- read_csv("data/alkohol-aggression.csv", show_col_types = FALSE)
alk_aggr$alkoholbedingung <- factor(alk_aggr$alkoholbedingung)

alk_aggr
```

```
# A tibble: 12 x 2
   aggressivitaet alkoholbedingung
            <dbl> <fct>
 1             64 kein_alkohol
 2             58 kein_alkohol
 3             64 kein_alkohol
 4             74 placebo
 5             79 placebo
 6             72 placebo
 7             71 anti_placebo
 8             69 anti_placebo
 9             67 anti_placebo
10             69 alkohol
11             73 alkohol
12             74 alkohol
```

Wir können `group_by()` verwenden, um den Datensatz in vier separate Teile zu teilen, berechnen dann den Gruppenmittelwert für jeden einzelnen, und verwenden diesen gleich, um die gruppenzentrierte Variable zu berechnen. Anschliessend werden die Teile wieder "zusammengesetzt" (d.h. es ist wieder *ein* Datensatz). Da wir das ganze wieder zuweisen, benutzen wir am Ende `ungroup()`.

```
alk_aggr <- alk_aggr |>
  group_by(alkoholbedingung) |>
  mutate(group_mean = mean(aggressivitaet),
```

```
         aggr_c = aggressivitaet - group_mean) |>
  ungroup()

alk_aggr
```

```
# A tibble: 12 x 4
   aggressivitaet alkoholbedingung group_mean aggr_c
            <dbl> <fct>                 <dbl>  <dbl>
 1             64 kein_alkohol             62      2
 2             58 kein_alkohol             62     -4
 3             64 kein_alkohol             62      2
 4             74 placebo                  75     -1
 5             79 placebo                  75      4
 6             72 placebo                  75     -3
 7             71 anti_placebo             69      2
 8             69 anti_placebo             69      0
 9             67 anti_placebo             69     -2
10             69 alkohol                  72     -3
11             73 alkohol                  72      1
12             74 alkohol                  72      2
```

4.4.10 Variablen zusammenfassen mit `summarize()`

Mit `summarize()` oder `summarise()` können wir Variablen zusammenfassen und deskriptive Kennzahlen berechnen. `summarize()` wird oft zusammen mit `group_by()` verwendet.

> Hinweis
>
> Im Gegensatz zu `mutate()` gibt `summarize()` nicht einen Wert für jede Beobachtung als Output, sondern einen Wert (z.B. eine deskriptive Statistik) für jede durch `group_by()` definierte Gruppe (bzw. für die Gesamtstichprobe, falls `group_by()` nicht verwendet wird).

Syntax:

```
df |> summarize(kennzahl = FUNKTION(variable))
```

`FUNKTION` ist ein Platzhalter für jede Funktion, die zum Zusammenfassen von Daten verwendet werden kann, z.B. `mean()` oder `sd()`.

Als Beispiel mit dem `therapie_long` Datensatz berechnen wir die Mittelwerte und die Standardabweichungen der Variable `symptome` getrennt für alle Gruppen und Messzeitpunkte:

```
# Gruppenmittelwerte pro Messzeitpunkt
therapie_long |>
  group_by(gruppe, zeit) |>
  summarize(Mittelwert = mean(symptome),
            Standardabweichung = sd(symptome))
```

```
`summarise()` has grouped output by 'gruppe'. You can override using the
`.groups` argument.
```

```
# A tibble: 4 x 4
# Groups:   gruppe [2]
  gruppe         zeit     Mittelwert Standardabweichung
  <fct>          <fct>         <dbl>              <dbl>
1 Kontrollgruppe pretest        5.06              0.908
2 Kontrollgruppe posttest       4.65              0.834
3 Therapiegruppe pretest        4.82              0.691
4 Therapiegruppe posttest       4.23              0.752
```

Statt wie oben `summarize()` nur auf eine Variable anzuwenden, können wir diese Funktion mit `across()` auch auf mehrere Variablen anwenden. Z.B. möchten wir die Mittelwerte der Items `swk4` bis `swk7` berechnen. Dazu wird die Funktion `mean()` wieder *ohne Funktionsklammern* benutzt, da sie innerhalb von `across()` ohne weiteres Argument verwendet wird:

```
beispieldaten |>
  summarize(across(swk4:swk7, mean))
```

```
# A tibble: 1 x 4
   swk4  swk5  swk6  swk7
  <dbl> <dbl> <dbl> <dbl>
1    NA    NA    NA    NA
```

Es scheint `NA`s zu geben, daher funktioniert `mean()` nicht ohne das Argument `na.rm = TRUE`. Damit wir dieses Argument aber innerhalb von `across()` aufrufen können, müssen wir wieder eine *anonymous function* definieren:

```
beispieldaten |>
  summarize(across(swk4:swk7, \(x) mean(x, na.rm = TRUE)))
```

```
# A tibble: 1 x 4
   swk4  swk5  swk6  swk7
  <dbl> <dbl> <dbl> <dbl>
1  5.01  6.17  6.16  4.97
```

Oder wir löschen die `NA`s zuerst:

```
beispieldaten |>
  select(swk4:swk7) |>
  drop_na() |>
  summarize(across(everything(), mean))
```

```
# A tibble: 1 x 4
   swk4  swk5  swk6  swk7
  <dbl> <dbl> <dbl> <dbl>
1  5.01  6.17  6.17  4.96
```

Warum unterscheiden sich die Ergebnisse minimal? - Weil oben die `NA`s pro Variable entfernt wurden, während hier alle Personen gelöscht wurden, die einen fehlenden Wert auf irgendeiner der vier Variablen haben.

Vertiefung

Es ist auch möglich, mehrere Funktionen gleichzeitig auf mehrere Variablen anzuwenden. Dann muss unter `.fns` eine `list()` von *anonymous functions* erstellt werden. Z.B. können wir zusätzlich zum Mittelwert noch die Standardabweichung berechnen:

```
beispieldaten |>
  summarize(across(.cols = swk4:swk7,
                   .fns = list(
                     mean = \(x) mean(x, na.rm = TRUE),
                     sd = \(x) sd(x, na.rm = TRUE)),
                   .names = NULL))
```

```
# A tibble: 1 x 8
  swk4_mean swk4_sd swk5_mean swk5_sd swk6_mean swk6_sd swk7_mean swk7_sd
      <dbl>   <dbl>     <dbl>   <dbl>     <dbl>   <dbl>     <dbl>   <dbl>
1      5.01    1.14      6.17    1.01      6.16   0.918      4.97    1.50
```

Ohne Argumentwert (`.names = NULL`) wird die Benennung der Output-Variablen nach dem Default *Variablenname_Funktionsname* gemacht. Übersetzt heisst das: `.names = "{.col}_{.fn}"`

Wenn wir es andersherum haben wollen, können wir das `.names`-Argument entsprechend ändern:

```
beispieldaten |>
  summarize(across(.cols = swk4:swk7,
                   .fns = list(
                     mean = \(x) mean(x, na.rm = TRUE),
                     sd = \(x) sd(x, na.rm = TRUE)),
                   .names = "{.fn}_{.col}"))
```

```
# A tibble: 1 x 8
  mean_swk4 sd_swk4 mean_swk5 sd_swk5 mean_swk6 sd_swk6 mean_swk7 sd_swk7
      <dbl>   <dbl>     <dbl>   <dbl>     <dbl>   <dbl>     <dbl>   <dbl>
1      5.01    1.14      6.17    1.01      6.16   0.918      4.97    1.50
```

Um den Output bei der Anwendung mehrerer Funktionen übersichtlicher zu machen, kann man im Anschluss `pivot_longer()` mit fortgeschrittenen Argumenten verwenden, die wir oben nicht betrachtet haben. Um das noch besser zu veranschaulichen, berechnen wir jetzt zusätzlich zu Mittelwert und Standardabweichung auch noch die Anzahl der fehlenden Werte (`nmiss`) mit `sum(is.na(x))`:

```
beispieldaten |>
  summarize(across(.cols = swk4:swk7,
                   .fns = list(
                     mean = \(x) mean(x, na.rm = TRUE),
                     sd = \(x) sd(x, na.rm = TRUE),
                     nmiss = \(x) sum(is.na(x))),
                   .names = "{.fn}_{.col}"
                   )
            ) |>
  pivot_longer(everything(),
               names_to = c("source", ".value"),
               names_sep = "_")
```

```
# A tibble: 3 x 5
  source  swk4  swk5  swk6  swk7
  <chr>  <dbl> <dbl> <dbl> <dbl>
1 mean    5.01  6.17 6.16   4.97
2 sd      1.14  1.01 0.918  1.50
3 nmiss   1     2    1      2
```

4.5 Übungsaufgaben

Jugendliche in West-/Ostdeutschland

```
library(haven)
beispieldaten <- read_sav("data/beispieldaten.sav")
beispieldaten$westost <- haven::as_factor(beispieldaten$westost)
beispieldaten$geschlecht <- haven::as_factor(beispieldaten$geschlecht)

head(beispieldaten)
```

```
# A tibble: 6 x 98
     ID westost geschlecht alter  swk1  swk2  swk3  swk4  swk5  swk6  swk7  swk8
  <dbl> <fct>   <fct>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1     1 West    weiblich      13     2     5     6     4     7     5     5     6
2     2 West    männlich      14     4     6     4     4     6     5     5     6
3    10 West    weiblich      14     5     4     5     6     5     7     7     6
4    11 West    weiblich      14     3     6     6     5     7     7     6     6
5    12 West    weiblich      14     5     6     6     5     7     6     5     5
6    14 West    männlich      14     3     5     5     4     5     7     5     6
# i 86 more variables: swk9 <dbl>, swk10 <dbl>, swk11 <dbl>, swk12 <dbl>,
#   swk13 <dbl>, swk14 <dbl>, swk15 <dbl>, swk16 <dbl>, swk17 <dbl>,
#   swk18 <dbl>, swk19 <dbl>, swk20 <dbl>, swk21 <dbl>, swk22 <dbl>,
#   swk23 <dbl>, swk24 <dbl>, swk25 <dbl>, swk26 <dbl>, swk27 <dbl>,
#   swk28 <dbl>, swk29 <dbl>, swk30 <dbl>, swk31 <dbl>, swk32 <dbl>,
#   swk33 <dbl>, swk34 <dbl>, swk35 <dbl>, swk36 <dbl>, unteltern1 <dbl>,
#   unteltern2 <dbl>, unteltern3 <dbl>, unteltern4 <dbl>, unteltern5 <dbl>, ...
```

Durchschnittsalter

Berechnen Sie das mittlere Alter und die Standardabweichung für beide Geschlechter.

Lösung

```
beispieldaten |>
  group_by(geschlecht) |>
  summarize(alter_mw = round(mean(alter), 1),
            alter_sd = round(sd(alter), 1))

# A tibble: 2 x 3
```

```
  geschlecht alter_mw alter_sd
  <fct>         <dbl>    <dbl>
1 männlich       14.8      0.7
2 weiblich       14.6      0.8
```

Unterstützung durch Freunde und Eltern

Wählen Sie die Variablen `unt_freunde` (Unterstützung durch Freunde) und `unt_eltern` (Unterstützung durch Eltern) aus, und machen sie daraus einen *long* Datensatz. Beide Variablen wurden mit denselben Items gemessen, sie unterscheiden sich lediglich durch die *Quelle* der Unterstützung (Freunde oder Eltern). Insofern können wir diese Unterstützungsquelle als messwiederholten Faktor auffassen.

Lösung

Schritt 1:

```
# Variablen auswählen
unterstuetzung <- beispieldaten |>
  select(ID, westost, unt_freunde, unt_eltern)
```

Schritt 2:

```
# gibt es fehlende Werte?
unterstuetzung[!complete.cases(unterstuetzung), ]
```

```
# A tibble: 2 x 4
     ID westost unt_freunde unt_eltern
  <dbl> <fct>         <dbl>      <dbl>
1    16 West           NA         5.17
2   332 Ost            4.83      NA
```

```
# fehlende Werte ausschliessen
unterstuetzung <- unterstuetzung |>
  drop_na()
```

Schritt 3:

```
# von wide zu long: wir nennen den Messwiederholungsfaktor "quelle" und die
# Variable mit den Werten "unterstuetzung"
library(tidyr)

unterstuetzung <- unterstuetzung |>
  pivot_longer(!c(ID, westost), names_to = "quelle",
               values_to = "unterstuetzung")
```

Schritt 4:

```
# quelle zu Faktor konvertieren
unterstuetzung <- unterstuetzung |>
  mutate(quelle = factor(quelle))
```

Schritt 5 (Fortgeschritten):

```
# Stufen umbennen ("unt_" entfernen)
library(stringr)
unterstuetzung <- unterstuetzung |>
  mutate(quelle = str_replace(quelle, ".*_", ""))
```

Schritt 6:

```
# Data Frame anschauen
unterstuetzung
```

```
# A tibble: 568 x 4
      ID westost quelle  unterstuetzung
   <dbl> <fct>   <chr>            <dbl>
 1     1 West    freunde           6.67
 2     1 West    eltern            6.67
 3     2 West    freunde           4.83
 4     2 West    eltern            5.17
 5    10 West    freunde           4
 6    10 West    eltern            6.83
 7    11 West    freunde           7
 8    11 West    eltern            7
 9    12 West    freunde           7
10    12 West    eltern            6.67
# i 558 more rows
```

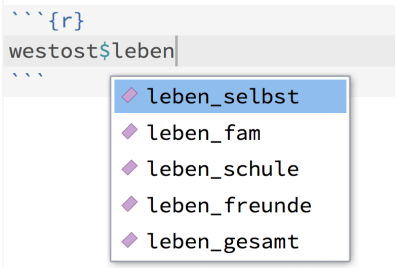
Und nun alles einem Schritt:

```r
unterstuetzung <- beispieldaten |>
  select(ID, westost, unt_freunde, unt_eltern) |>
  # fehlende Werte ausschliessen
  drop_na() |>
  # von wide zu long
  pivot_longer(!c(ID, westost), names_to = "quelle",
               values_to = "unterstuetzung") |>
  # Variable quelle zu Faktor konvertieren und
  # davor Faktorstufen bearbeiten
  mutate(quelle = factor(str_replace(quelle, ".*_", "")))

unterstuetzung
```

```
# A tibble: 568 x 4
      ID westost quelle  unterstuetzung
   <dbl> <fct>   <fct>            <dbl>
 1     1 West    freunde           6.67
 2     1 West    eltern            6.67
 3     2 West    freunde           4.83
 4     2 West    eltern            5.17
 5    10 West    freunde           4
 6    10 West    eltern            6.83
 7    11 West    freunde           7
 8    11 West    eltern            7
 9    12 West    freunde           7
10    12 West    eltern            6.67
# i 558 more rows
```

Zufriedenheit

Wir wollen nun die Zufriedenheit mit verschiedenen Lebensbereichen untersuchen. Wir brauchen dazu nicht den ganzen Datensatz. Die relevanten Variablen beginnen alle mit `leben_`, und sollen ausgewählt werden. Die Variable `leben_gesamt` ist aber schon eine Zusammenfassung der Zufriedenheit mit allen Bereichen, diese wollen wir nicht berücksichtigen.



Lösung

Dies bedeutet, dass wir alle Variablen beginnend mit `leben_` auswählen möchten, ausser `leben_gesamt`. Eine Möglichkeit wäre, die Variablen mit `starts_with("leben_")` auszuwählen, und dann `leben_gesamt` individuell auszuschliessen (das funktioniert hier nur mit `-leben_gesamt`, nicht mit `!leben_gesamt`). Zusätzlich wollen wir die IDs der Versuchspersonen behalten.

```
zufriedenheit_wide <- beispieldaten |>
  select(ID, starts_with("leben_"), -leben_gesamt)

zufriedenheit_wide
```

```
# A tibble: 286 x 5
      ID leben_selbst leben_familie leben_schule leben_freunde
   <dbl>        <dbl>         <dbl>        <dbl>         <dbl>
 1     1          5.5             7         4.33           6.5
 2     2          4.5          4.67         4.67             5
 3    10          6.5          6.33         3.33             5
 4    11            6             7            4           6.5
 5    12            6          5.67         5.33           6.5
 6    14            5          5.33            5             6
 7    15            6          5.67            4             5
 8    16            6          5.67         6.33             5
 9    17            6             6            5             6
10    18          5.5          4.67         4.33             5
# i 276 more rows
```

Weitere Möglichkeiten, dasselbe zu tun, sind:

```
beispieldaten |>
  select(ID, leben_selbst, leben_fam, leben_schule, leben_freunde)
```

```
variablen <- c("leben_selbst", "leben_fam",
               "leben_schule", "leben_freunde")
beispieldaten |>
  select(ID, one_of(variablen))
```

Die ausgewählten Zufriedenheitsvariablen können nun alle als Stufen eines messwiederholten Faktors aufgefasst werden. Der erstellte Datensatz ist aber noch im *wide* Format. Nun konvertieren wir diesen ins *long* Format.

```
zufriedenheit_long <- zufriedenheit_wide |>
  pivot_longer(!ID, names_to = "lebensbereich",
               values_to = "zufriedenheit")
```

Nun müssen wir uns an dieser Stelle überlegen, was mit fehlenden Werten geschehen soll. Wollen wir alle Versuchspersonen ausschliessen, welche auf mindestens einem Lebensbereich einen fehlenden Wert haben, oder wollen wir zulassen, dass Personen `NAs` haben, und dies später berücksichtigen?
Zuerst suchen wir Zeilen mit fehlenden Werten.

```
# Zeilen mit complete.cases() auswählen
# Die Verneinung ist !complete.cases()

# mit head(20) werden nur die ersten 20 in der Konsole angezeigt:
!complete.cases(zufriedenheit_wide) |>
  head(20)
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Dies hat einen `logical vector` als Output zur Folge. Damit können wir die Zeilen indizieren.

```
zufriedenheit_wide[!complete.cases(zufriedenheit_wide), ]
```

```
# A tibble: 2 x 5
     ID leben_selbst leben_familie leben_schule leben_freunde
  <dbl>        <dbl>         <dbl>        <dbl>         <dbl>
1   175           NA            NA        NA               NA
2   206            5            NA         5.33             5
```

Wir sehen, dass zwei Personen fehlende Werte haben und entscheiden uns, diese auszuschliessen.

```
zufriedenheit_long <- zufriedenheit_wide |>
  drop_na() |>
  pivot_longer(!ID, names_to = "lebensbereich",
               values_to = "zufriedenheit")
```

Die Variable `lebensbereich` sollte nun noch zu einem Faktor konvertiert werden.

```
zufriedenheit_long <- zufriedenheit_long |>
  mutate(lebensbereich = factor(lebensbereich))
zufriedenheit_long
```

```
# A tibble: 1,136 x 3
      ID lebensbereich zufriedenheit
   <dbl> <fct>                 <dbl>
 1     1 leben_selbst            5.5
 2     1 leben_familie           7
 3     1 leben_schule            4.33
 4     1 leben_freunde           6.5
 5     2 leben_selbst            4.5
 6     2 leben_familie           4.67
 7     2 leben_schule            4.67
 8     2 leben_freunde           5
 9    10 leben_selbst            6.5
10    10 leben_familie           6.33
# i 1,126 more rows
```

Vertiefung

Wir können die Faktorstufen noch "schöner" machen, indem wir das Präfix `leben_` entfernen.

```
library(stringr)
zufriedenheit_long <- zufriedenheit_long |>
  mutate(lebensbereich = factor(str_replace(lebensbereich, ".*_", "")))

levels(zufriedenheit_long$lebensbereich)
```

```
[1] "familie" "freunde" "schule"  "selbst"
```

Wir sortieren nun noch nach `ID`, und zwar in aufsteigender Reihenfolge.

```
zufriedenheit_long <- zufriedenheit_long |>
  arrange(ID)

zufriedenheit_long
```

```
# A tibble: 1,136 x 3
      ID lebensbereich zufriedenheit
   <dbl> <fct>                 <dbl>
 1     1 selbst                 5.5
 2     1 familie                7
 3     1 schule                 4.33
 4     1 freunde                6.5
 5     2 selbst                 4.5
 6     2 familie                4.67
 7     2 schule                 4.67
 8     2 freunde                5
 9    10 selbst                 6.5
10    10 familie                6.33
# i 1,126 more rows
```

Auch hier hätten wir alle Teilschritte zusammen ausführen können.

```
library(stringr) # für die Funktion str_replace()

zufriedenheit_long <- zufriedenheit_wide |>
  drop_na() |>
  pivot_longer(!ID, names_to = "lebensbereich",
               values_to = "zufriedenheit") |>
  mutate(lebensbereich = factor(str_replace(lebensbereich, ".*_", ""))) |>
  arrange(ID)
```

Stress

Verschiedene Befunde in der Literatur zeigen, dass Mädchen im Jugendalter stärker unter depressiven Verstimmungen leiden als Jungen. Diese Hypothese können wir mit einem t-Test überprüfen, aber bevor wir das tun, können wir uns die mittlere psychische Symptomatik, nach Geschlecht aufgeteilt, deskriptiv anschauen. Berechnen Sie Mittelwert und Standardabweichung der Variablen `stress_psychisch` für beide Geschlechter getrennt, und runden Sie diese auf drei Nachkommastellen.

Hinweis: Damit wir überhaupt drei Nachkommastellen angezeigt bekommen können, müssen

wir die Befehlsfolge immer mit `as.data.frame()` abschliessen. Der Grund dafür ist, dass der `tibble`, den die `tidyverse`-Funktionen erstellen, eine `default` Printmethode hat, die in manchen Fällen (z.B. in dieser Online-Version des Skripts) automatisch auf zwei Dezimalstellen rundet.

Lösung

```
stress <- beispieldaten |>
  select(ID, geschlecht, stress_psychisch) |>
  as.data.frame()

head(stress)
```

```
  ID geschlecht stress_psychisch
1  1   weiblich         1.666667
2  2   männlich         3.500000
3 10   weiblich         3.666667
4 11   weiblich         1.500000
5 12   weiblich         2.500000
6 14   männlich         1.000000
```

Zuerst noch ohne Runden:

```
stress |>
  group_by(geschlecht) |>
  summarize(mean = mean(stress_psychisch),
            sd = sd(stress_psychisch)) |>
  as.data.frame()
```

```
  geschlecht     mean       sd
1   männlich       NA       NA
2   weiblich 3.099502 1.120575
```

Vergessen Sie nicht, dass es möglicherweise fehlende Werte hat. Es gibt zwei Möglichkeiten, damit umzugehen:

1) Wir entfernen alle Personen mit fehlenden Werten.
2) Wir belassen die Zeilen mit fehlenden Werten im Datensatz und benützen das `na.rm` Argument der Funktionen `mean()` und `sd()`.

Jetzt runden wir die mit `summarize()` erstellten Statistiken, indem mit die gebildeten Variablen `mean` und `sd` mit `mutate()` verändern:

```
stress |>
  drop_na() |>
  group_by(geschlecht) |>
  summarize(mean = mean(stress_psychisch),
            sd = sd(stress_psychisch)) |>
  mutate(mean = round(mean, 3),
         sd = round(sd, 3)) |>
  as.data.frame()
```

```
  geschlecht mean    sd
1   männlich 2.89 1.254
2   weiblich 3.10 1.121
```

Wir können die `round()` Funktion aber auch innerhalb von `summarize()` benutzen, und das sogar in Kombination mit der Pipe! Ausserdem jetzt mit dem Argument `na.rm = TRUE` für `mean()`und `sd()` zur Entfernung von fehlenden Werten.

```
stress |>
  group_by(geschlecht) |>
  summarize(mean = mean(stress_psychisch, na.rm = TRUE) |> round(3),
            sd = sd(stress_psychisch, na.rm = TRUE) |> round(3)) |>
  as.data.frame()
```

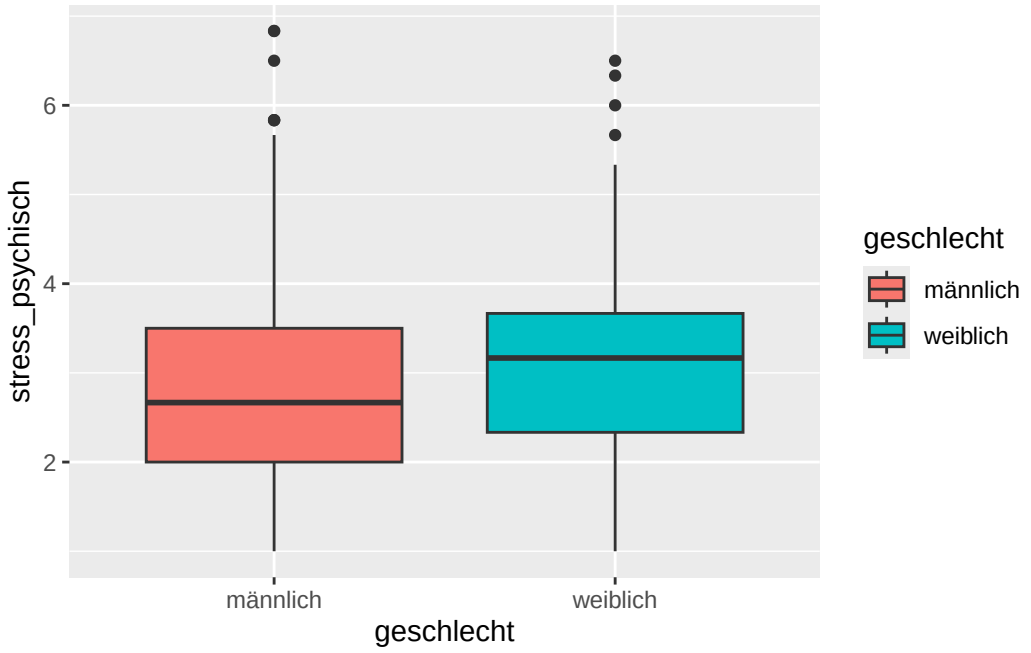
```
  geschlecht mean    sd
1   männlich 2.89 1.254
2   weiblich 3.10 1.121
```

Fun fact

Dass die beiden Geschlechts-Mittelwerte jetzt nur zwei Kommastellen aufweisen, ist ein Zufall: wenn nämlich die Rundung zu einer Nullstelle als letzter Stelle führt, dann schneidet `R` diese 0 ab, weil sie in gewisser Weise überflüssig ist. Der Mittelwert der Mädchen hat sogar noch eine Nullstelle an der zweiten Nachkommastelle, diese wird aber nicht abgeschnitten, da der Mittelwert der Jungen in der Zeile darüber zwei Nachkommastellen benötigt… Probieren Sie es einmal mit vier Nachkommastellen, dann sehen Sie es selbst!

Und jetzt zur Veranschaulichung noch ein Boxplot. Die Erstellung eines solchen Plots mit dem Package `ggplot2` ist Thema des nächsten Kapitels:

```r
library(ggplot2)
stress |>
  drop_na() |>
  ggplot(aes(x = geschlecht, y = stress_psychisch, fill = geschlecht)) +
  geom_boxplot()
```



5 Grafiken mit ggplot2

Grafiken sind für die Datenanalyse sehr wichtig. Einerseits können wir sie für explorative Datenanalyse einsetzen, um eventuell verborgene Zusammenhänge zu entdecken oder uns einfach einen Überblick zu verschaffen. Andererseits brauchen wir Grafiken, um Resultate darzustellen und anderen zu kommunizieren.

Wir haben schon mehrmals in diesem Skript Grafiken mit dem Package `ggplot2` erstellt, ohne uns den Code genauer anzuschauen. In diesem Kapitel werden wir nun die Syntax von `ggplot2` kennenlernen.

> Hinweis
>
> Es gibt in `R` verschiedene Möglichkeiten, Grafiken zu erstellen. Mit dem ursprünglichen Grafiksystem (`R` Base Graphics) kann man sehr schnell einfache Grafiken erstellen. Es ist auch sehr mächtig und flexibel, aber das Problem ist, dass die Syntax etwas archaisch erscheint, und es für Anfänger schwierig ist, Grafiken selber anzupassen.

Im Gegensatz dazu basiert `ggplot2` auf einer intuitiven Syntax, der sogenannten Grammar of graphics. Sobald man sich daran gewöhnt hat, kann man mit einer eleganten und konsistenten "Grammatik" sehr komplexe Grafiken erstellen. `ggplot2` ist darauf ausgelegt, mit `tidy` Data zu arbeiten, d.h. wir brauchen Datensätze im *long* Format. Grafiken werden nun immer nach demselben Prinzip erstellt:

Schritt 1: Wir beginnen mit einem Datensatz und erstellen ein Plot-Objekt mit der Funktion `ggplot()`.

Schritt 2: Wir definieren sogenannte "aesthetic mappings", d.h. wir bestimmen welche Variablen auf den X-, bzw. Y-Achsen dargestellt werden sollen, und welche Variablen benutzt werden, um die Daten zu gruppieren. Die Funktion, welche wir dafür benutzen heisst `aes()`.

Schritt 3: Wir fügen dem Plot eine oder mehrere "Layers" oder "Schichten" hinzu. Diese Layers definieren, wie etwas dargestellt werden soll, z.B. als Linie oder als Histogramm. Die Funktionen beginnen mit dem Präfix `geom_`, z.B. `geom_line()`.

Um `ggplot2` zu benutzen brauchen wir nun noch einen zusätzlichen Operator: `+`. Diesen kennen Sie bereits als mathematischen Operator, aber in diesem Zusammenhang bedeutet die Verwendung von `+`, dass wir einzelne Elemente eines Plot-Objektes zusammenfügen.

Nach dieser etwas abstrakten Einführung illustrieren wir diese Schritte an einem praktischen Beispiel.

Am Ende des letzten Kapitels haben wir den Zusammenhang zwischen psychischem Stress und Geschlecht untersucht. Wir laden nun nochmals den Datensatz:

```
library(tidyverse)
library(haven)
beispieldaten <- read_sav("data/beispieldaten.sav") |>
    # Faktoren konvertieren und SPSS-Labels zuweisen
    mutate(across(where(is.labelled), haven::as_factor)) |>
    # Labels der Bildungsfaktoren vereinfachen
    mutate(across(c(bildung_vater, bildung_mutter),
                  \(x) fct_recode(x,
                                  Hauptschule = "Hauptschulabschluss oder niedriger",
                                  Realschule = "Realschulabschluss (mittlere Reife)",
                                  Abitur = "Fachabitur, Abitur",
                                  Hochschule = "Fachhochschulabschluss, Universitätsabschluss
```

und erstellen einen Datensatz, der nur die Variablen `ID`, `geschlecht` und `stress_psychisch` enthält.

```
stress <- beispieldaten |>
  select(ID, geschlecht, stress_psychisch) |>
  drop_na()
stress
```

```
# A tibble: 284 x 3
      ID geschlecht stress_psychisch
   <dbl> <fct>                 <dbl>
 1     1 weiblich               1.67
 2     2 männlich               3.5
 3    10 weiblich               3.67
 4    11 weiblich               1.5
 5    12 weiblich               2.5
 6    14 männlich               1
 7    15 männlich               2.5
 8    16 weiblich               3.5
 9    17 männlich               1.67
10    18 männlich               2.5
# i 274 more rows
```

In diesem Datensatz haben wir eine numerische Variable, `stress_psychisch` und eine Gruppierungsvariable, `geschlecht`. Wir wollen die Verteilung von `stress_psychisch` zwischen männlichen und weiblichen Jugendlichen vergleichen. Die Verteilung können wir auf verschiedene Arten grafisch darstellen: mit Punkten, einem Boxplot oder einem Violin-Plot. Diese drei Methoden sind in der Sprache von `ggplot2` verschiedene `geoms` und können so benutzt werden: `geom_point()`, `geom_boxplot()` oder `geom_violin()`. Zusätzlich gibt es noch eine Funktion `geom_jitter()`, welche die Punkte in einem Punktdiagramm nicht aufeinander zeichnet, sondern mit einem zufälligen horizontalen "jittering" (Flackern) versieht.

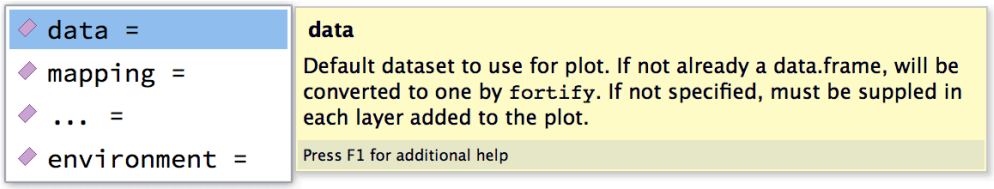
Das `ggplot2` Package können wir entweder individuell oder als Teil des `tidyverse` laden:

```
library(ggplot2)

# oder

library(tidyverse)
```

5.1 Schritt 1: Plot-Objekt erstellen

Wir beginnen mit einem Datensatz und erstellen ein Plot-Objekt mit der Funktion `ggplot()`. Diese Funktion hat als erstes Argument einen Dataframe. Dies bedeutet, dass wir den `pipe` Operator verwenden können:

```
>
> ggplot()
```



Wir haben also zwei Möglichkeiten. Wir bevorzugen hier die `pipe` Notation, aber es ist selbstverständlich auch möglich, den Dataframe innerhalb der Funktion als Argument anzugeben. Gleichzeitig weisen wir das Objekt einer Variablen zu, und nennen diese `p`.

```
# 1. Variante
p <- ggplot(data = stress)

# 2. Variante
```

```r
p <- stress |>
  ggplot()
```

5.2 Schritt 2: Aesthetic mappings

Nun definieren wir mit dem zweiten Argument `mapping` die *aesthetic mappings*. Diese bestimmen, wie die Variablen benutzt werden, um die Daten darzustellen, und werden mit der Funktion `aes()` definiert. Wir wollen die Gruppierungsvariable `geschlecht` auf der X-Achse darstellen und `stress_psychisch` soll auf der Y-Achse angezeigt werden. Zusätzlich kann `aes()` weitere Argumente haben: `fill`, `color`, `shape`, `linetype`, `group`. Diese werden dazu benutzt, um den Stufen der Gruppierungsvariablen (Faktoren) unterschiedliche Farben, Formen, Linien, etc. zuzuweisen.

In diesem Beispiel haben wir die Gruppierungsvariable `geschlecht` und wir wollen, dass die beiden Stufen von `geschlecht` verschiedene Farben haben und mit verschiedenen Farben "ausgefüllt" werden.

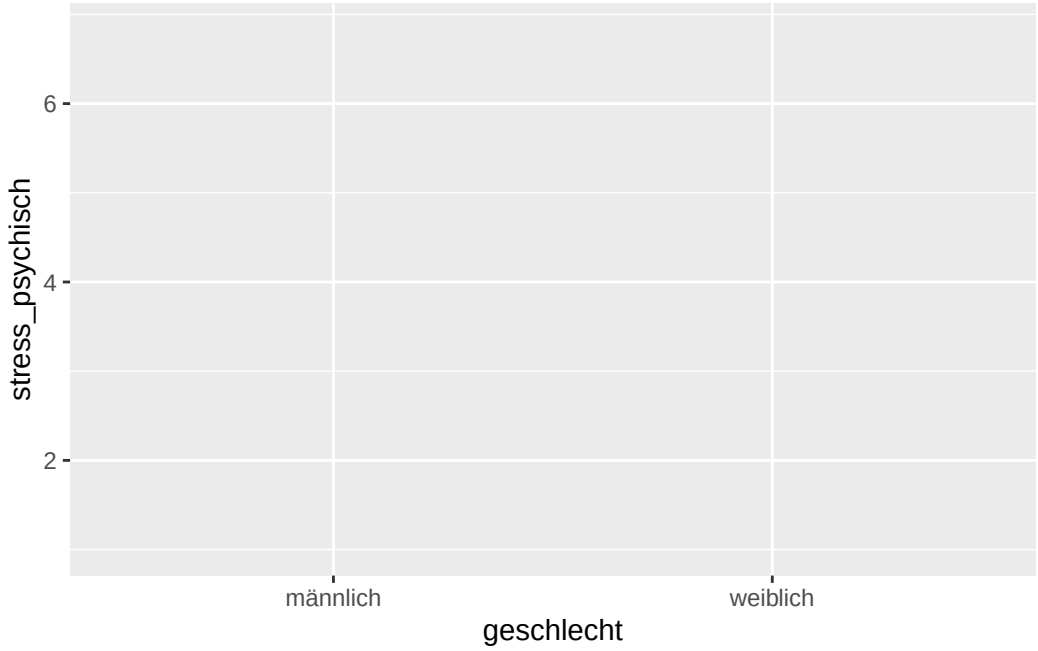
> Hinweis
>
> `color` ist ein Attribut von Linien oder Punkten, `fill` ist ein Attribut von Flächen.

Wenn wir die *aesthetic mappings* innerhalb der Funktion `ggplot()` definieren, gelten sie für alle *Layers*, d.h. für alle Elemente des Plots. Wir könnten diese mappings auch für jede *Layer* separat definieren.

```r
p <- stress |>
  ggplot(mapping = aes(
    x = geschlecht,
    y = stress_psychisch,
    color = geschlecht,
    fill = geschlecht
  ))
```

`p` ist nun ein "leeres" Plot-Objekt. Wir können es uns anschauen, aber es wird noch nichts angezeigt, da es noch keine *Layers* enthält:

```r
p
```




```
# oder print(p)
```

Wir sehen, dass `ggplot2` für uns schon die Achsen anhand der Variablennamen beschriftet hat.

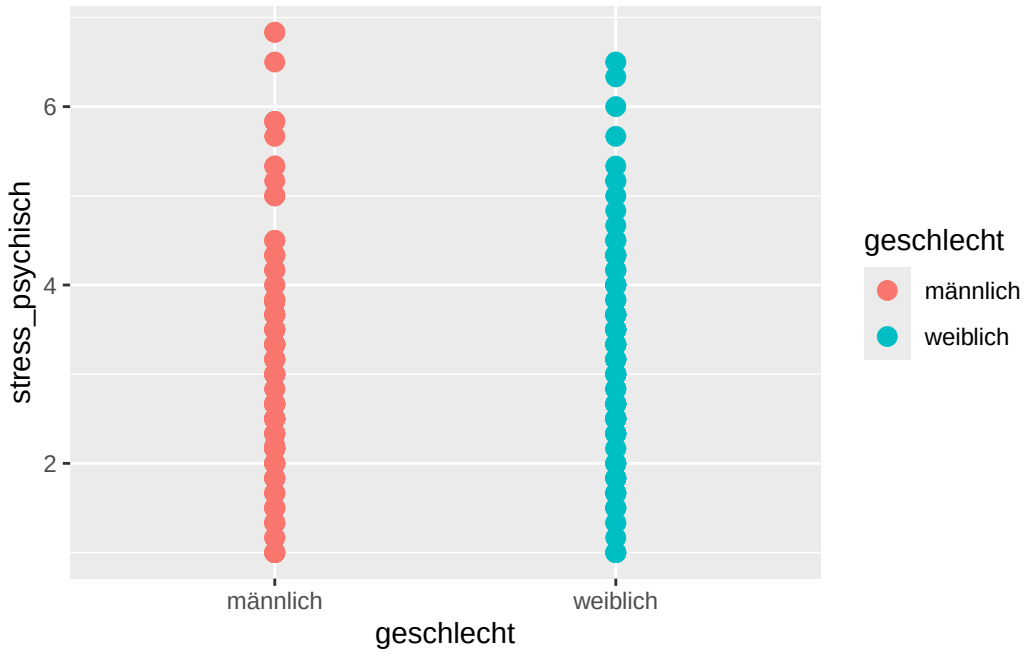
5.3 Schritt 3: geoms hinzufügen

Dem Plot-Objekt `p` können wir nun mit `geom_` Funktionen *Layers* hinzufügen. Die Syntax funktioniert so: Wir "addieren" zu dem Plot-Objekt `p` ein `geom`: `p + geom_`

5.3.1 Punktdiagramm

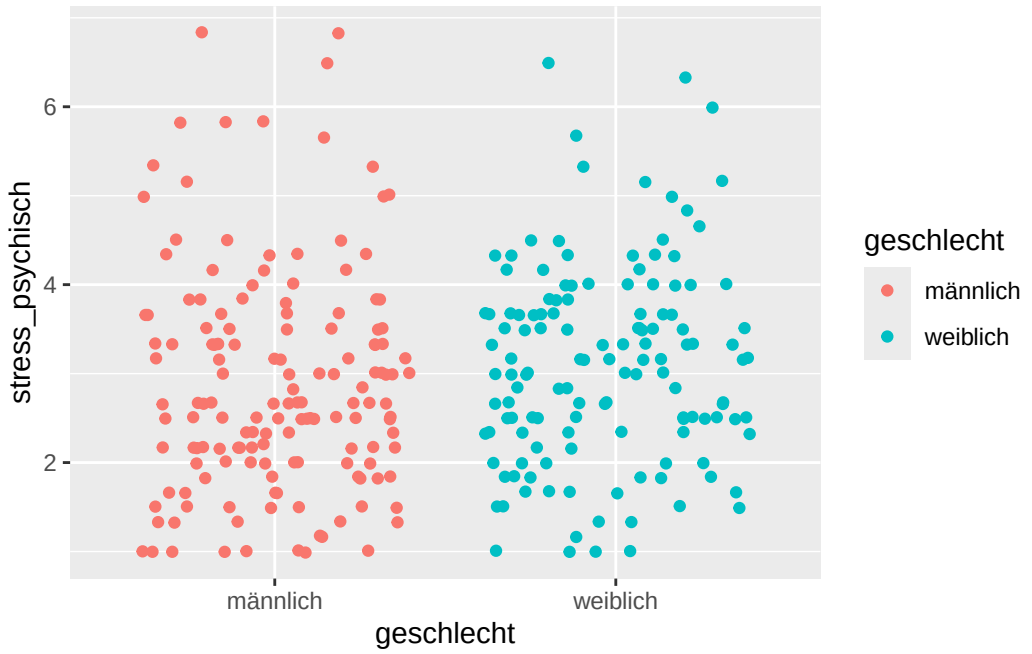
Wir versuchen zuerst, die Beobachtungen als Punkte darzustellen:

```
# die Funktion geom_point() hat ein size Argument
p + geom_point(size = 3)
```



Die Punkte werden nun in verschiedenen Farben dargestellt, aber innerhalb eines Geschlechts werden Punkte eventuell übereinander geplottet, wenn sie denselben Wert haben (overplotting). Für diesen Fall gibt es die Funktion `geom_jitter()`, welche Punkte mit einem *jittering* nebeneinander zeichnet:

```
p + geom_jitter()
```



`geom_jitter()` hat ein Argument `width`, mit dem wir bestimmen können, wie breit die Streuung der Punkte ist.

```
p + geom_jitter(width = 0.2)
```

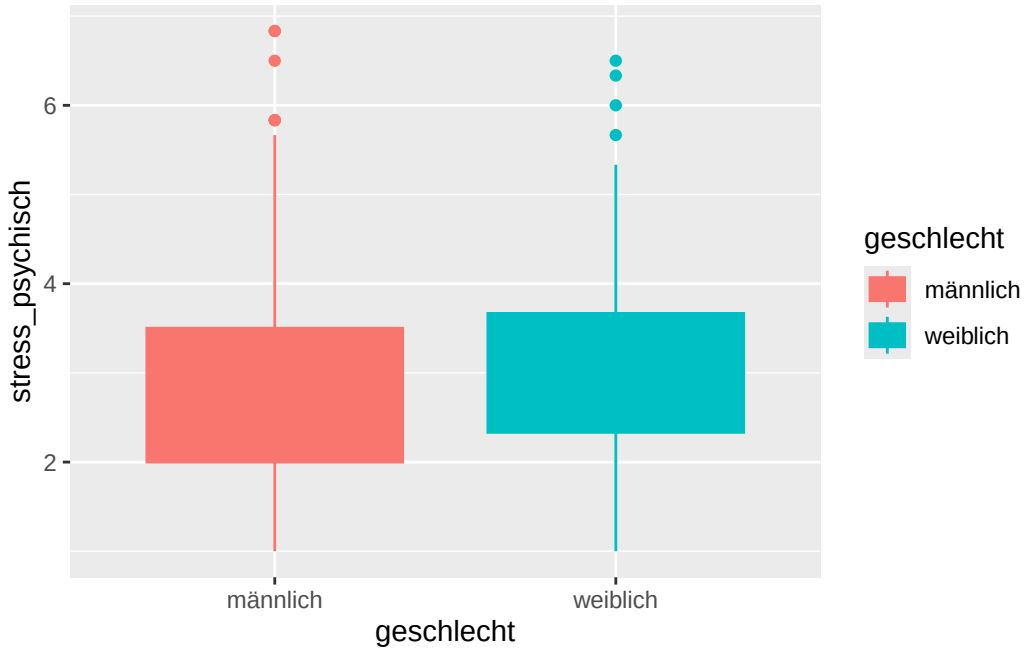
`geom_jitter()` hat weitere Argumente: `size` bestimmt den Durchmesser der Punkte, und `alpha` bestimmt die Transparenz.

```
p + geom_jitter(width = 0.2, size = 4, alpha = 0.6)
```

5.3.2 Verteilung grafisch darstellen

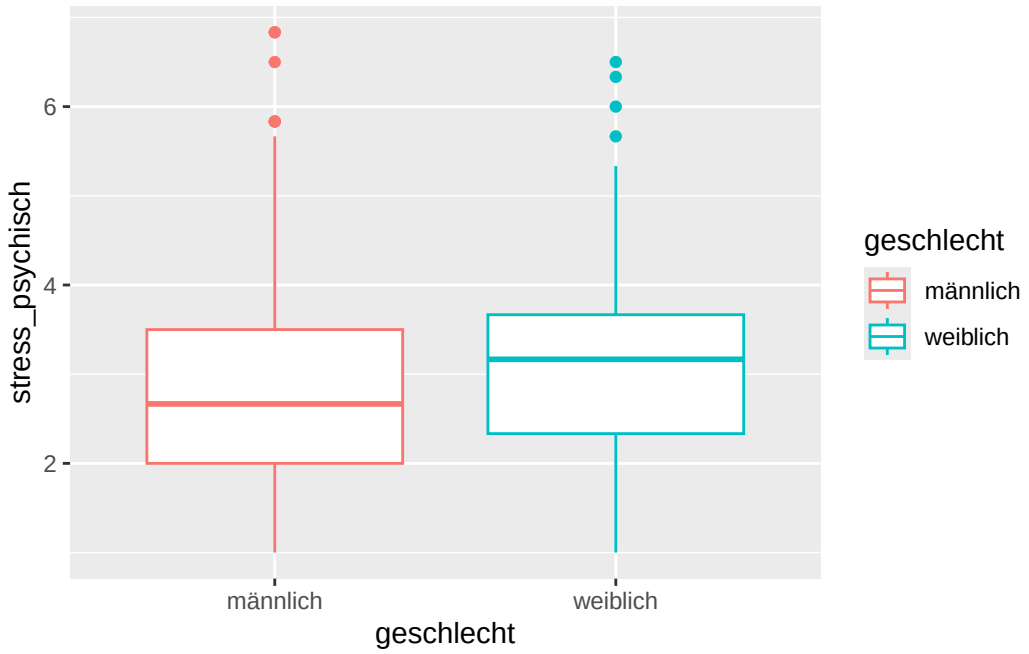
Eine weitere Möglichkeit wäre, die zentrale Tendenz und Streuung der Daten mit einem Boxplot- oder Violin-Diagramm darzustellen.

```
p + geom_boxplot()
```



In einem Boxplot wird der Median dargestellt, das Rechteck repräsentiert die mittleren 50%, und die "whiskers" zeigen 1.5 * den Interquartilsbereich. Ausreisser werden mit Punkten dargestellt. Um den Median zu sehen, ist es besser, wenn wir das `fill` Attribut weglassen:

```
p <- stress |>
  ggplot(mapping = aes(
    x = geschlecht,
    y = stress_psychisch,
    color = geschlecht
  ))

p + geom_boxplot()
```



Ein Violin-Plot ist ähnlich wie ein Boxplot, zeigt aber nicht die Quantile, sondern ein "kernel density estimate". Ein Violin-Plot sieht am besten aus, wenn wir das `fill` Attribut verwenden.

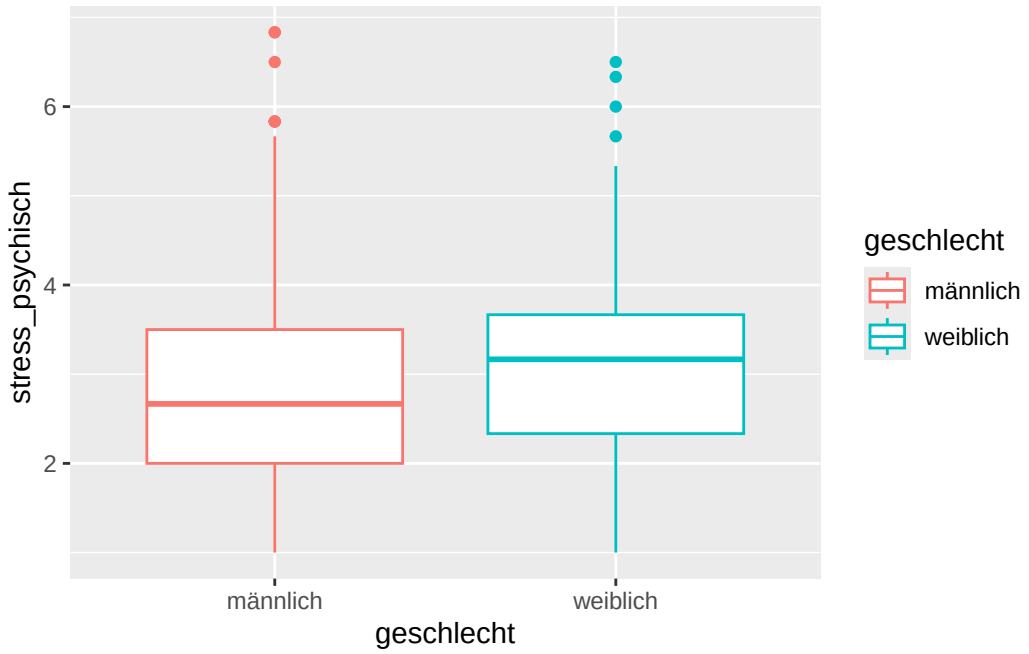
```
p <- stress |>
  ggplot(mapping = aes(
    x = geschlecht,
    y = stress_psychisch,
    fill = geschlecht
  ))
p + geom_violin()
```

Wenn wir feststellen, dass ein Mapping nicht für alle "Layers" gelten soll, dann können wir es für jede *Layer* individuell definieren, anstatt in der `ggplot()` Funktion:

```
p <- stress |>
  ggplot(mapping = aes(
    x = geschlecht,
    y = stress_psychisch
  ))
```

```
p + geom_boxplot(mapping = aes(color = geschlecht))
```

```
# oder einfach
p + geom_boxplot(aes(color = geschlecht))
```




```
p + geom_violin(aes(fill = geschlecht))
```

5.3.3 Mehrere Layers kombinieren

Wir können auch mehrere *Layers* verwenden. Wir müssen lediglich mehrere `geom_` Funktionen mit einem `+` zusammenfügen:

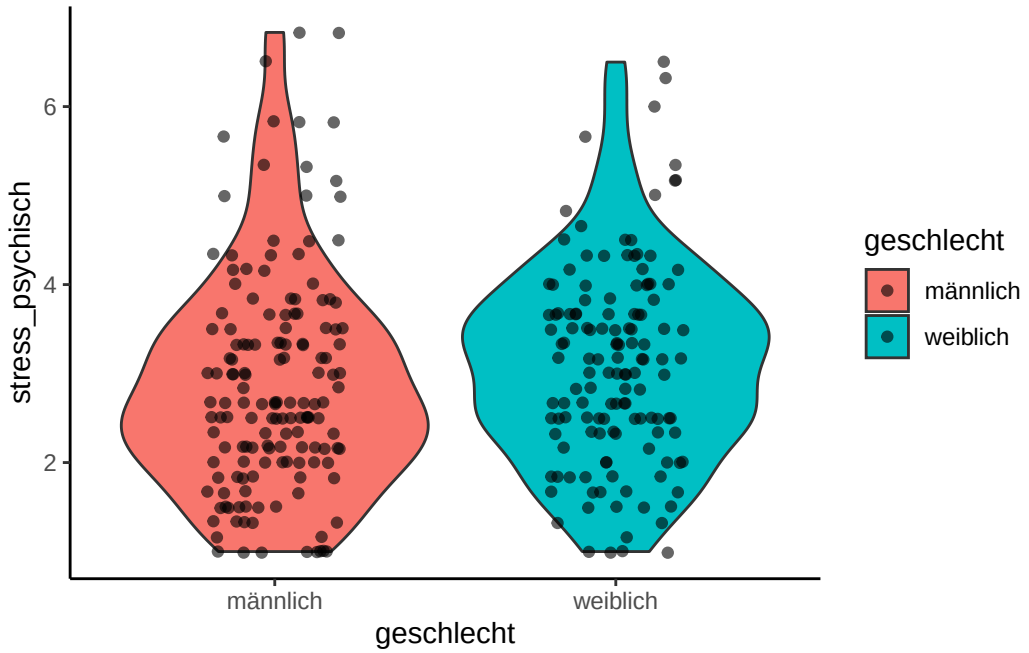
```
p +
  geom_violin(aes(fill = geschlecht)) +
  geom_jitter(width = 0.2, alpha = 0.6)
```

> Übung
>
> Schauen Sie sich die Grafikbeispiele in den vorangegangenen Kapiteln an. Verstehen Sie nun den Code?

In den bisherigen Beispielen (vgl. Kapitel 3) haben wir kein Plot-Objekt erstellt, sondern den Datensatz mit dem `pipe` Operator an die `ggplot()` Funktion geschickt, und dann mit `+` direkt die `geoms` hinzugefügt. Die Erstellung eines Plot-Objekts `p` ist also nicht zwingend notwendig, vereinfacht aber die Arbeit mit `ggplot2` besonders am Anfang. Ausserdem haben wir weitere Funktionen verwendet, wie z.B. `theme_classic()`, um den Hintergrund weiss darzustellen. Auf diese zusätzlichen Funktionen kommen wir weiter unten zurück.

```
stress |>
  ggplot(mapping = aes(
    x = geschlecht,
```

```
    y = stress_psychisch,
    fill = geschlecht
)) +
geom_violin() +
geom_jitter(width = 0.2, alpha = 0.6) +
theme_classic()
```



5.4 Geoms für verschiedene Datentypen

Wir fassen zusammen: bisher haben wir gelernt, dass wir einen Plot in mehreren Schritten zusammenstellen. Wir beginnen mit einem Dataframe und definieren mit der `ggplot()` Funktion ein `ggplot2` Objekt. Mit der `aes()` Funktion weisen wir Variablen eines Dataframes der X-, bzw. der Y-Achse zu und definieren weitere *aesthetic mappings*, z.B. eine farbliche Codierung anhand einer Gruppierungsvariablen. Anschliessen fügen wir dem Plot-Objekt Grafikelemente mit `geom_` Funktionen als Layers hinzu.

Nun schauen wir uns eine Auswahl an `geoms` für verschiedene Kombination von Variablen an. Wir können dabei entweder eine Variable auf der X-Achse oder zwei Variablen auf den X- und Y-Achsen darstellen und diese Variablen können entweder kontinuierlich oder kategorial sein.

Hinweis

Wir werden hier nur eine kleine Auswahl der möglichen `ggplot2` Funktionen betrachten. Das Package ist sehr umfangreich und hat eine sehr übersichtliche Website, auf der alles dokumentiert ist: ggplot2 Dokumentation
Nachdem Sie dieses Kapitel durchgearbeitet haben, sind Sie in der Lage, selber Lösungen für grafische Darstellungen zu finden. Datenvisualisierung kann ein sehr kreativer Prozess sein und macht Spass! Weitere Beispiele für spezifische Datenanalysemethoden sehen Sie in den nachfolgenden Kapiteln.

Für die folgenden Beispiele verwenden wir die Datensätze `beispieldaten` und `kinderwunsch`. Während wir `beispieldaten` schon weiter oben eingelesen haben, müssen wir den Datensatz `kinderwunsch.sav` noch herunterladen, in unserem `data`-Ordner abspeichern (`kinderwunsch.sav`) und anschliessend einlesen:

```
kinderwunsch <- read_sav("data/kinderwunsch.sav") |>
  mutate(geschlecht = haven::as_factor(geschlecht))
```

5.4.1 Eine Variable

Wenn wir nur eine Variable auf der X-Achse grafisch darstellen möchten, müssen wir aber dennoch Werte auf der Y-Achse darstellen. Dies wird oft eine deskriptive Zusammenfassung wie z.B. Häufigkeiten sein.

Kategoriale Variablen

Wenn wir eine kategoriale Variable grafisch darstellen, verwenden wir oft einen **bar chart** or **bar graph**. Dieser stellt z.B. Häufigkeiten der verschiedenen Kategorien anhand eines Rechtecks (rectangular bar) dar. Die Funktion, welche dafür verwendet wird, heisst `geom_bar()`.

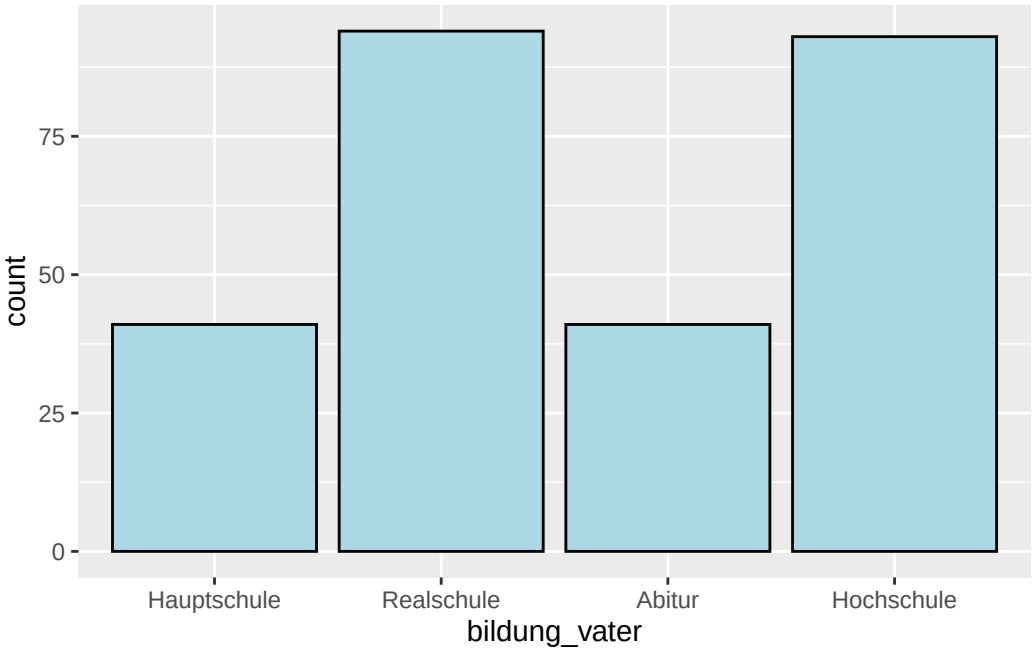
Als Beispiel wollen wir die Häufigkeiten der vier Bildungsstufen des Vaters plotten.

Hinweis

Wenn wir `fill = "lightblue", color = "black"` nicht innerhalb der `aes()` Funktion verwenden, dann werden diese Argumente nicht als Gruppierungsanweisung aufgefasst. Wir können z.B. mit `fill = "lightblue"` einfach alle Elemente hellblau einfärben.

```
p <- beispieldaten |>
  select(bildung_vater) |>
  drop_na() |>
```

```
  ggplot(aes(x = bildung_vater))
p + geom_bar(fill = "lightblue", color = "black")
```



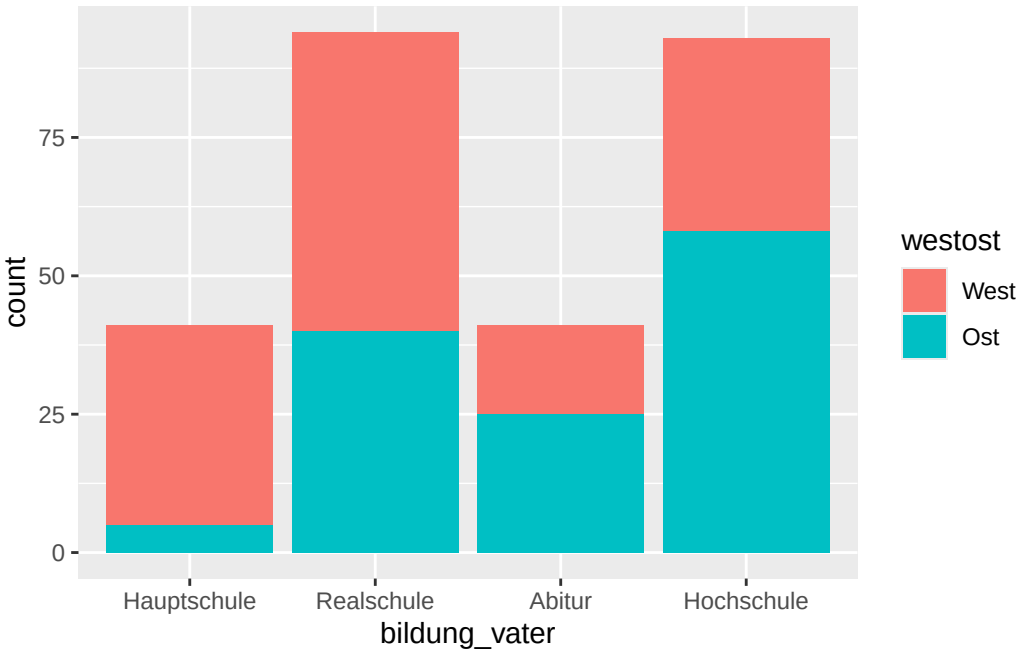
Eine Übersicht über die möglichen Farbnamen erhalten Sie mit der Funktion `colors()`. Es gibt 657 davon, wir zeigen hier mit `sample(15)` nur 15 zufällig ausgewählte an:

```
colors() |> sample(15)
```

```
 [1] "maroon4"           "hotpink4"          "gray8"
 [4] "aquamarine2"       "mediumspringgreen" "grey53"
 [7] "gray40"            "grey87"            "cornflowerblue"
[10] "purple4"           "palegreen2"        "darkslategray4"
[13] "blue4"             "orchid1"           "lightslategrey"
```

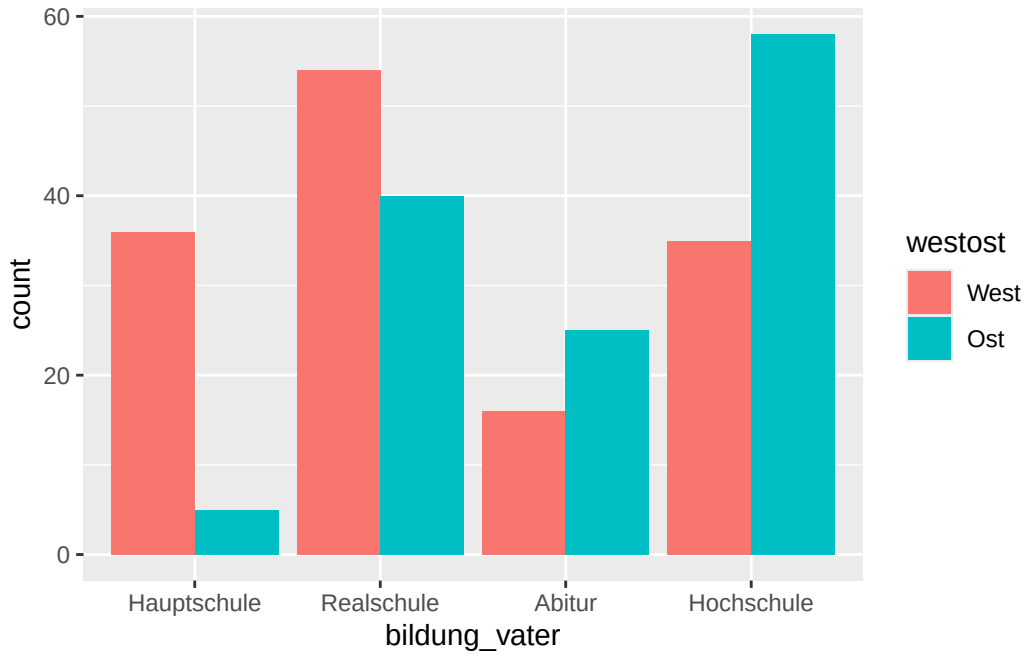
Auch hier können wir zusätzlich eine Gruppierungsvariable angeben, anhand derer wir die Rechtecke farblich kodieren.

```
p <- beispieldaten |>
  select(bildung_vater, westost) |>
  drop_na() |>
  ggplot(aes(x = bildung_vater, fill = westost))
p + geom_bar()
```



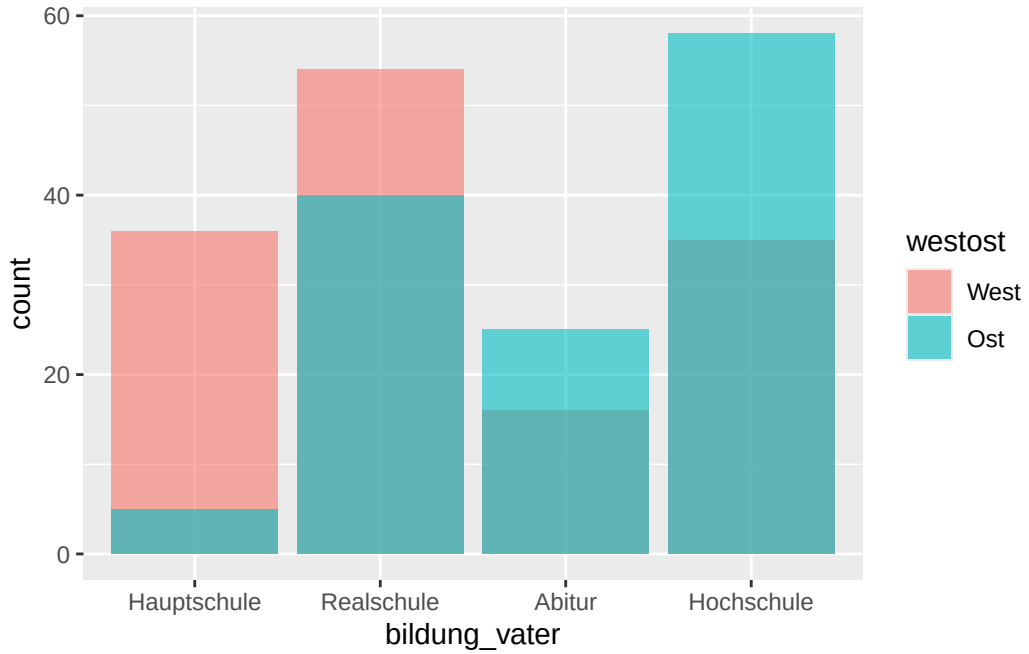
Standardmässig kreiert `ggplot2` einen *stacked* Bar Chart, d.h. die Rechtecke werden aufeinander gestapelt. Wenn dies nicht erwünscht ist, können wir das Argument `position = "dodge"` der Funktion `geom_bar()` verwenden. Damit teilen wir mit, dass die Bars nebeneinander gezeichnet werden sollen.

```
p + geom_bar(position = "dodge")
```



Als dritte Variante können wir `position = "identity"` verwenden; so werden die Bars übereinander gezeichnet. Da das hintere Rechteck nicht mehr sichtbar ist, verwenden wir das `alpha` Argument, um die Bars transparent zu machen.

```
p + geom_bar(position = "identity", alpha = 0.6)
```



Kontinuierliche Variablen

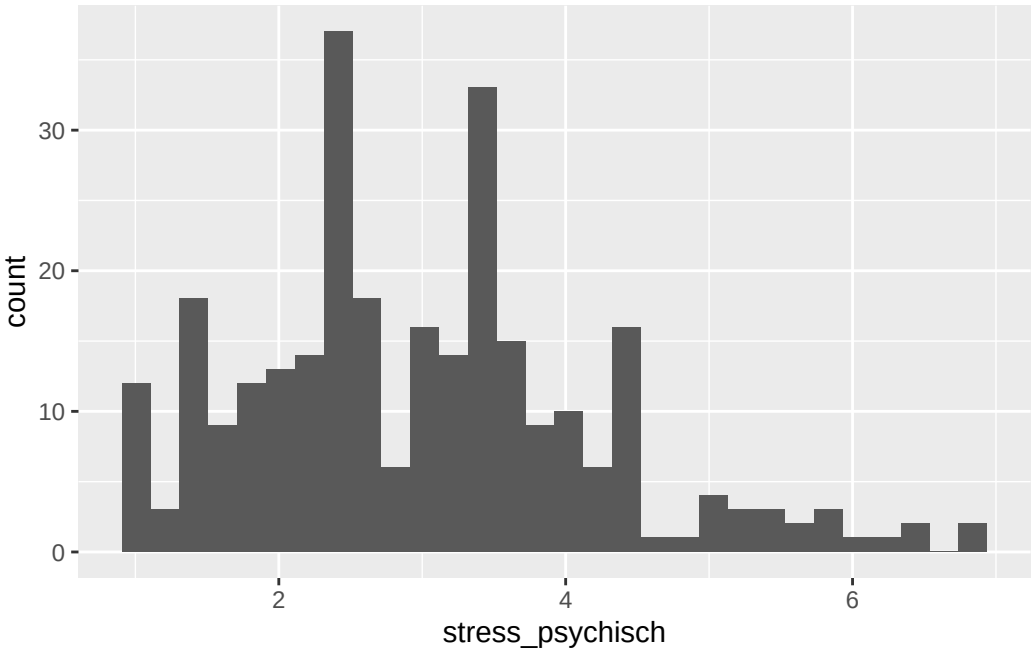
Falls die Variable, welche wir grafisch darstellen wollen, nicht kategorial, sondern kontinuierlich ist, bietet sich ein Histogramm an; dies erzeugen wir mit der Funktion `geom_histogram()`. Als Beispiel betrachten wir den psychischen Stress.

Ein Histogramm bietet eine grafische Darstellung der Verteilung einer numerischen Variablen. Dazu werden die Werte dieser Variablen in diskrete Intervalle, oder `bins`, unterteilt. Auf der Y-Achse werden dann, analog zu einem Bar Chart, die Häufigkeiten in den jeweiligen Intervallen dargestellt. Die Bestimmung der Grösse der Intervalle (`binwidth`) ist kritisch. Wenn wir nichts spezifieren, wählt `ggplot2` selber eine `binwidth` aus, aber wir können diese mit dem Argument `binwidth` auch selber angeben.

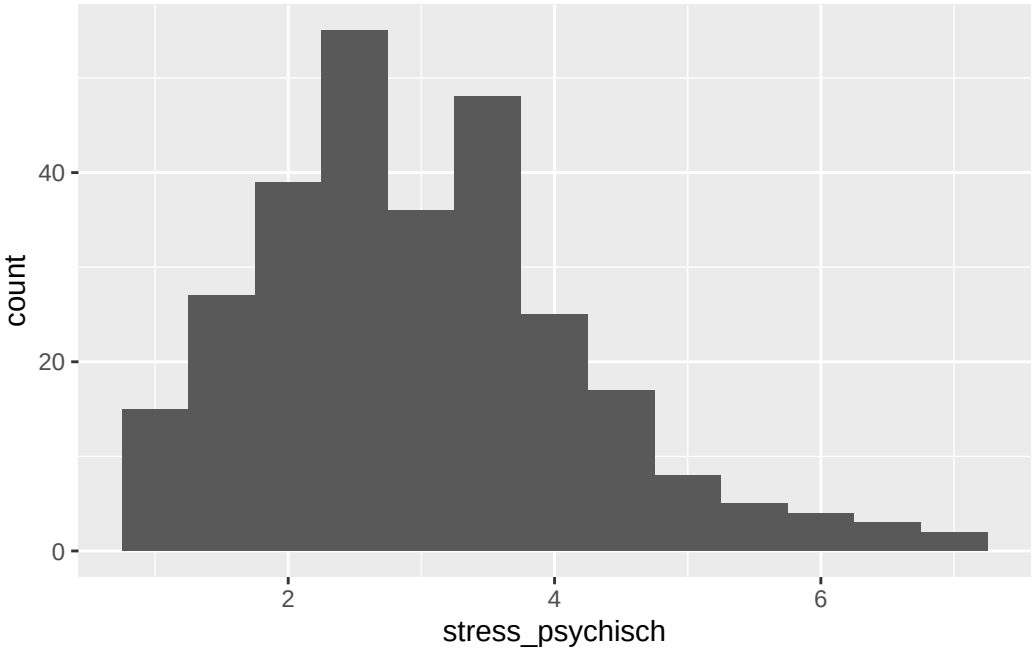
Wir verwenden wieder den bereits oben gebildeten Datensatz `stress`, der nur die Variablen `ID`, `stress_psychisch` und `geschlecht` beinhaltet:

```
p <- stress |>
  ggplot(mapping = aes(x = stress_psychisch))

# Wir lassen die binwidth automatisch auswählen
p + geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



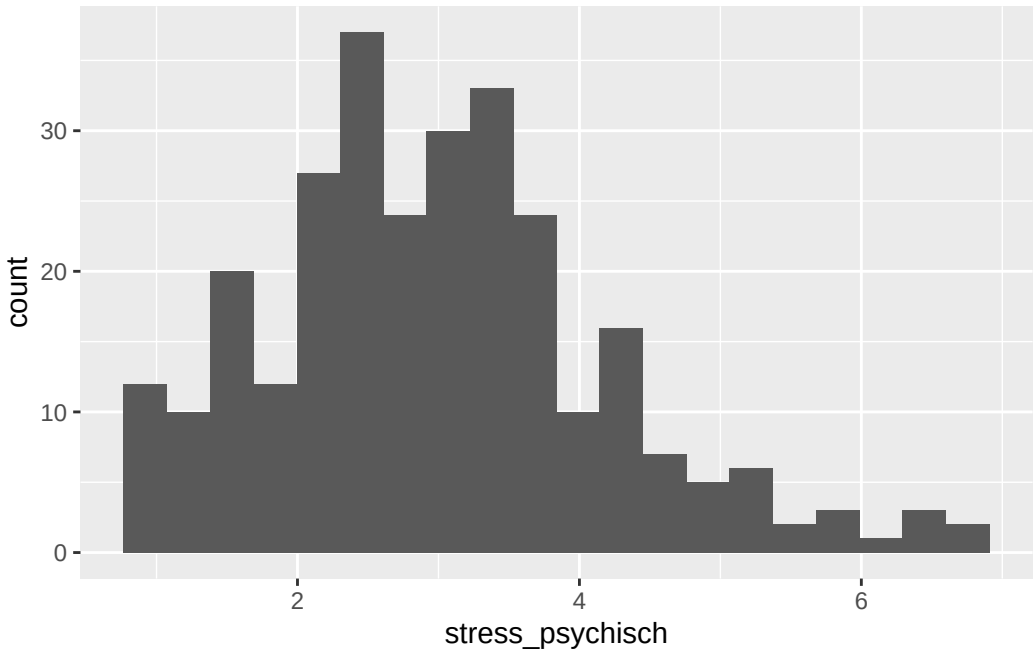
```
# Wir bestimmen die binwidth selber
p + geom_histogram(binwidth = 0.5)
```



Die Bestimmung der `binwidth` hängt natürlich von der Skala der Variablen ab und sollte weder zu fein noch zu grob sein.

Statt der `binwidth` kann mit `bins` auch die Anzahl der insgesamt zu bildenden Bins angegeben werden:

```
p + geom_histogram(bins = 20)
```

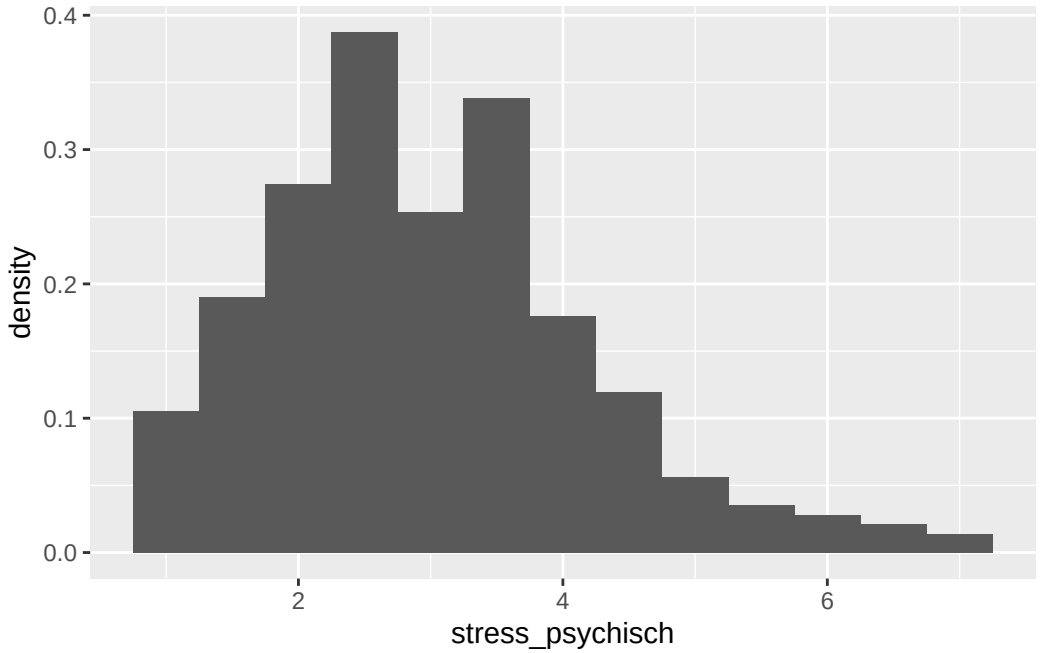


Übung

Probieren Sie mehrere Werte für `binwidth` und/oder `bins` aus. Für welchen Plot würden Sie sich entscheiden?

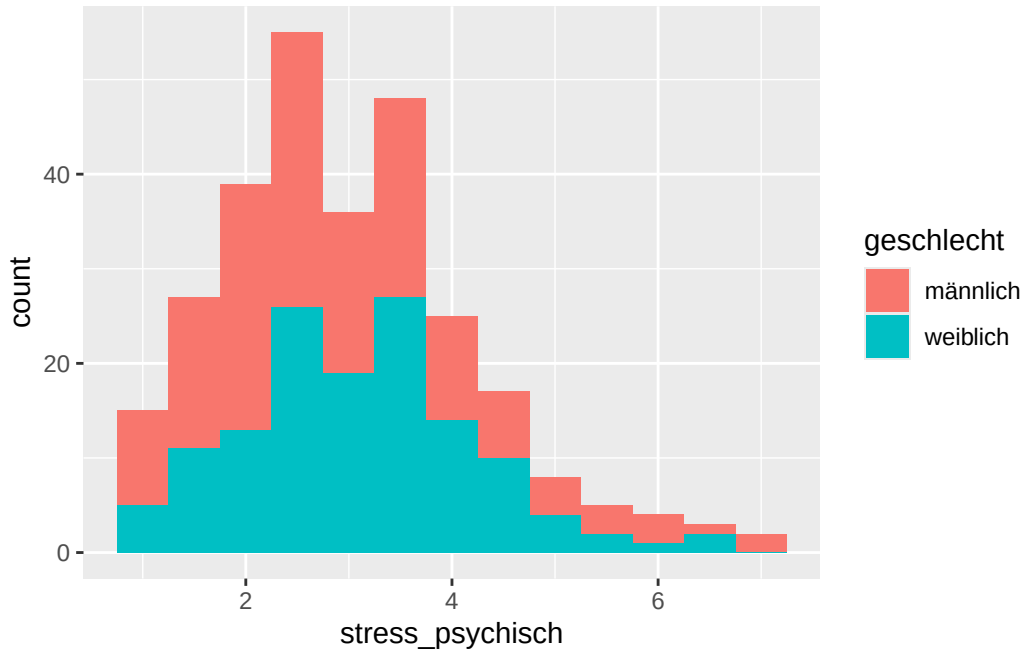
Wenn wir auf der Y-Achse anstelle der absoluten die relativen Häufigkeiten sehen wollen, können wir `y = after_stat(density)` als Argument der `aes()` Funktion verwenden.

```
p + geom_histogram(binwidth = 0.5, aes(y = after_stat(density)))
```



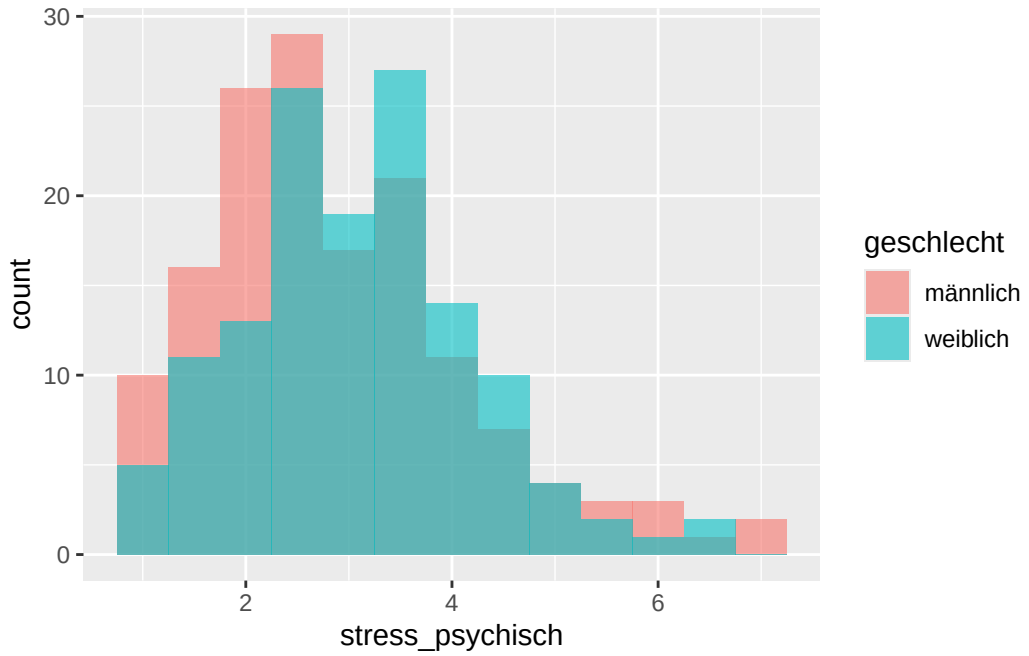
Selbsverständlich gibt es auch für Histogramme die Möglichkeit, einen Faktor als Gruppierungsvariable zu verwenden.

```
p <- stress |>
  ggplot(mapping = aes(x = stress_psychisch, fill = geschlecht))

p + geom_histogram(binwidth = 0.5)
```



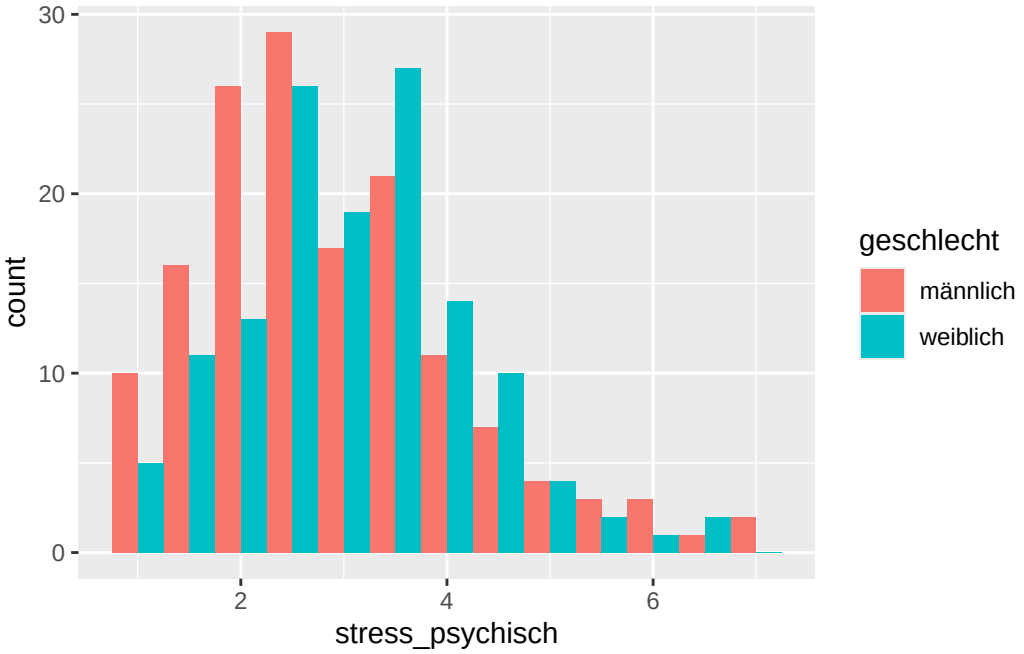
Wie beim Bar Chart werden die Histogramme übereinander (*stacked*) geplotted. Wollen wir sie aufeinander, verwenden wir `position = "identity"`.

```
p + geom_histogram(
  binwidth = 0.5,
  position = "identity",
  alpha = 0.6
)
```



Nebeneinander geht auch:

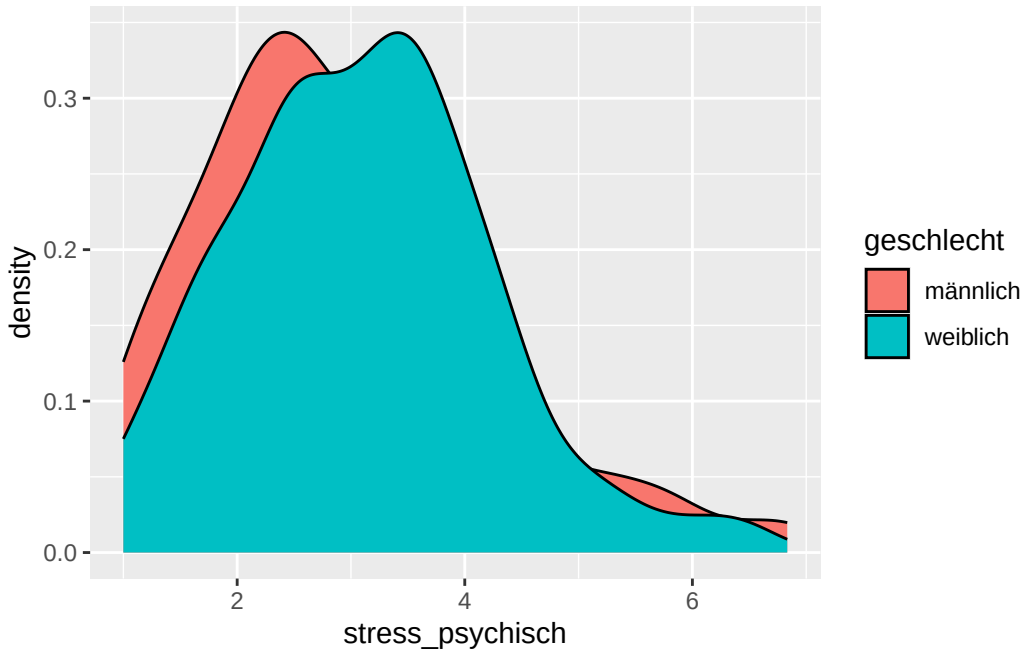
```
p + geom_histogram(
  binwidth = 0.5,
  position = "dodge"
)
```



Eine gute Alternative zum Histogramm ist ein Density-Plot, den man mit `geom_density()` erhält. Ein Density-Plot ist eine geglättete ("smoothed") Version eines Histogramms mit *kernel density estimates*, also mit geglätteten relativen Häufigkeiten. Wir kennen solche Density-Kurven schon vom Violin-Plot weiter oben (dort vertikal und symmetrisch gespiegelt dargestellt). Ein Density-Plot hat den Vorteil, dass die als kontinuierlich/stetig angenommene Variable `stress_psychisch` auch in einer kontinuierlichen Form dargestellt wird.

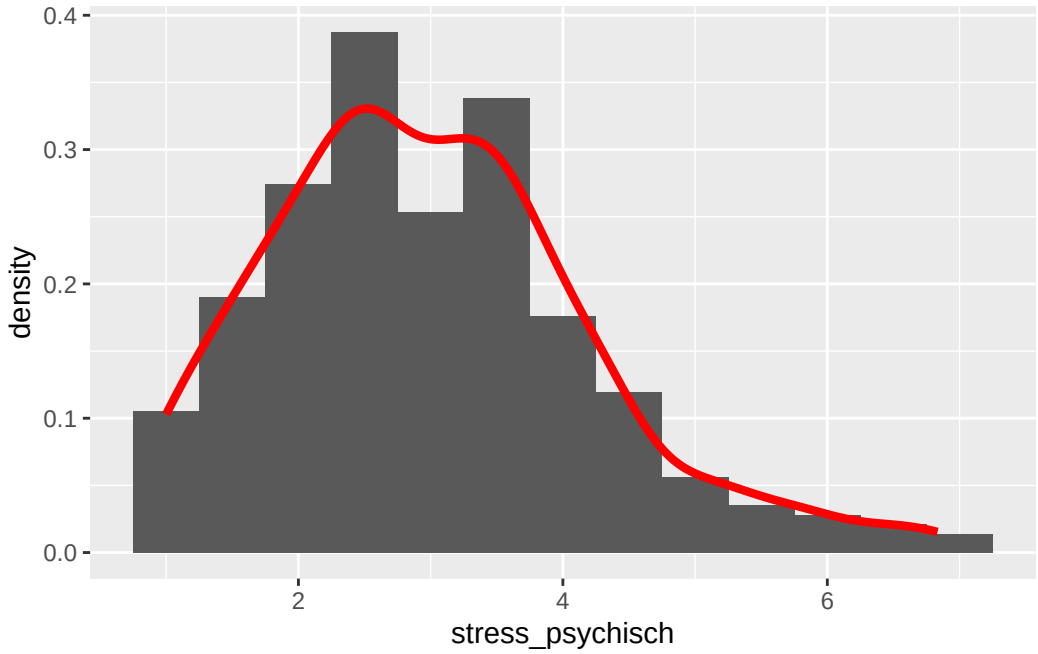
Hier gleich mit dem Plot-Objekt von oben, also getrennt für die beiden Geschlechter mit unterschiedlich farbigen Füllungen:

```
p + geom_density()
```



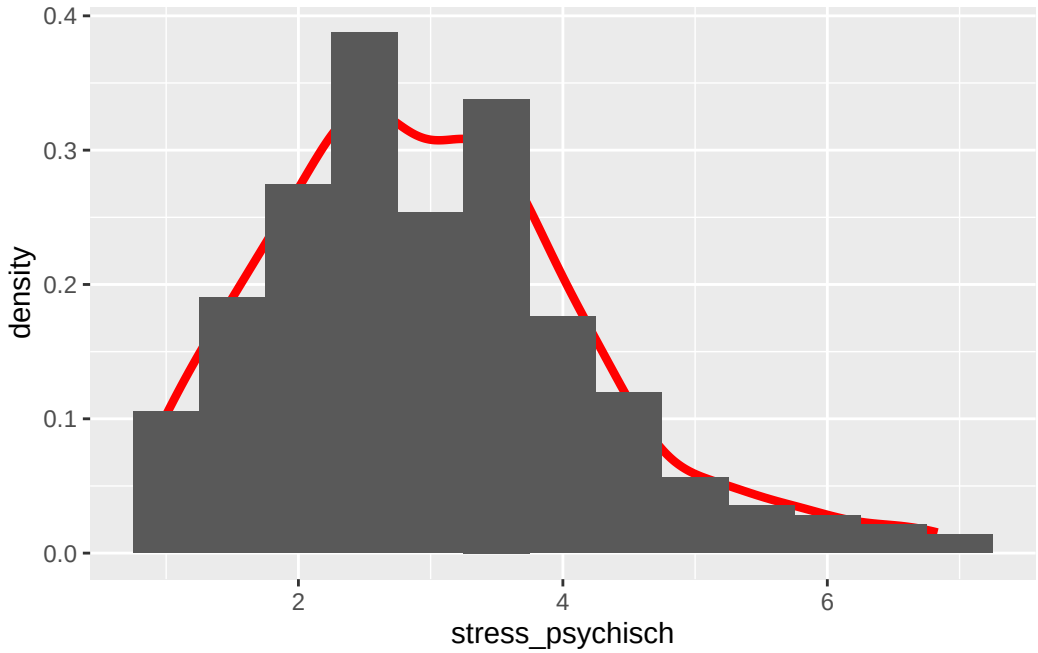
Ein Histogramm kann auch durch eine Density-Kurve ergänzt werden. Hier nochmal das Histogramm für die Gesamtstichprobe (mit `aes(y = after_stat(density))`) und mit roter Density-Kurve als zweitem Layer:

```
p <- stress |>
  ggplot(mapping = aes(x = stress_psychisch))

p +
  geom_histogram(binwidth = 0.5, aes(y = after_stat(density))) +
  geom_density(color = "red", linewidth = 1.5)
```



Die beiden Layer in umgekehrter Reihenfolge zu plotten wäre dagegen nicht optimal:

```
p +
  geom_density(color = "red", linewidth = 1.5) +
  geom_histogram(binwidth = 0.5, aes(y = after_stat(density)))
```



Häufig lässt man sich ein Histogramm ausgeben und legt nicht wie oben die empirische Density-Kurve darüber, sondern eine normalverteilte Kurve (Normalverteilung mit Mittelwert und der Standardabweichung der betreffenden Variable). Dazu verwendet man `stat_function()` mit den unten angegebenen Argumenten. Auf diese Weise lässt sich die empirische Verteilung der Variable (graues Histogramm) mit der unter einer Annahme einer Normalverteilung erwarteten Kurve vergleichen (rot). Falls die Abweichungen zwischen Histogramm und Kurve sehr gross sind, ist die Normalverteilungsannahme deutlich verletzt.

```
p +
  geom_histogram(binwidth = 0.5, aes(y = after_stat(density))) +
  stat_function(fun = dnorm,
                args = list(mean = mean(stress$stress_psychisch),
                            sd = sd(stress$stress_psychisch)),
                lwd = 1.5,
                col = 'red')
```

5.4.2 Zwei Variablen

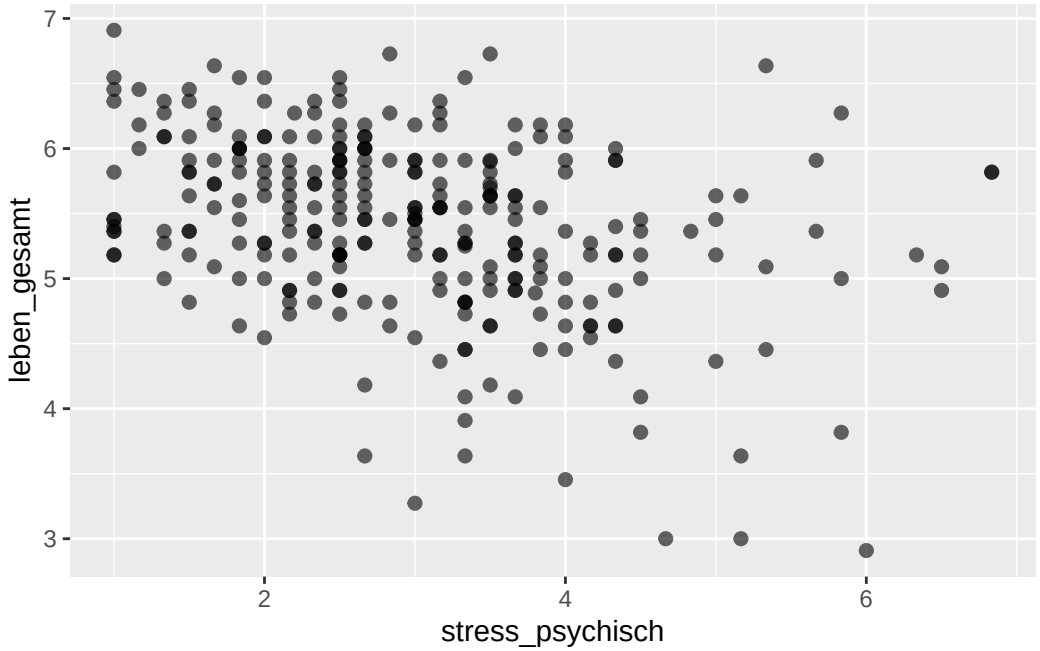
Nun stellen wir zwei Variablen eines Datensatzes gemeinsam dar. Auch hier hängen die möglichen `geoms` vom Datentyp der Variablen ab.

X und Y kontinuierlich

Wenn beide Variablen kontinuierlich sind, können wir deren Zusammenhang anhand eines 'Scatterplots' oder eines Liniendiagrams darstellen. Wir verwenden die Funktionen `geom_point()`, bzw. `geom_line()`.

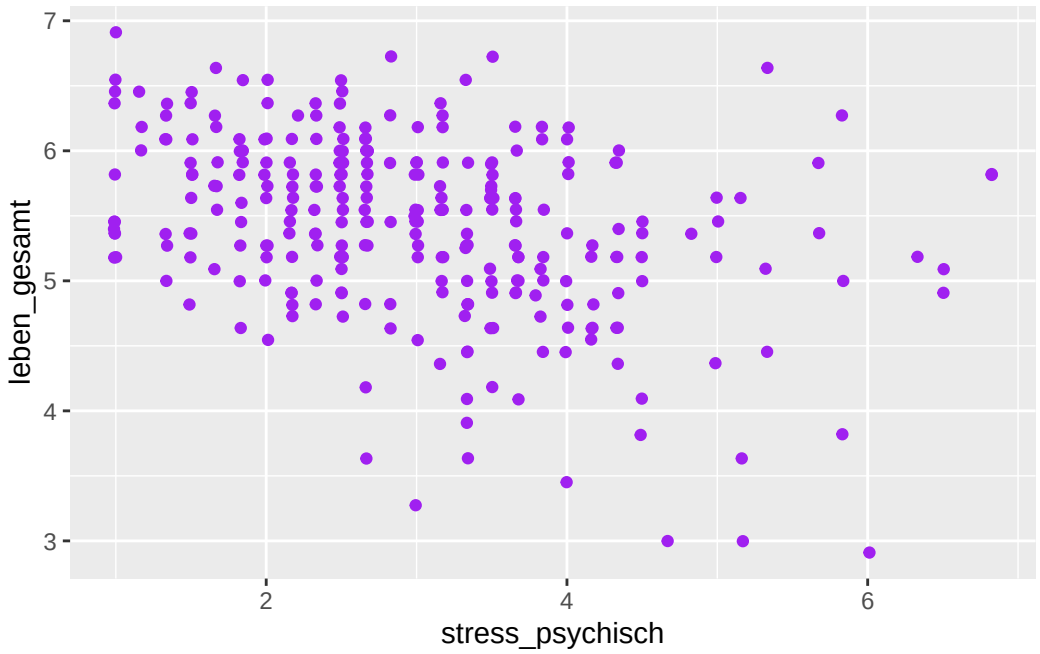
Als Beispiel wollen wir den Zusammenhang zwischen psychischem Stress und Lebenszufriedenheit visualisieren.

```
p <- beispieldaten |>
  select(stress_psychisch, leben_gesamt) |>
  drop_na() |>
  ggplot(mapping = aes(x = stress_psychisch, y = leben_gesamt))

p + geom_point(size = 2, alpha = 0.6)
```



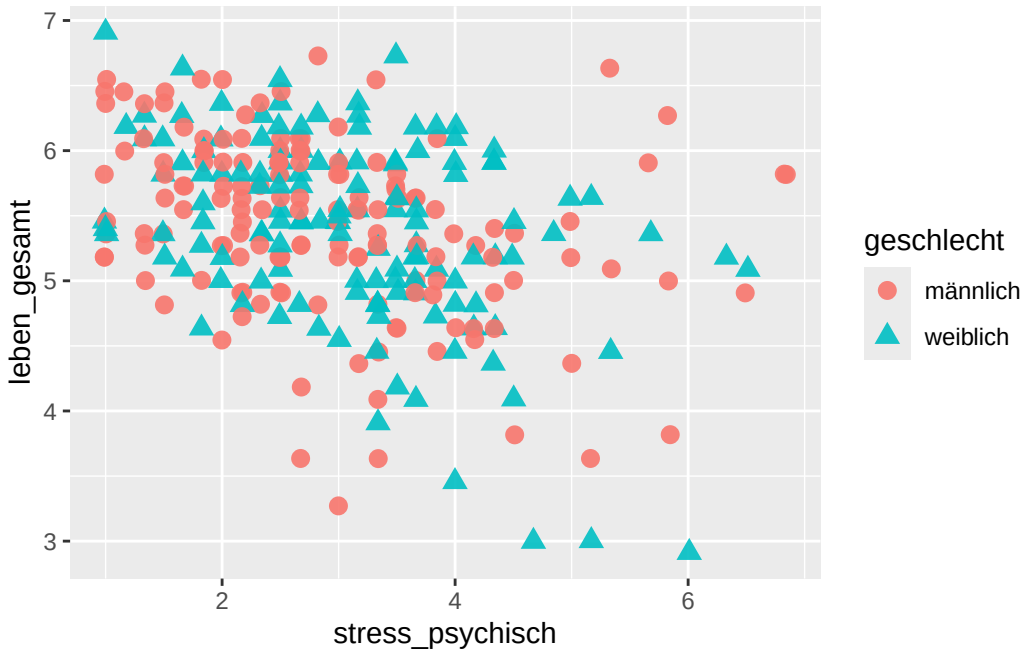
Die `size` und `alpha` Argumente haben wir weiter oben bereits kennengelernt, sowie die Möglichkeit, 'overplotting' mit der Funktion `geom_jitter()` zu vermeiden. Sowohl `geom_jitter()` als auch `geom_point()` haben auch eine `colour` oder ein `color` Argument.

```
p + geom_jitter(colour = "purple")
```



Die Gruppierung anhand einer kategorialen Variablen funktioniert auch hier. Wir verwenden sowohl die Farbe als auch die Form der Punkte, um die Kategorien besser unterscheiden zu können.

```
p <- beispieldaten |>
  select(stress_psychisch, leben_gesamt, geschlecht) |>
  drop_na() |>
  ggplot(mapping = aes(
    x = stress_psychisch,
    y = leben_gesamt,
    color = geschlecht,
    shape = geschlecht
  ))

p + geom_jitter(size = 3, alpha = 0.9)
```



Mit der Funktion `geom_line()` können wir Liniendiagramme erstellen. Als Beispiel wollen wir in einem neuen Dataframe die Mittelwerte der Gesamtnote der Jugendlichen für die verschiedenen Bildungsniveaus des Vaters berechnen, und dann grafisch darstellen. Bevor wir die Noten-Mittelwerte plotten, konvertieren wir den Faktor `bildung_vater` zu einer numerischen Variable.

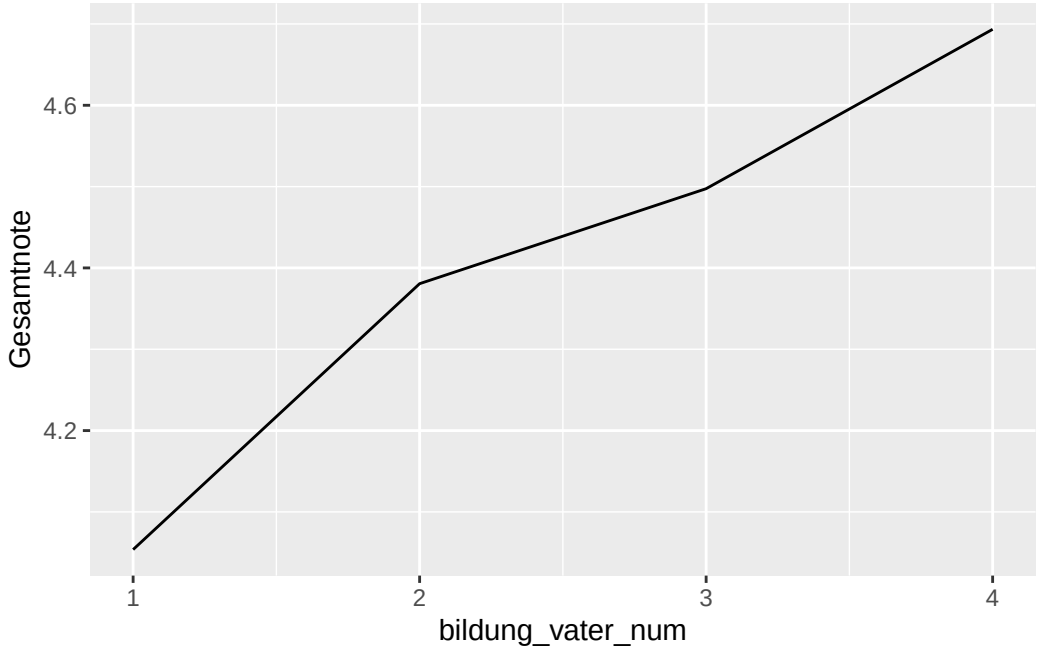
> Vertiefung
>
> Wir könnten `bildung_vater` auch als Faktor verwenden, müssten dann aber eine Gruppierungsvariable für `geom_line()` definieren. In diesem Fall würden wir `group = 1` verwenden, da wir nur eine Gruppe haben: `p + geom_line(group = 1)`

```
bildung_vater <- beispieldaten |>
  select(Gesamtnote, bildung_vater) |>
  drop_na() |>
  group_by(bildung_vater) |>
  summarize(Gesamtnote = mean(Gesamtnote)) |>
  mutate(bildung_vater_num = as.numeric(bildung_vater))

bildung_vater
```

```
# A tibble: 4 x 3
  bildung_vater Gesamtnote bildung_vater_num
```

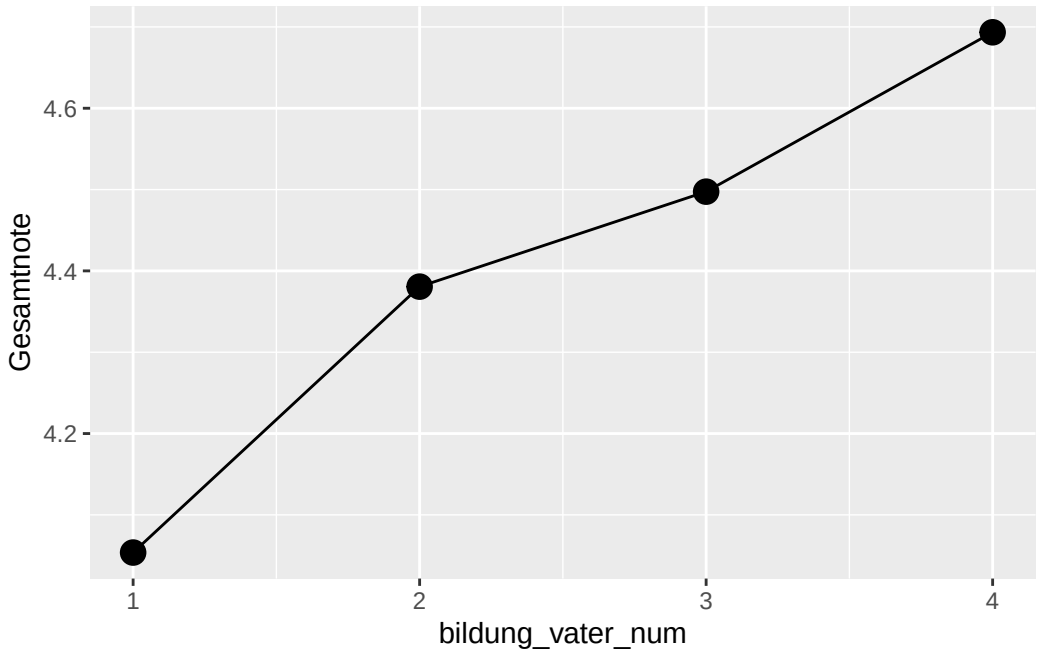
```
  <fct>              <dbl>              <dbl>
1 Hauptschule         4.05                  1
2 Realschule          4.38                  2
3 Abitur              4.50                  3
4 Hochschule          4.69                  4
```

```
p <- bildung_vater |>
  ggplot(aes(
    x = bildung_vater_num,
    y = Gesamtnote
  ))

p + geom_line()
```



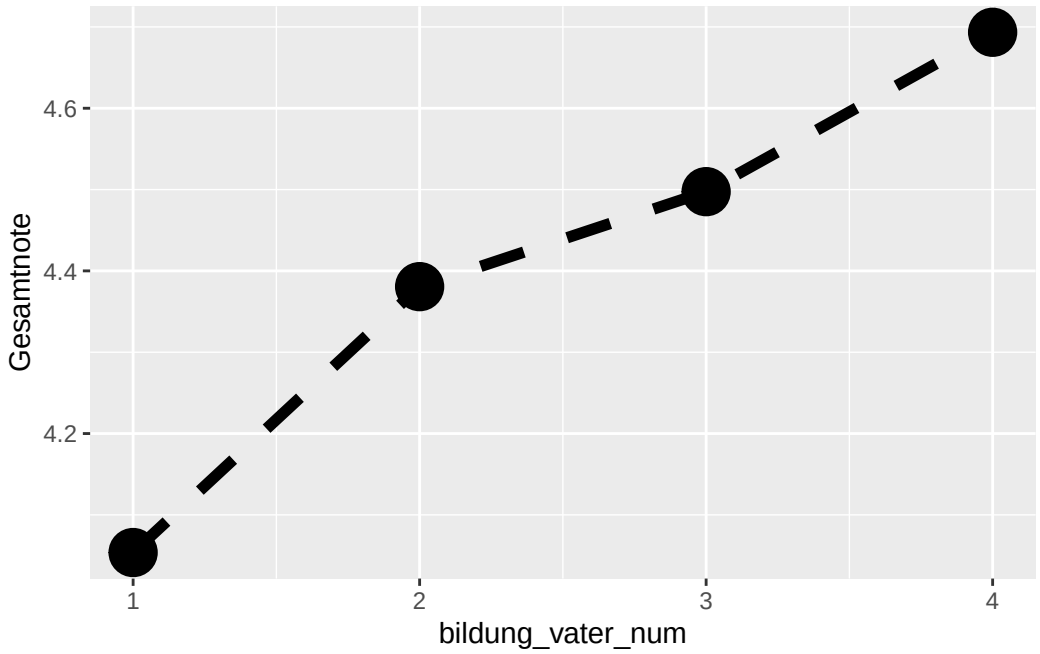
Wir können das Liniendiagramm auch um Punkte ergänzen:

```
p + geom_line() +
  geom_point(size = 4)
```



Auch `geom_line()` hat Argumente, um die Eigenschaften zu ändern. In diesem Fall benützen wir das Argument `linteype`, welches die Werte `"blank"`, `"solid"`, `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"` oder `"twodash"` anehmen kann.

```
p + geom_line(linetype = "dashed", linewidth = 2) +
  geom_point(size = 8)
```



X kategorial und Y kontinuierlich

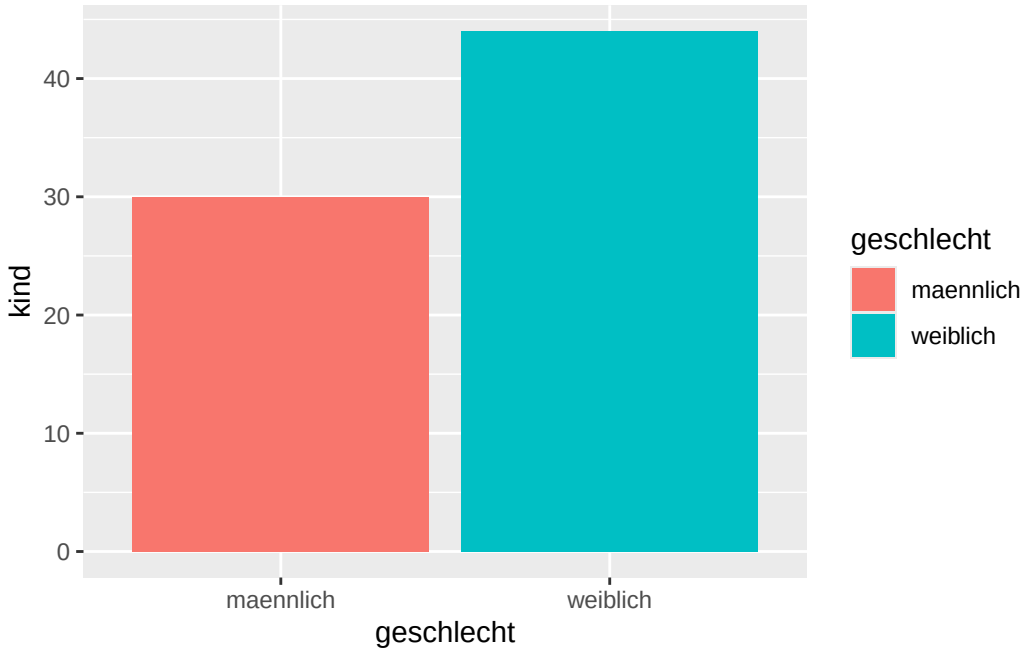
Wenn eine der Variablen kategorial ist, können wir diese, anstatt sie als Gruppierungsvariable zu verwenden, auf einer Achse darstellen.

Beipiele dafür haben wir oben schon gesehen: dort haben wir die Variablen `geschlecht` und `psychischer Stress` dargestellt und die Funktionen `geom_boxplot()` und `geom_violin()` benutzt. Wir können aber auch die Funktion `geom_bar()` für zwei Variablen verwenden. Die Variable auf der Y-Achse wird in diesem Fall für alle Beobachtungen in den Kategieren auf der X-Achse summiert. Da dies keine statistische Transformation benötigt, verwenden wir das Argument `stat = 'identity'`.

Als Beispiel betrachten wir den Kinderwunsch-Datensatz. In diesem wurden $n = 100$ jugendliche Proband:innen gefragt, ob sie später Kinder haben wollen oder nicht (binäre Antwort). In der Variable `kind` wurde die Antwort "Nein" (kein Kinderwunsch) wurde mit dem Wert `0` codiert, die Antwort "Ja" (vorhandener Kinderwunsch) mit dem Wert `1`. Zusätzlich wurde das Geschlecht der Jugendlichen erhoben. Auf der Y-Achse stellen wir die absoluten Häufigkeiten einer "Ja"-Antwort dar, auf der X-Achse das `geschlecht` der Jugendlichen.

```
p <- kinderwunsch |>
  ggplot(aes(
    x = geschlecht,
    y = kind,
    fill = geschlecht
```

```
  ))

p + geom_bar(stat = "identity")
```



Diese absoluten Häufigkeiten sind allerdings schwer zu interpretieren, da in der Stichprobe nicht gleich viele männliche ($n = 44$) und weibliche ($n = 56$) Jugendliche enthalten sind. Zum besseren Verständnis berechnen wir daher zusätzlich noch die relativen Häufigkeiten einer "Ja"-Antwort pro Geschlecht.

```
kinderwunsch |>
  group_by(geschlecht) |>
  summarize(
    n = n(),
    Ja = sum(kind),
    prop_Ja = sum(kind) / n
  )
```

```
# A tibble: 2 x 4
  geschlecht     n    Ja prop_Ja
  <fct>      <int> <dbl>   <dbl>
1 maennlich     44    30   0.682
2 weiblich      56    44   0.786
```

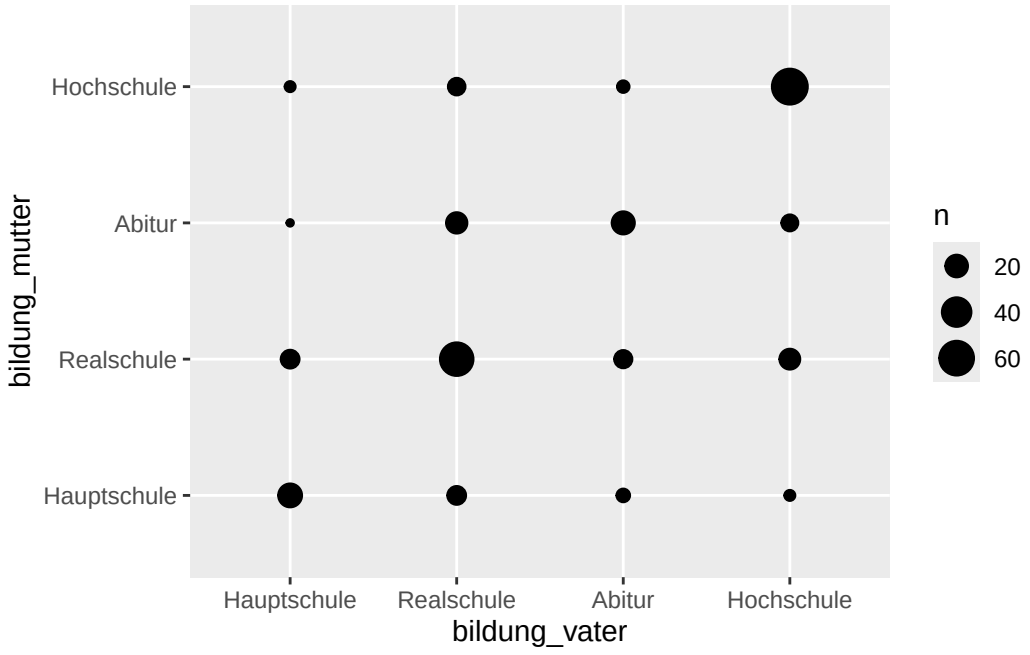
Der Kinderwunsch ist bei weiblichen im Vergleich zu männlichen Jugendlichen also nicht so viel höher ausgeprägt wie es die beiden in der Grafik dargestellten Häufigkeiten vermuten lassen.

X und Y kategorial

Zuletzt können die Variablen sowohl auf der X- als auch auf der Y-Achse kategorial sein. In diesem Fall wäre es sinnvoll, die gemeinsamen Häufigkeiten grafisch darzustellen. Dafür gibt es die Funktion `geom_count()`.

Als Beispiel wollen wir die gemeinsame Häufigkeitsverteilung der Bildung des Vaters und der Bildung der Mutter betrachten.

```
p <- beispieldaten |>
  select(starts_with("bildung")) |>
  drop_na() |>
  ggplot(aes(
    x = bildung_vater,
    y = bildung_mutter
  ))

p + geom_count()
```



`geom_count()` zählt die gemeinsamen Häufigkeiten der Kategorien der beiden Variablen und stellt diese als Durchmesser der Punkte dar.

Hinweis

Die Häufigkeitstabelle erhalten wir mit der Funktion `table()`.

```
table(beispieldaten$bildung_vater, beispieldaten$bildung_mutter)
```

```
             Hauptschule Realschule Abitur Hochschule
  Hauptschule          23         12      2          3
  Realschule           12         55     17         10
  Abitur                5         11     21          4
  Hochschule            3         16      9         65
```

Beispiele

Wir betrachten nun zwei Übungsbeispiele.

Arbeitszufriedenheit im Verlauf

In diesem Beispiel soll der Verlauf der Arbeitszufriedenheit (Skala von 1 bis 15) über die ersten sechs Monate der Anstellung (3 Messzeitpunkte: Anfang, nach 3 Monaten, nach 6 Monaten) zwischen 2 verschiedenen Firmen verglichen werden. In jeder der beiden Firmen wurden jeweils $n = 5$ Angestellte zu ihrer Arbeitszufriedenheit befragt. Es handelt sich um fiktive Daten, die wir zuerst herunterladen und in unserem `data`-Ordner abspeichern müssen: arbeitszufriedenheit.csv

```
library(readr)
arbeitszufriedenheit <- read_csv("data/arbeitszufriedenheit.csv", show_col_types = FALSE)

arbeitszufriedenheit
```

```
# A tibble: 10 x 5
      id firma anfang drei_monate sechs_monate
   <dbl> <dbl>  <dbl>       <dbl>        <dbl>
 1     1     1      9           4            5
 2     2     1      9           4            8
 3     3     1      9           7           14
 4     4     1     10           9            5
```

```
 5     5     1      8           1            3
 6     6     2     10          10           10
 7     7     2      9          11           10
 8     8     2     10          11           12
 9     9     2     12          13           14
10    10     2      9           9            9
```

Die Daten befinden sich im *wide* Format und müssen daher zunächst ins *long* Format gebracht werden. Ausserdem müssen die Variablen `id` und `firma`noch zu Faktoren konvertiert werden:

```
library(tidyverse)
arbeitszufriedenheit_long <- arbeitszufriedenheit |>
  pivot_longer(!c(firma, id),
    names_to = "messzeitpunkt",
    values_to = "arbeitszufriedenheit"
    ) |>
  mutate(id = as.factor(id),
         firma = as.factor(firma),
         messzeitpunkt = as.factor(messzeitpunkt)
         )

arbeitszufriedenheit_long
```

```
# A tibble: 30 x 4
   id    firma messzeitpunkt arbeitszufriedenheit
   <fct> <fct> <fct>                         <dbl>
 1 1     1     anfang                            9
 2 1     1     drei_monate                       4
 3 1     1     sechs_monate                      5
 4 2     1     anfang                            9
 5 2     1     drei_monate                       4
 6 2     1     sechs_monate                      8
 7 3     1     anfang                            9
 8 3     1     drei_monate                       7
 9 3     1     sechs_monate                     14
10 4     1     anfang                           10
# i 20 more rows
```

Zum Plotten benötigen wir die Mittelwerte der Arbeitszufriedenheit pro `firma` und `messzeitpunkt`:

```
arbeitszufriedenheit_means <- arbeitszufriedenheit_long |>
  group_by(firma, messzeitpunkt) |>
  summarize(mean_zufrieden = mean(arbeitszufriedenheit)) |>
  ungroup()
```

```
`summarise()` has grouped output by 'firma'. You can override using the
`.groups` argument.
```

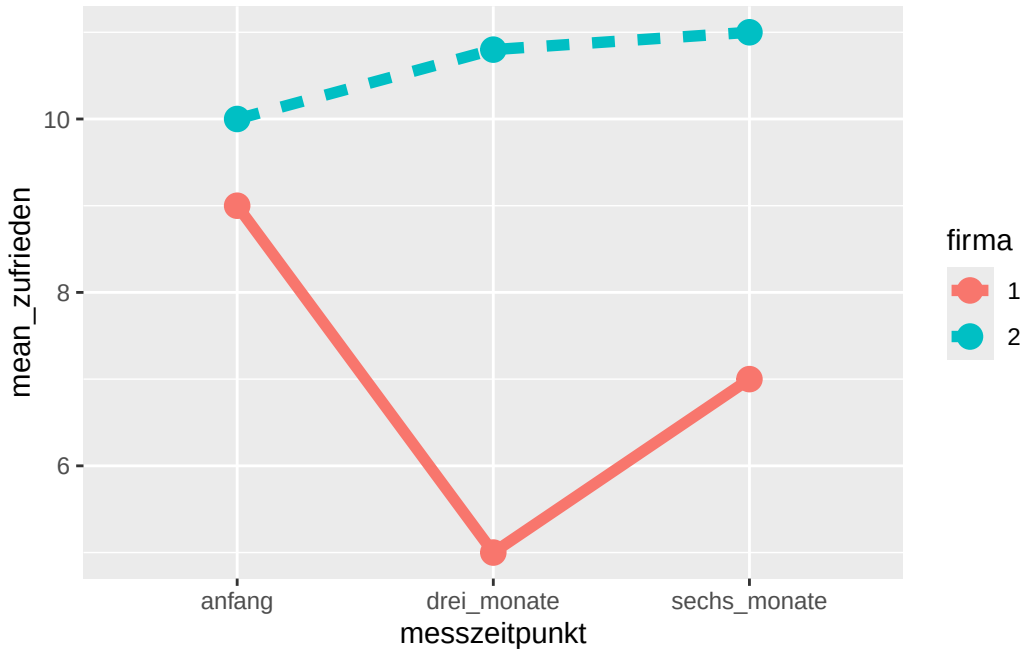
```
arbeitszufriedenheit_means
```

```
# A tibble: 6 x 3
  firma messzeitpunkt mean_zufrieden
  <fct> <fct>                  <dbl>
1 1     anfang                   9
2 1     drei_monate              5
3 1     sechs_monate             7
4 2     anfang                  10
5 2     drei_monate             10.8
6 2     sechs_monate            11
```

Wir stellen nun anhand eines Liniendiagrams die mittlere Arbeitszufriedenheit über die Messzeitpunkte hinweg dar, mit `firma` als Gruppierungsfaktor. `group = firma` ist hier wichtig, die anderen beiden Argumente, `color = firma` und `linetype = firma` sind nur aus ästhetischen Gründen da und könnten auch weggelassen werden.

> Vertiefung
>
> Wie weiter oben schon angedeutet, braucht `ggplot2` das `group` Argument, wenn wir ein Liniendiagramm mit einer kategorialen Variable auf der X-Achse erstellen wollen.

```
p <- arbeitszufriedenheit_means |>
  ggplot(aes(
    x = messzeitpunkt,
    y = mean_zufrieden,
    color = firma,
    linetype = firma,
    group = firma
  ))

p + geom_point(size = 4) +
  geom_line(linewidth = 2)
```



Während in Firma 1 die Arbeitszufriedenheit nach drei Monaten stark abgefallen ist und sich nach sechs Monaten wieder etwas erholt hat, ist sie in Firma 2 über alle drei Messzeitpunkte hoch bzw. steigt sogar etwas an.

Liniendiagramme werden oft benutzt, um den zeitlichen Verlauf einer Variablen darzustellen. Dies bedeutet, dass wir auf der X-Achse die Zeit darstellen, wie in diesem Beispiel. Meistens wird Zeit jedoch als kontinuierliche Variable verwendet und nicht, wie hier, als Faktor.

Wide vs. Long: Bildung der Eltern

Anhand des nächsten Beipiels betrachten wir den Unterschied zwischen dem *long* und dem *wide* Format. Wir haben, als wir die gemeinsame Häufigkeitsverteilung der Bildung des Vaters und der Mutter dargestellt haben, `bildung_vater` und `bildung_mutter` als separate Variablen verwendet, um sie auf separaten Achsen darzustellen. Wir könnten jedoch auch `bildung_vater` und `bildung_mutter` als Stufen eines Messwiederholungsfaktors `elternteil` (`key`) zusammenfassen, und die Bildungsniveaus als Messvariable `bildung` (`value`), d.h. als `key`/`value` Paar. Dies machen wir, wenn wir `bildung` als Variable auf einer Achse verwenden wollen und `elternteil` als Gruppierungsvariable.

Dies ist vielleicht nicht ganz einfach zu verstehen, deshalb betrachten wir gleich ein konkretes Beispiel. Wir wollen nun, ähnlich wie oben, die mittlere Schulnote der Jugendlichen für die verschiedenen Bildungsniveaus der Eltern grafisch darstellen. Diesmal machen wir dies für beide Elternteile. Wir wollen jedoch unterschiedliche Linien für *Vater* und *Mutter*. Nun ist es wichtig, dass wir einen *long* Datensatz bilden.

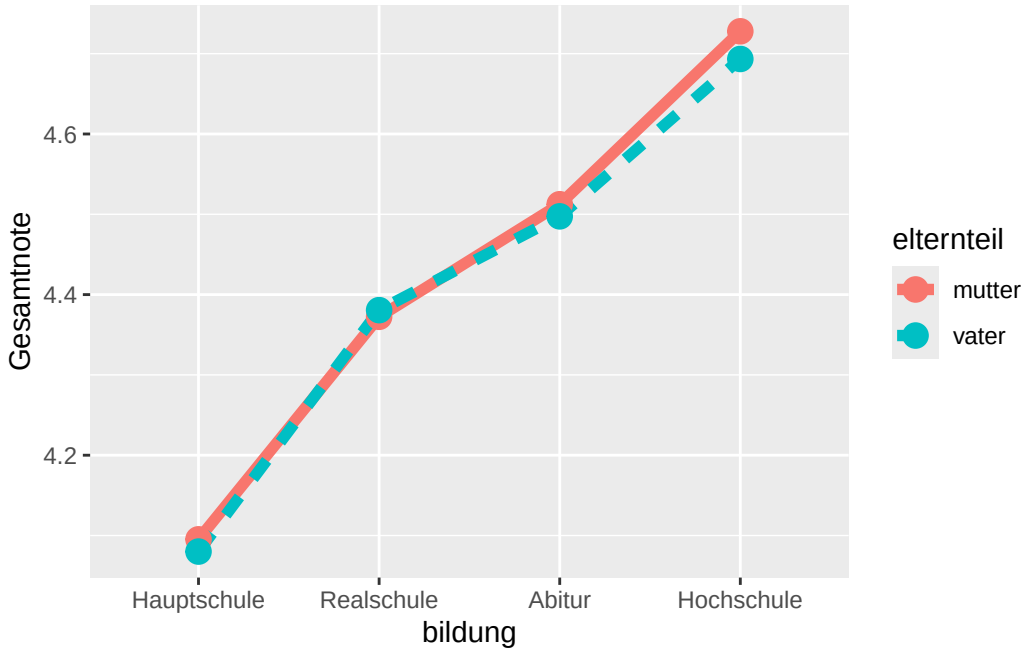
```
bildung <- beispieldaten |>
  # Variablen auswählen
  select(Gesamtnote, bildung_vater, bildung_mutter) |>
  # Fehlende Werte ausschliessen
  drop_na() |>
  # wide zu long
  pivot_longer(!Gesamtnote, names_to = "elternteil", values_to = "bildung") |>
  # Präfix bildung_ bei elternteil-Variable entfernen
  mutate(elternteil = str_replace(elternteil, ".*_", "")) |>
  # zu Faktoren konvertieren
  mutate(
    elternteil = factor(elternteil, levels = c("mutter", "vater")),
    bildung = factor(bildung, levels = c(
      "Hauptschule", "Realschule",
      "Abitur", "Hochschule"
    ))
  ) |>
  # Gruppieren: zuerst Eltern, dann Bildungsniveaus
  group_by(elternteil, bildung) |>
  # Mittlere Note berechnen
  summarize(Gesamtnote = mean(Gesamtnote)) |>
  ungroup()
```

```
`summarise()` has grouped output by 'elternteil'. You can override using the
`.groups` argument.
```

```
bildung
```

```
# A tibble: 8 x 3
  elternteil bildung     Gesamtnote
  <fct>      <fct>            <dbl>
1 mutter     Hauptschule       4.10
2 mutter     Realschule        4.37
3 mutter     Abitur            4.51
4 mutter     Hochschule        4.73
5 vater      Hauptschule       4.08
6 vater      Realschule        4.38
7 vater      Abitur            4.50
8 vater      Hochschule        4.69
```

```r
p <- bildung |>
  ggplot(aes(
    x = bildung,
    y = Gesamtnote,
    colour = elternteil,
    linetype = elternteil,
    group = elternteil
  ))

p + geom_line(linewidth = 2) +
  geom_point(size = 4)
```



Hinweis

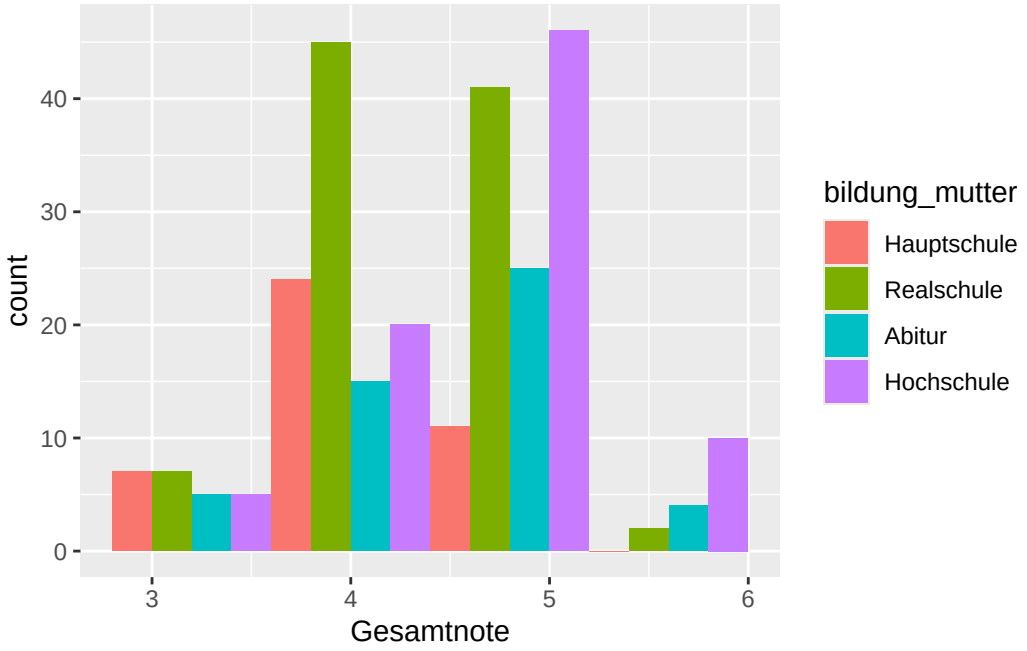
An diesem Beispiel wir deutlich, dass ein grosser Teil der Arbeit mit `ggplot2` darin besteht, die Daten zuerst ins 'richtige' Format zu bringen. Wenn dies getan ist, ist es jedoch relativ einfach, eine Grafik zu erstellen. Dies ist ein nicht zu unterschätzender Vorteil von `ggplot2`.

5.5 Facets

Bisher haben wir Gruppierungsvariablen dazu benutzt, um unterschiedliche Farben/Formen/Linien für die Kategorien der Gruppierungsvariable innerhalb eines Plots zu erzeugen. Manchmal ist dies jedoch zu unübersichtlich.

Wollen wir zum Beispiel ein Histogram der Schulnoten erstellen, und zwar für jede Stufe der Bildung der Mutter, dann wäre die Grafik völlig überladen.

```
p <- beispieldaten |>
  select(Gesamtnote, bildung_mutter) |>
  drop_na() |>
  ggplot(mapping = aes(
    x = Gesamtnote,
    fill = bildung_mutter
  ))

p + geom_histogram(
  binwidth = 0.8,
  position = "dodge"
)
```

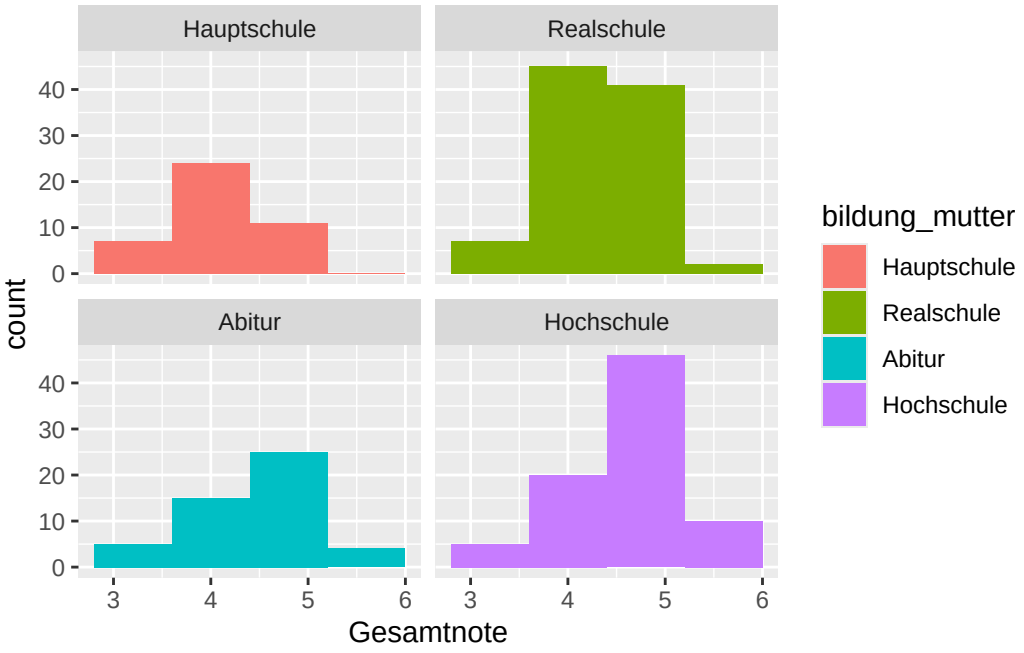


Eine offensichtliche Lösung wäre, die Histogramme für die Bildungsniveaus der Mutter in separaten Grafiken darzustellen.

Genau dies können wir mit den Funktionen `facet_wrap()` und `facet_grid()` machen.

Mit `facet_wrap()` erstellen wir so eine Grafik für jede Kategorie der Gruppierungsvariable:

```
p <- beispieldaten |>
  select(Gesamtnote, bildung_mutter) |>
  drop_na() |>
  ggplot(mapping = aes(
    x = Gesamtnote,
    fill = bildung_mutter
  )) +
  facet_wrap(~bildung_mutter)

p + geom_histogram(binwidth = 0.8)
```

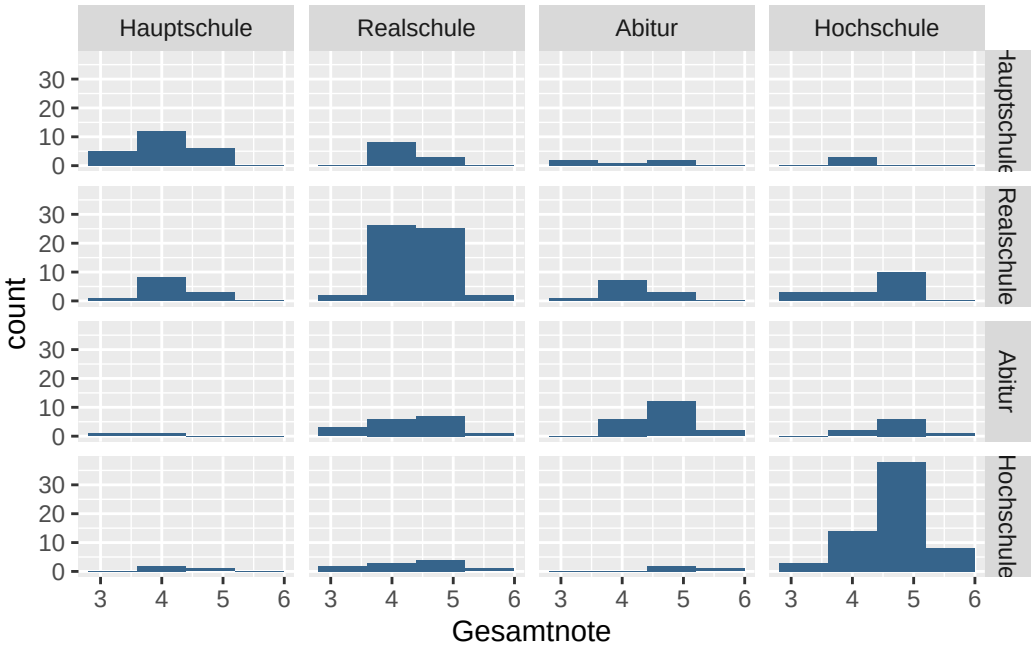


> **Hinweis**
>
> Das ~ (Tilde) Zeichen bedeutet hier ungefähr: in Abhängigkeit oder als "Funktion" von.

Wenn wir zwei Gruppierungsvariablen haben, können wir mit `facet_grid()` ein Raster erzeugen.

```
p <- beispieldaten |>
  select(Gesamtnote, bildung_mutter, bildung_vater) |>
```

```
  drop_na() |>
  ggplot(mapping = aes(x = Gesamtnote)) +
  facet_grid(bildung_mutter ~ bildung_vater)

p + geom_histogram(
  binwidth = 0.8,
  fill = "steelblue4"
)
```



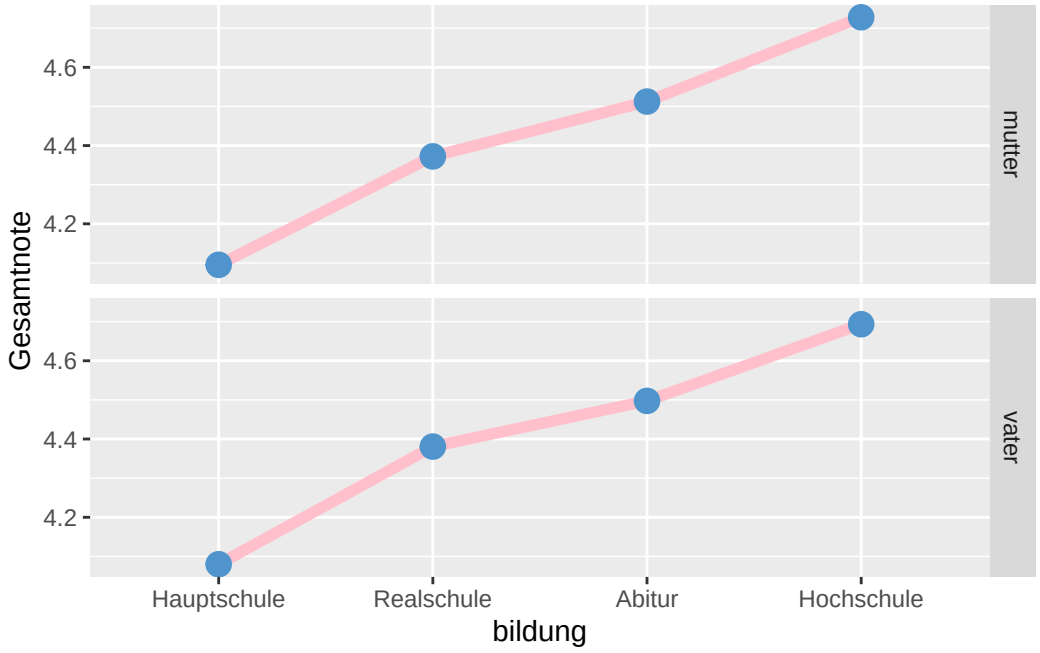
Hier werden die Stufen von `bildung_mutter` in den Zeilen dargestellt, die Stufen von `bildung_vater` in den Spalten.

Als zweites Beispiel können wir den Notenschnitt in Abhängigkeit der Bildung der Eltern als Liniendiagram darstellen, und zwar in separaten Plots für Väter und Mütter getrennt. Anhand dieses Beispiels sehen wir, dass wir `facet_grid()` auch dann verwenden können, wenn wir nur eine Gruppierungsvariable haben, und zwar um die Anzahl Zeilen bzw. Spalten festzulegen.

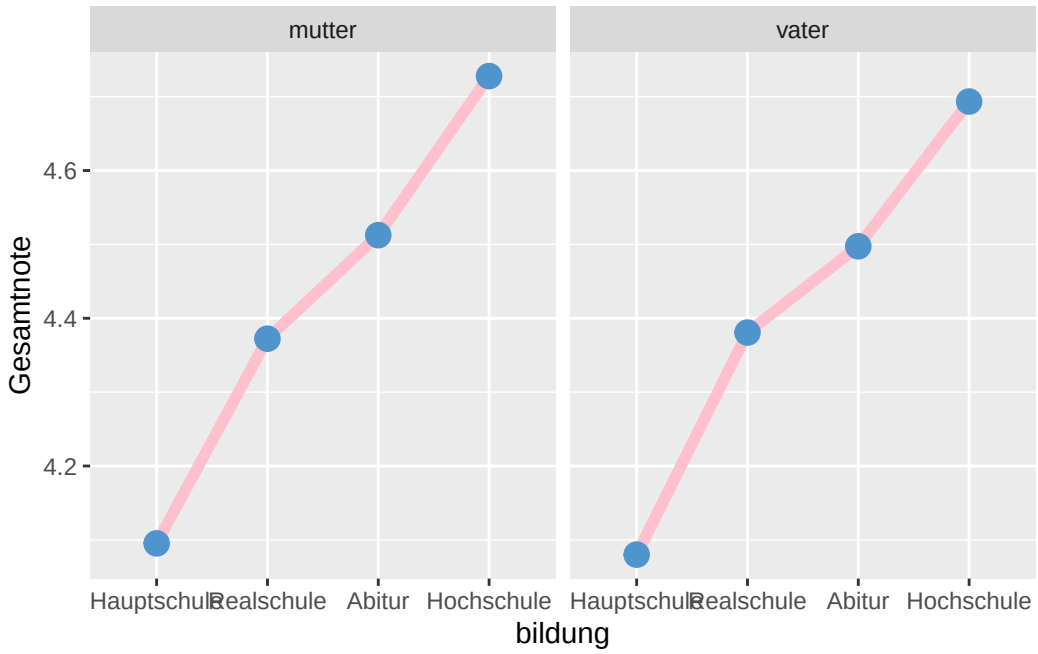
Wenn wir die Gruppierung in den Zeilen haben wollen, schreiben wir `facet_grid(Gruppierungsvariable ~ .)`, wenn sie in den Spalten wollen, schreiben wir `facet_grid(. ~ Gruppierungsvariable)`. Der Punkt `.` bedeutet hier, dass wir für die Zeilen/Spalten keine Gruppierungsvariable verwenden.

```
p <- bildung |>
  ggplot(aes(
```

```
    x = bildung,
    y = Gesamtnote,
    group = elternteil
  ))

p + geom_line(linewidth = 2, color = "pink") +
  geom_point(size = 4, color = "steelblue3") +
  # Stufen von 'elternteil' in die Zeilen
  facet_grid(elternteil ~ .)
```




```
p <- bildung |>
  ggplot(aes(
    x = bildung,
    y = Gesamtnote,
    group = elternteil
  ))

p + geom_line(linewidth = 2, color = "pink") +
  geom_point(size = 4, color = "steelblue3") +
  # Stufen von 'elternteil' in die Spalten
  facet_grid(. ~ elternteil)
```



5.6 Farben und Themes

Bisher hat `ggplot2` automatisch für uns Farben gewählt, wenn wir Farben für eine Gruppierung verlangt haben. Die Standard Farbpalette ist jedoch für Farbenblinde äusserst schlecht geeignet. Es gibt viele Farbpaletten, welche wir verwenden könnten.

Wir definieren hier jedoch eine eigene, für Farbenblinde geeignete Farbpalette.

```
palette <- c(
  "#000000", "#E69F00",
  "#56B4E9", "#009E73",
  "#F0E442", "#0072B2",
  "#D55E00", "#CC79A7"
)
```

Wir definieren hier also einen Vektor von acht Hex Codes. Folglich dürfte unsere Gruppierungsvariable nicht mehr als acht Kategorien haben.

> Hinweis
>
> Sie können hier natürlich eine eigene Farbpalette erstellen, indem Sie entweder Hex-Codes oder Farbennamen angeben.

Die Palette verwenden wir so:

- um Formen auszufüllen, verwenden wir

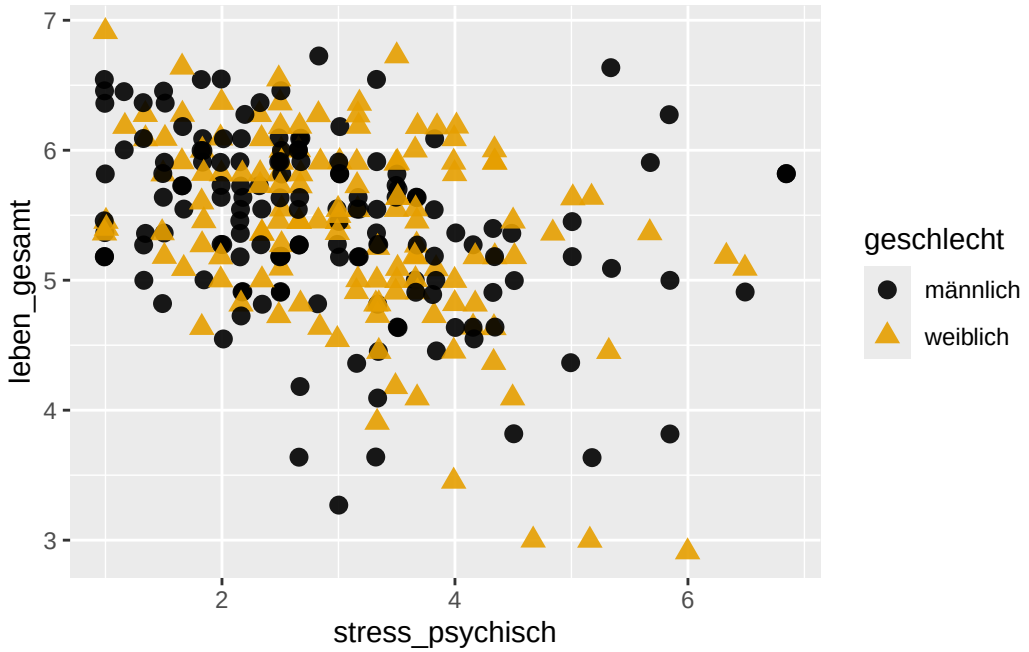
```
scale_fill_manual(values = palette)
```

- für Linien und Punkte verwenden wir:

```
scale_colour_manual(values = palette)
```

Als Beispiel plotten wir nochmals den Zusammenhang zwischen `stress_psychisch` und `leben_gesamt`, diesmal mit unserer Farbpalette.

```
p <- beispieldaten |>
  select(stress_psychisch, leben_gesamt, geschlecht) |>
  drop_na() |>
  ggplot(mapping = aes(
    x = stress_psychisch,
    y = leben_gesamt,
    color = geschlecht,
    shape = geschlecht
  ))

p + geom_jitter(size = 3, alpha = 0.9) +
  scale_colour_manual(values = palette)
```

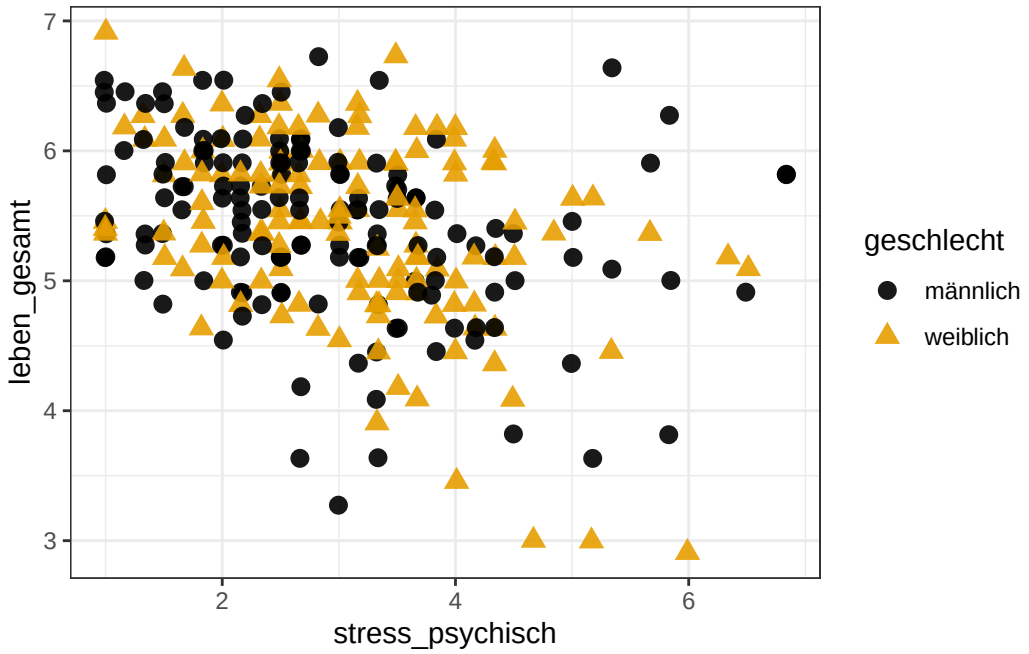


Wir könnten die Farben auch so 'von Hand' bestimmen:

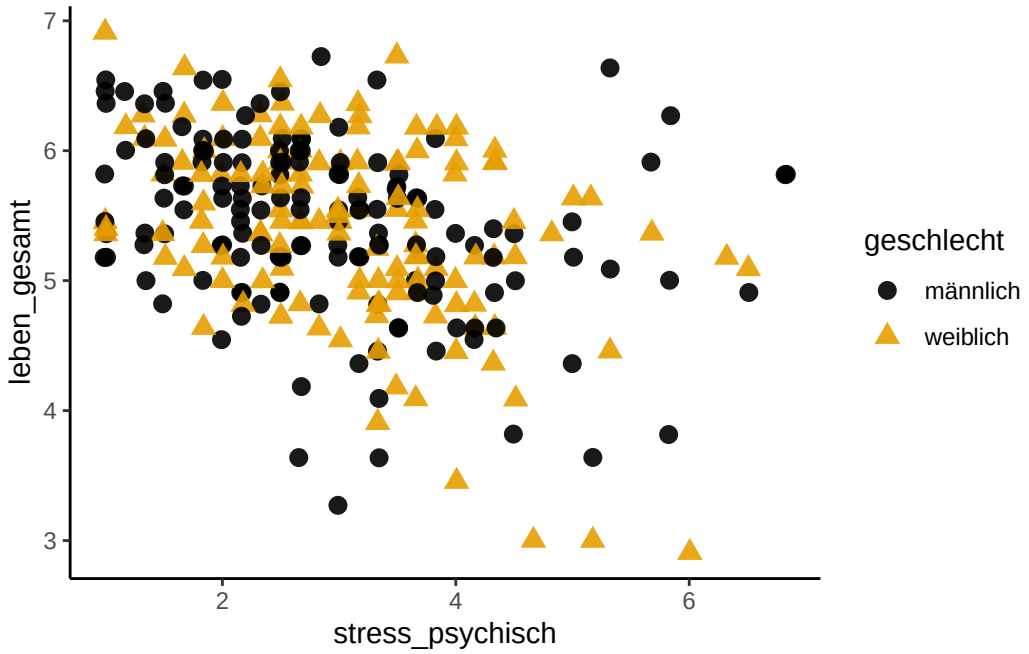
```
p + geom_jitter(size = 3, alpha = 0.9) +
  scale_colour_manual(values = c("pink2", "steelblue3"))
```

Ein weiterer heikler Punkt ist der graue Hintergrund, den `ggplot2` automatisch wählt. Diesen können wir am einfachsten ändern, indem wir ein `theme` definieren. Es gibt zwei solcher `themes`, welche einen weissen Hintergrund haben: `theme_bw()` und `theme_classic()`. Diese unterscheiden sich darin, dass `theme_classic()` keine *grid lines* zeichnet, sondern nur die linke und die untere Achse.

```
p <- beispieldaten |>
  select(stress_psychisch, leben_gesamt, geschlecht) |>
  drop_na() |>
  ggplot(mapping = aes(
    x = stress_psychisch,
    y = leben_gesamt,
    color = geschlecht,
    shape = geschlecht
  ))

p + geom_jitter(size = 3, alpha = 0.9) +
  scale_colour_manual(values = palette) +
  theme_bw()
```




```
p <- beispieldaten |>
  select(stress_psychisch, leben_gesamt, geschlecht) |>
  drop_na() |>
  ggplot(mapping = aes(
    x = stress_psychisch,
    y = leben_gesamt,
    color = geschlecht,
    shape = geschlecht
  ))

p + geom_jitter(size = 3, alpha = 0.9) +
  scale_colour_manual(values = palette) +
  theme_classic()
```



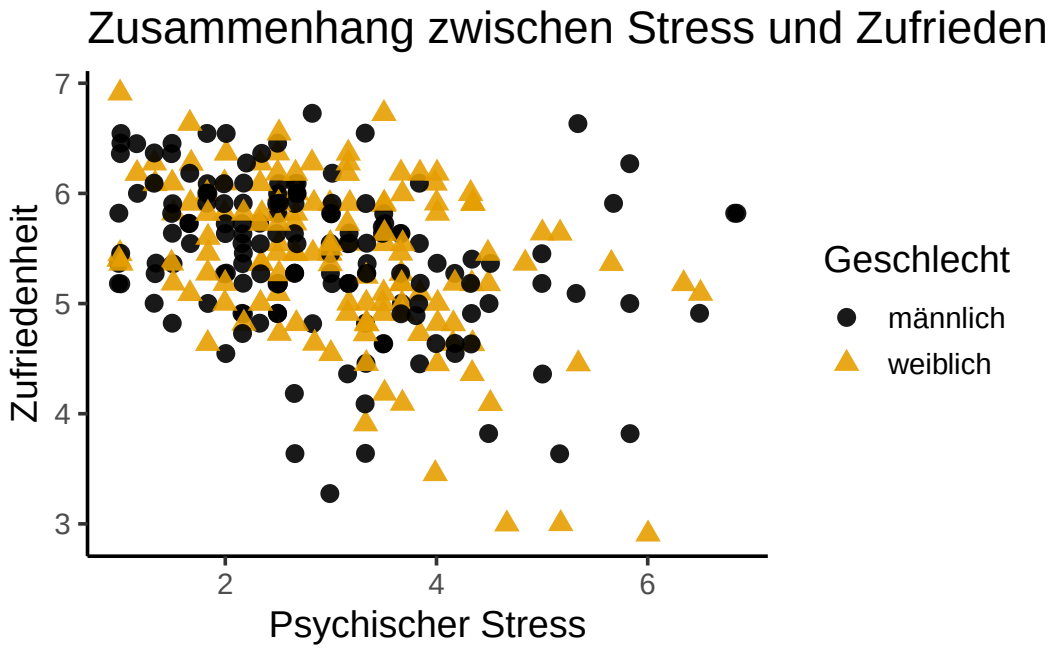
5.7 Beschriftungen

Wir können nun auch noch mit `xlab()` und `ylab()` die Beschriftungen der X/Y-Achsen ändern, und mit der Funktion `ggtitle()` dem Plot einen Titel geben. Mit der Funktion `labs()` können wir zusätzlich noch den Titel der Legende ändern.

Zuletzt wollen wir auch die Schriftgrösse ändern, da die Standardgrösse oft zu klein erscheint. Dies erreichen wir mit dem Argument `base_size = SCHRIFTGRÖSSE)` der `theme_` Funktionen.

```
p <- beispieldaten |>
  select(stress_psychisch, leben_gesamt, geschlecht) |>
  drop_na() |>
  ggplot(mapping = aes(
    x = stress_psychisch,
    y = leben_gesamt,
    color = geschlecht,
    shape = geschlecht
  ))

p + geom_jitter(size = 3, alpha = 0.9) +
  scale_colour_manual(values = palette) +
  theme_classic(base_size = 14) +
```

```
  ggtitle("Zusammenhang zwischen Stress und Zufriedenheit") +
  xlab("Psychischer Stress") +
  ylab("Zufriedenheit") +
  # "Geschlecht" als Titel der color- und shape-Legende
  labs(
    color = "Geschlecht",
    shape = "Geschlecht"
  )
```



5.8 Grafiken speichern

Wenn wir eine schöne Grafik erstellt haben, wollen wir sie natürlich speichern. Dies können wir mit der Funktion `ggsave()` machen. Die Funktion nimmt als Argumente den Dateinamen, den Namen des Plot-Objekts und weitere Eigenschaften, wie die gewünschte Höhe und Breite des Plots. Diese können z.B. in "cm" angebenen werden, mit dem Argument `units = "cm"`. Um die Grafik zu speichern, müssen wir unseren fertigen Plot vorher einer Variablen/einem Objekt zuweisen:

```
p <- beispieldaten |>
  select(stress_psychisch, leben_gesamt, geschlecht) |>
  drop_na() |>
  ggplot(mapping = aes(
```

```
    x = stress_psychisch,
    y = leben_gesamt,
    color = geschlecht,
    shape = geschlecht
  ))

# Wir nennen die Grafik 'my_plot'
my_plot <- p + geom_jitter(size = 3, alpha = 0.9) +
  scale_colour_manual(values = palette) +
  theme_classic(base_size = 14) +
  ggtitle("Zusammenhang zwischen Stress und Zufriedenheit") +
  xlab("Psychischer Stress") +
  ylab("Zufriedenheit") +
  labs(
    color = "Geschlecht",
    shape = "Geschlecht"
  )
```

`my_plot` kann nun gespeichert werden:

```
ggsave(
  filename = "my_plot.png",
  plot = my_plot
)
```

Die Grafik kann auch in den Formaten eps, ps, tex, pdf, jpeg, tiff, bmp, svg und wmf gespeichert werden. Dazu muss lediglich die Endung `.png` ersetzt werden, beispielsweise durch `.pdf`. Je nach Anwendungszweck lohnt es sich, auf ein anderes Dateiformat zu setzen. Insbesondere wenn die Grafiken vergrössert werden sollen, lohnt es sich, auf ein Vektorgrafikformat wie `.svg` oder `.wmf` zu setzen.

5.9 Übungsaufgaben

In diesen Übungsaufgaben wollen wir die Zusammenhänge zwischen den sechs Selbstwirksamkeitsskalen im Datensatz `beispieldaten` und der psychischen Belastung untersuchen (Aufgaben 1 - 3). Zudem wollen wir Geschlechtsunterschiede in Bezug auf die durchschnittliche Ausprägung von Selbstwirksamkeit und psychischen sowie somatischen Stressymptomen darstellen (Aufgaben 4 und 5).

Aufgabe 1

a) Bilden Sie einen Subdatensatz `selbstwirksamkeit_wide`, der nur die für die Aufgaben 1 - 3 relevanten Variablen des Datensatzes enthält (`ID`, `geschlecht`, `stress_psychisch`, `swk_neueslernen`, `swk_lernregulation`, `swk_motivation`, `swk_durchsetzung`, `swk_sozialkomp`, `swk_beziehung`) und entfernen Sie fehlende Werte.

Lösung

```
selbstwirksamkeit_wide <- beispieldaten |>
  select(ID, geschlecht, stress_psychisch, starts_with("swk_")) |>
  drop_na()

selbstwirksamkeit_wide
```

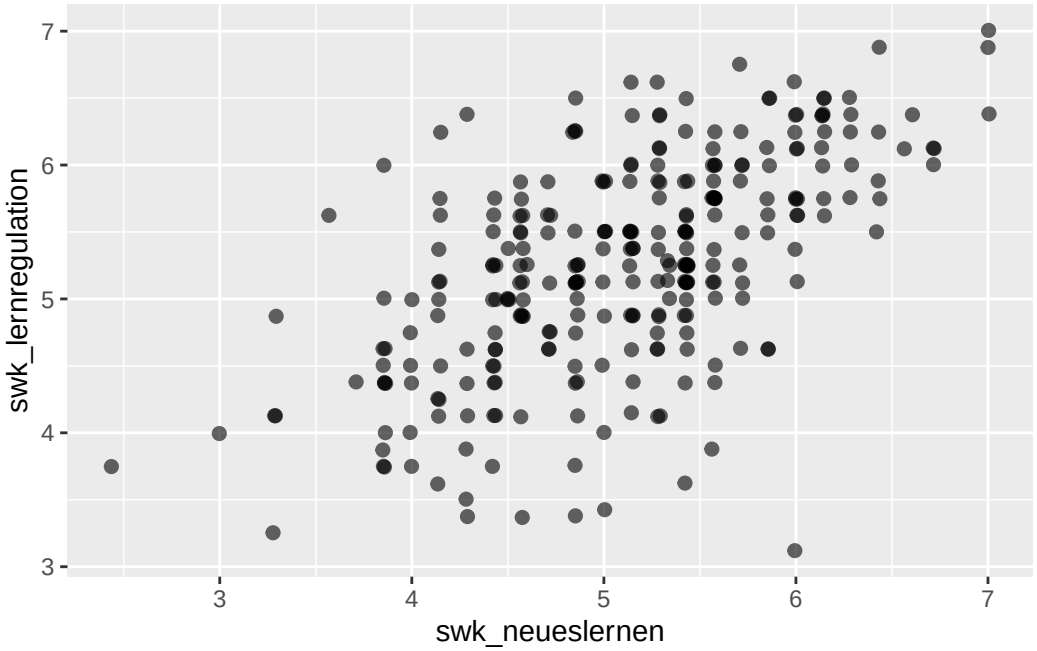
```
# A tibble: 283 x 9
      ID geschlecht stress_psychisch swk_neueslernen swk_lernregulation
   <dbl> <fct>                 <dbl>           <dbl>               <dbl>
 1     1 weiblich               1.67            4.57                5.5
 2     2 männlich               3.5             5                   4
 3    10 weiblich               3.67            5                   4.88
 4    11 weiblich               1.5             4.57                5.38
 5    12 weiblich               2.5             5.33                5.29
 6    14 männlich               1               4.86                4.5
 7    15 männlich               2.5             4                   4.38
 8    16 weiblich               3.5             4.83                6.25
 9    17 männlich               1.67            6.14                5.62
10    18 männlich               2.5             5.14                4.62
# i 273 more rows
# i 4 more variables: swk_motivation <dbl>, swk_durchsetzung <dbl>,
#   swk_sozialkomp <dbl>, swk_beziehung <dbl>
```

b) Versuchen Sie zunächst, Zusammenhänge der sechs Selbstwirksamkeitsskalen untereinander zu entdecken, indem Sie diese grafisch darstellen. Erstellen Sie dazu für jedes Variablenpaar einen Scatterplot (Streudiagramm). Insgesamt müssen also 15 Scatterplots erstellt werden.

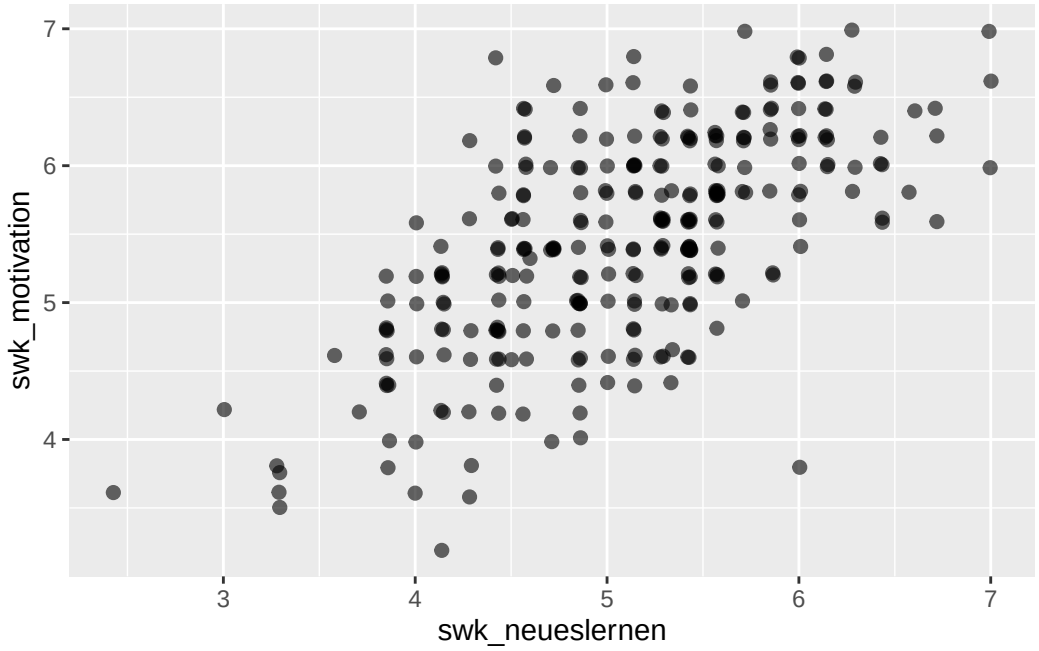
Lösung

```
p <- selbstwirksamkeit_wide |>
  ggplot()
```

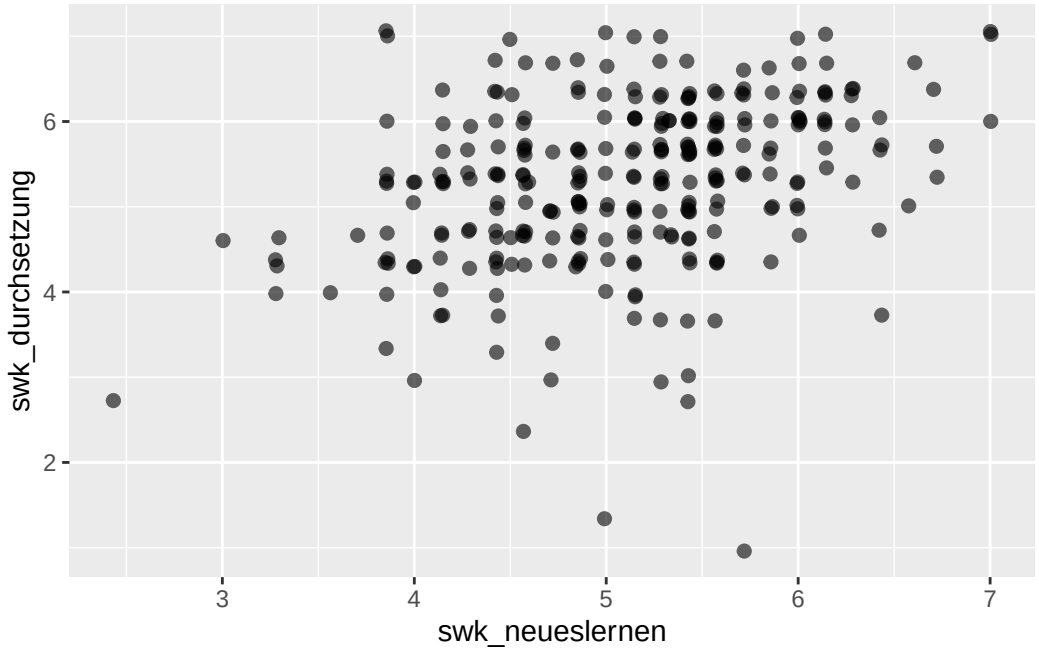
```
p + geom_jitter(aes(x = swk_neueslernen, y = swk_lernregulation),
                alpha = 0.6, size = 2)
```



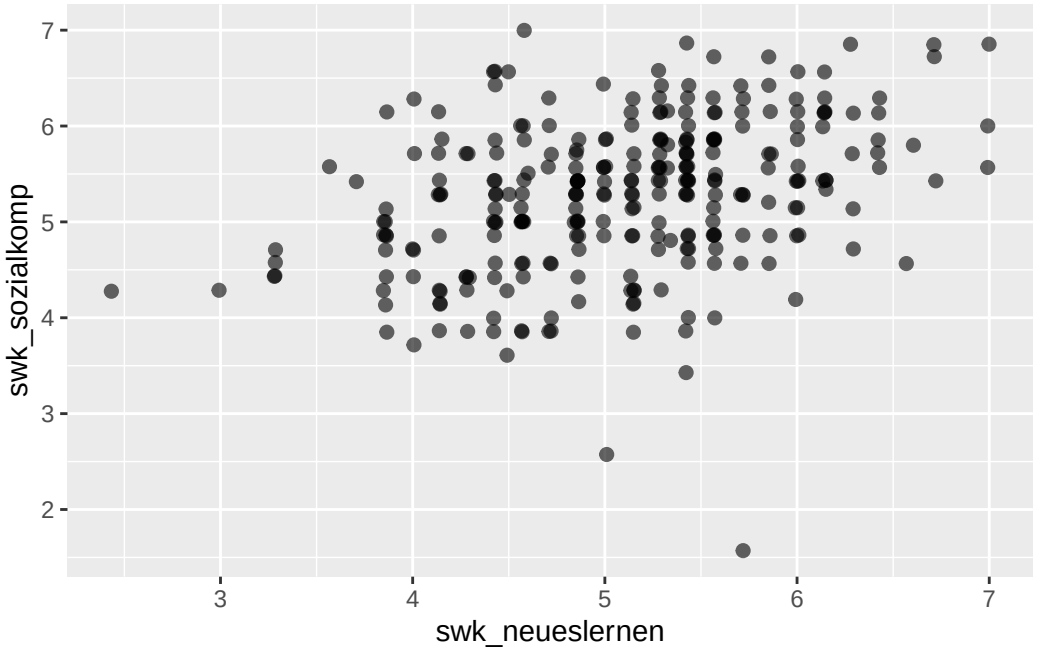

```
p + geom_jitter(aes(x = swk_neueslernen, y = swk_motivation),
                alpha = 0.6, size = 2)
```



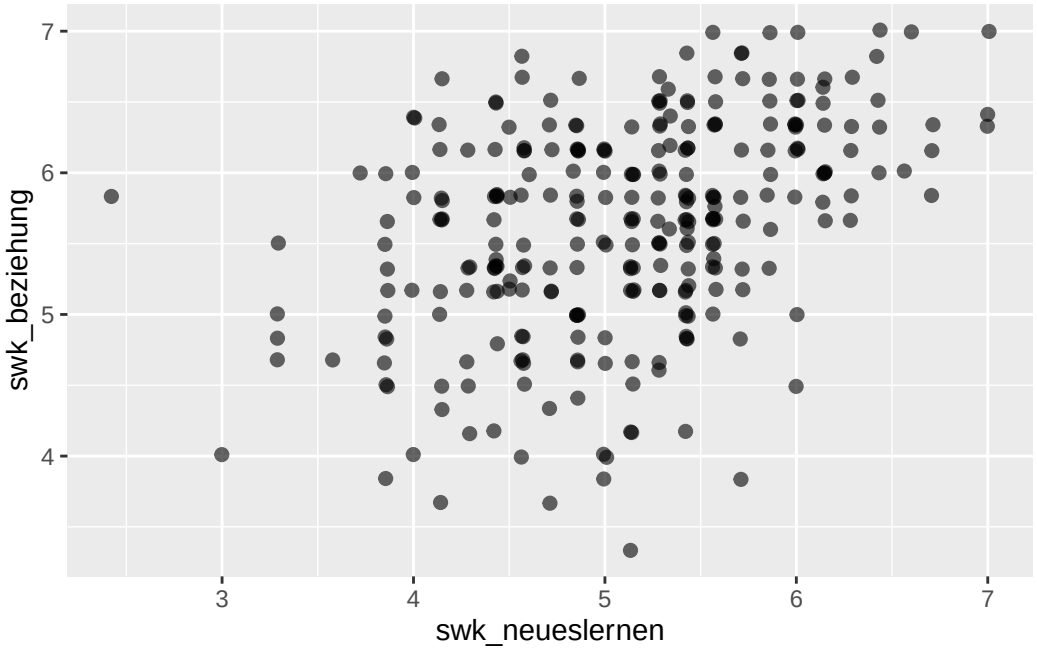

```
p + geom_jitter(aes(x = swk_neueslernen, y = swk_durchsetzung),
                alpha = 0.6, size = 2)
```



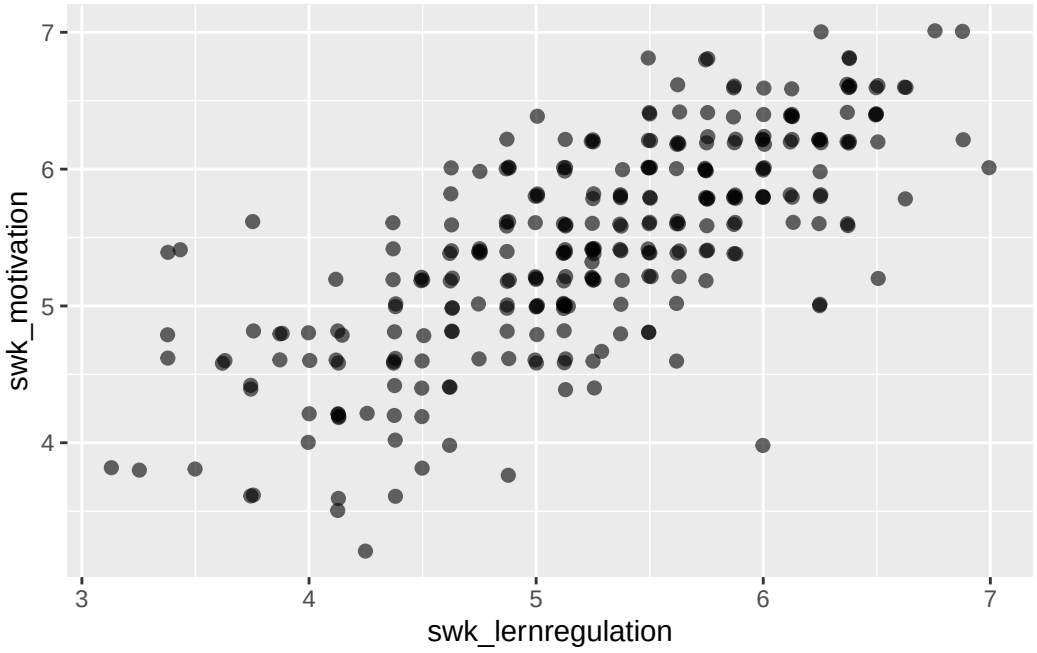
```
p + geom_jitter(aes(x = swk_neueslernen, y = swk_sozialkomp),
                alpha = 0.6, size = 2)
```



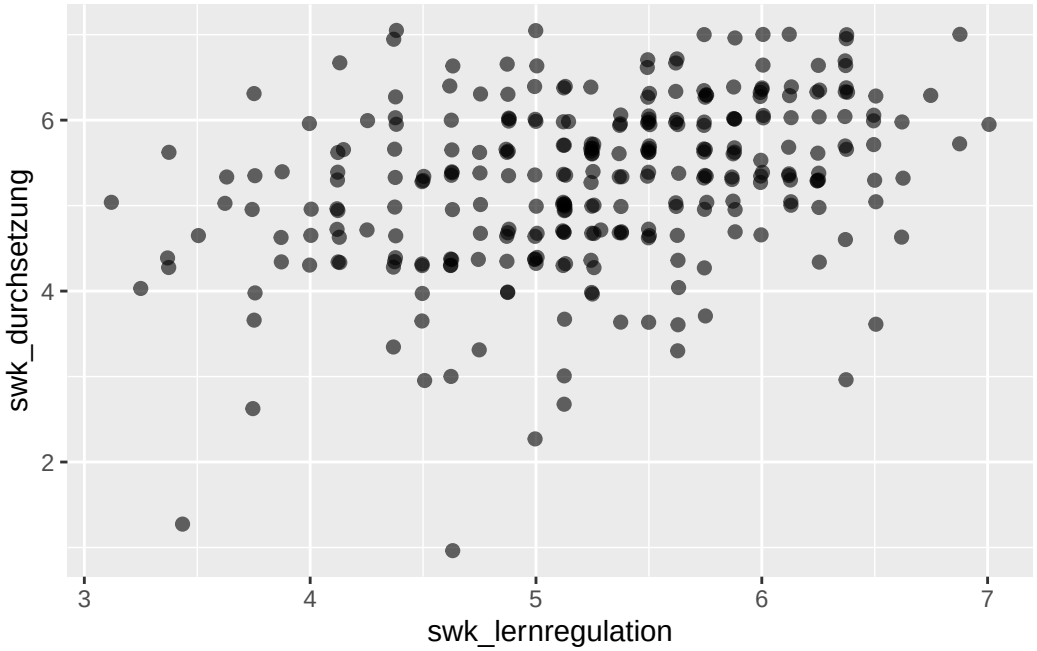

```
p + geom_jitter(aes(x = swk_neueslernen, y = swk_beziehung),
                alpha = 0.6, size = 2)
```



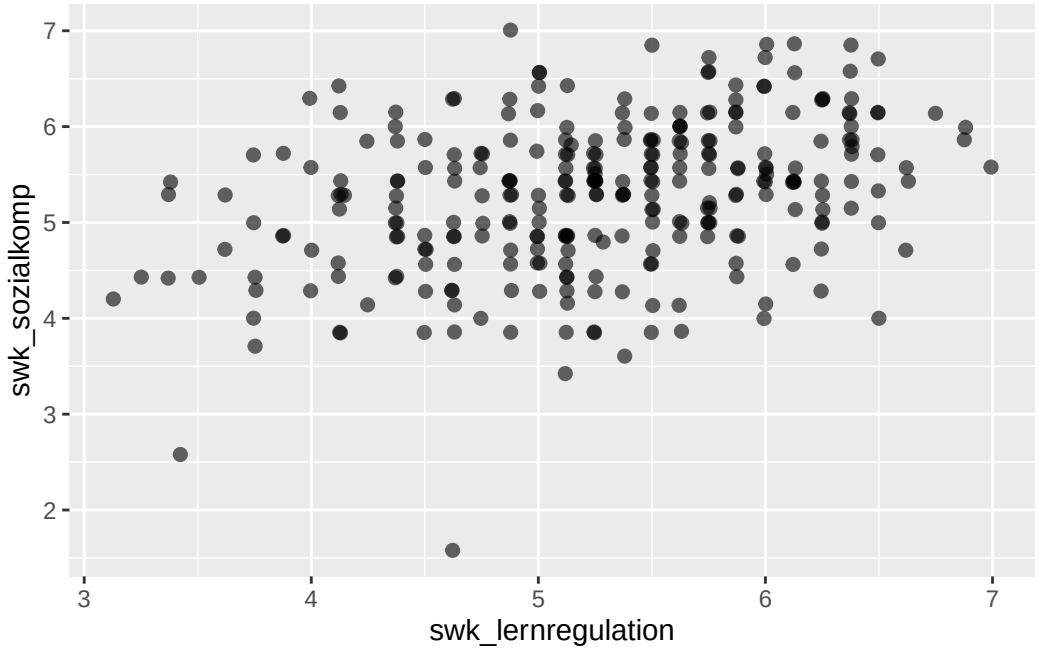

```
p + geom_jitter(aes(x = swk_lernregulation, y = swk_motivation),
                alpha = 0.6, size = 2)
```



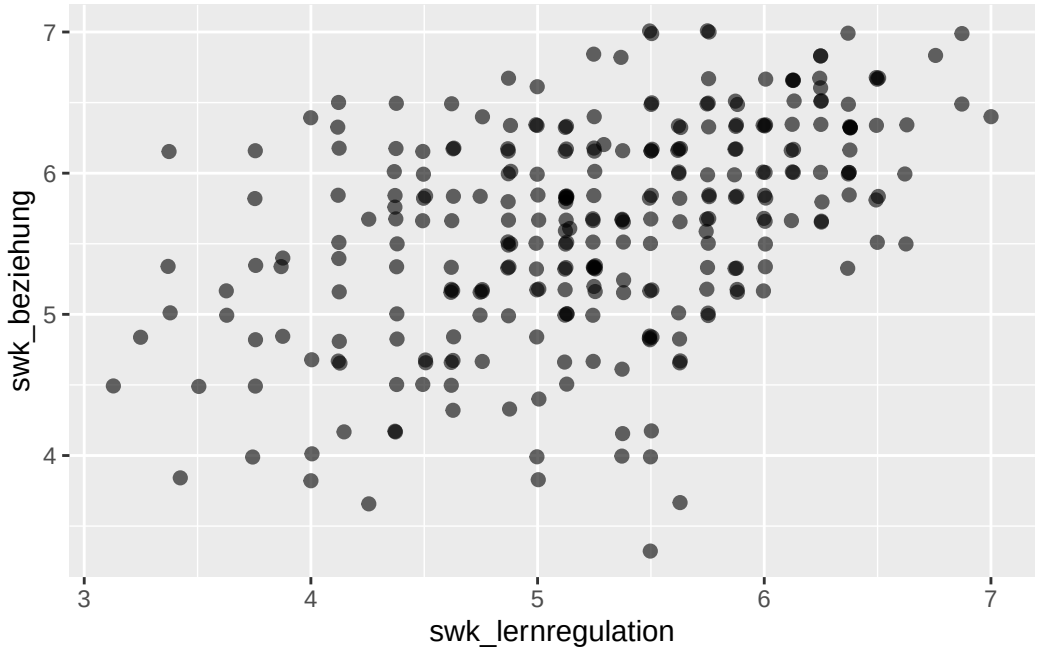
```
p + geom_jitter(aes(x = swk_lernregulation, y = swk_durchsetzung),
                alpha = 0.6, size = 2)
```



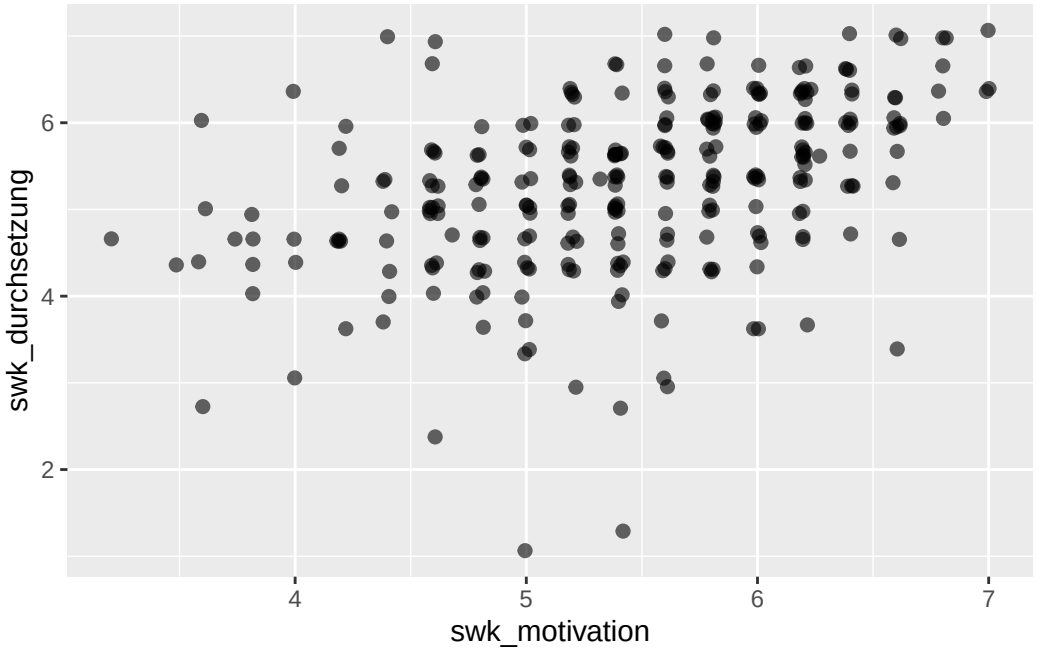

```
p + geom_jitter(aes(x = swk_lernregulation, y = swk_sozialkomp),
                alpha = 0.6, size = 2)
```



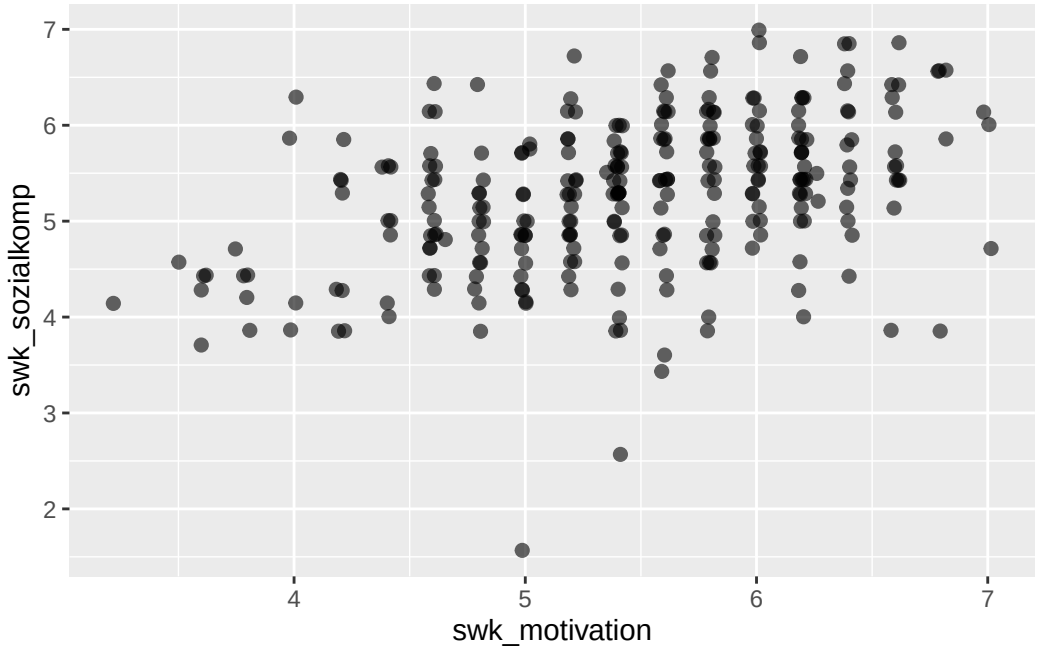

```
p + geom_jitter(aes(x = swk_lernregulation, y = swk_beziehung),
                alpha = 0.6, size = 2)
```



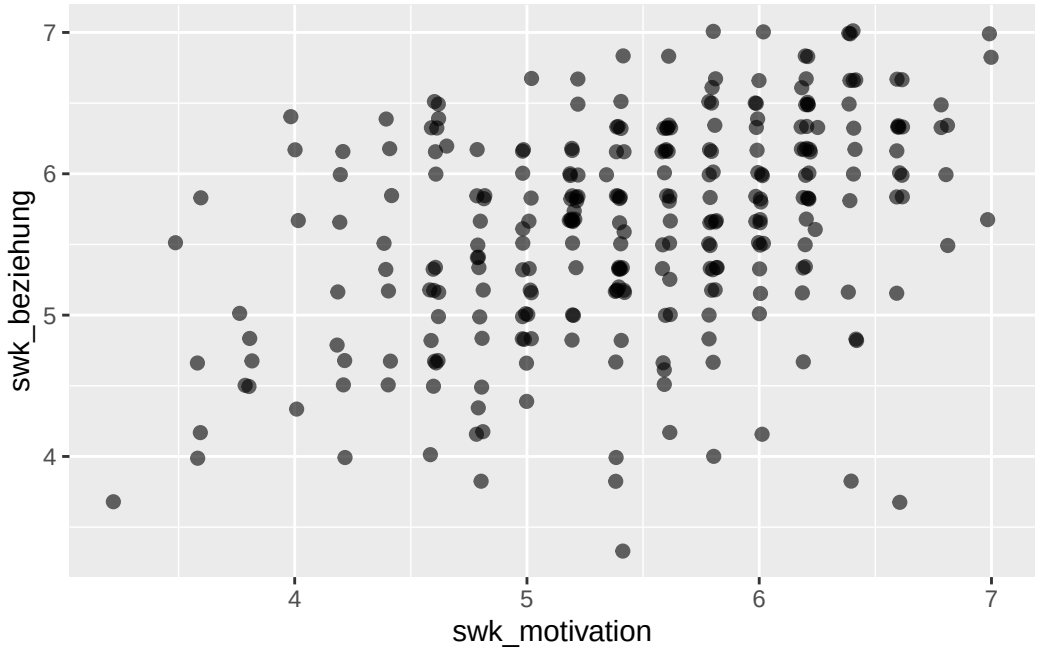
```
p + geom_jitter(aes(x = swk_motivation, y = swk_durchsetzung),
                alpha = 0.6, size = 2)
```



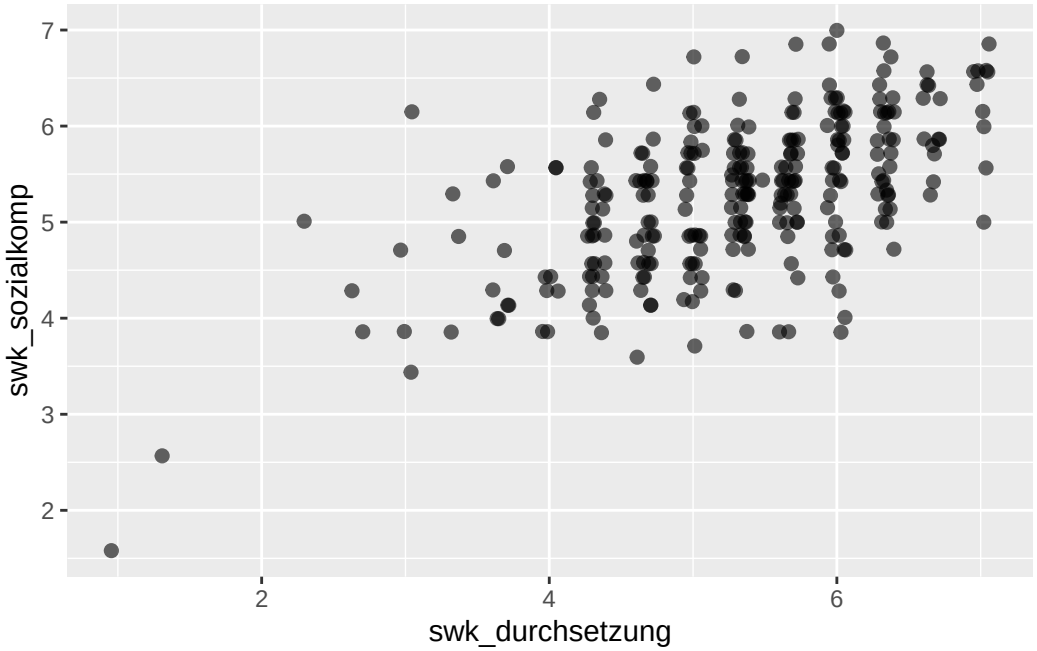

```
p + geom_jitter(aes(x = swk_motivation, y = swk_sozialkomp),
                alpha = 0.6, size = 2)
```



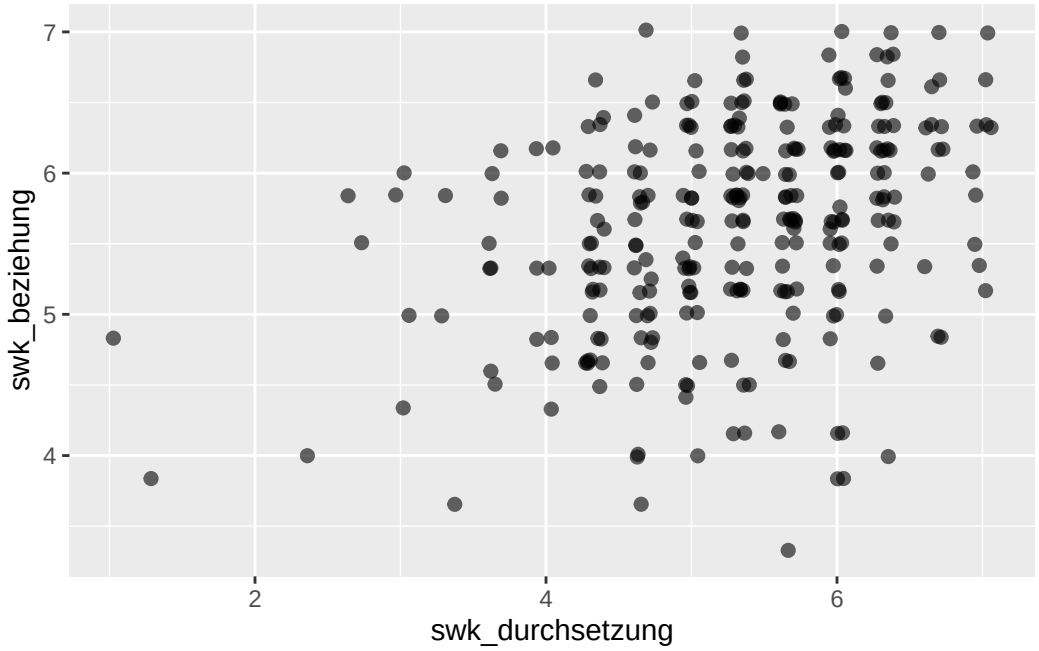

```
p + geom_jitter(aes(x = swk_motivation, y = swk_beziehung),
                alpha = 0.6, size = 2)
```



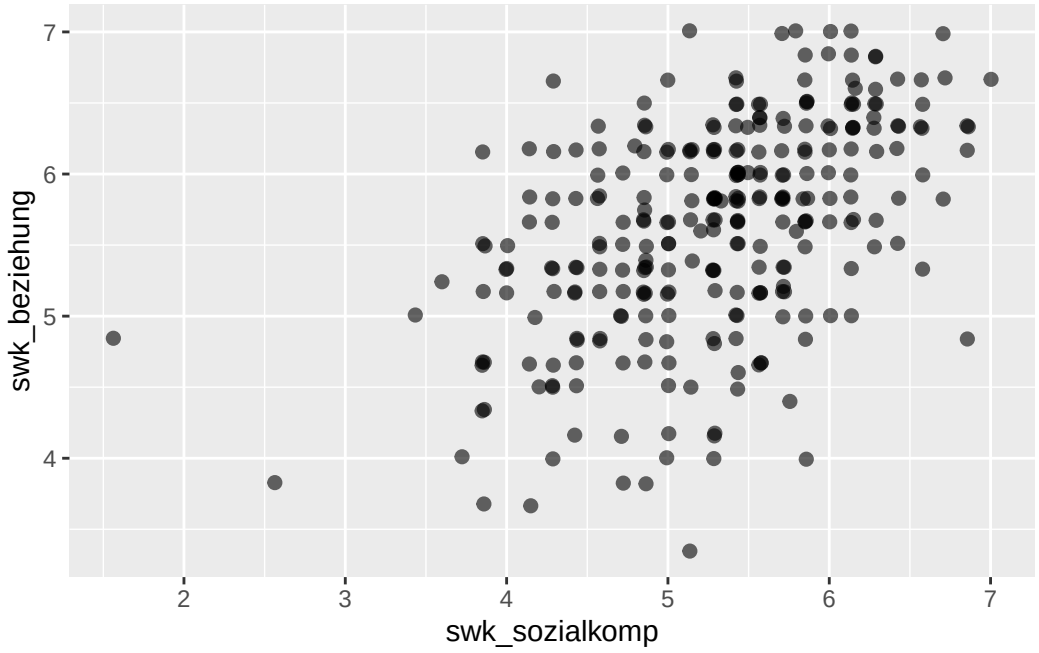
```
p + geom_jitter(aes(x = swk_durchsetzung, y = swk_sozialkomp),
                alpha = 0.6, size = 2)
```




```
p + geom_jitter(aes(x = swk_durchsetzung, y = swk_beziehung),
                alpha = 0.6, size = 2)
```




```
p + geom_jitter(aes(x = swk_sozialkomp, y = swk_beziehung),
                alpha = 0.6, size = 2)
```

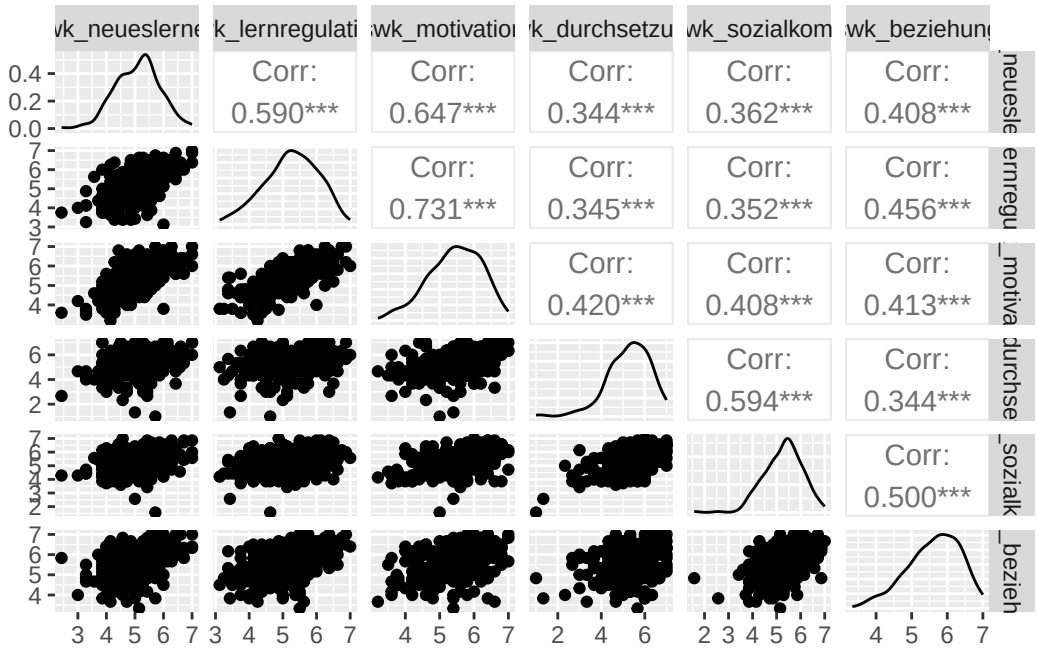


Vertiefung

Einfacher geht das mit der Funktion `ggpairs()` aus dem Package `GGally`, das wir zunächst installieren müssen:

```
install.packages("GGally")
```

```
library(GGally)
selbstwirksamkeit_wide |>
  select(starts_with("swk_")) |>
  ggpairs()
```



Hier bekommen wir neben den insgesamt 15 Scatterplots (untere Dreiecksmatrix) auch noch einen Density-Plot pro Variable in der Diagonalen ausgegeben, sowie die bivariaten Pearson-Korrelationen in der oberen Dreiecksmatrix (inkl. Signifikanztest, erkennbar an der Sternchen-Kennzeichnung: `***` bedeutet `p < 0.001`).

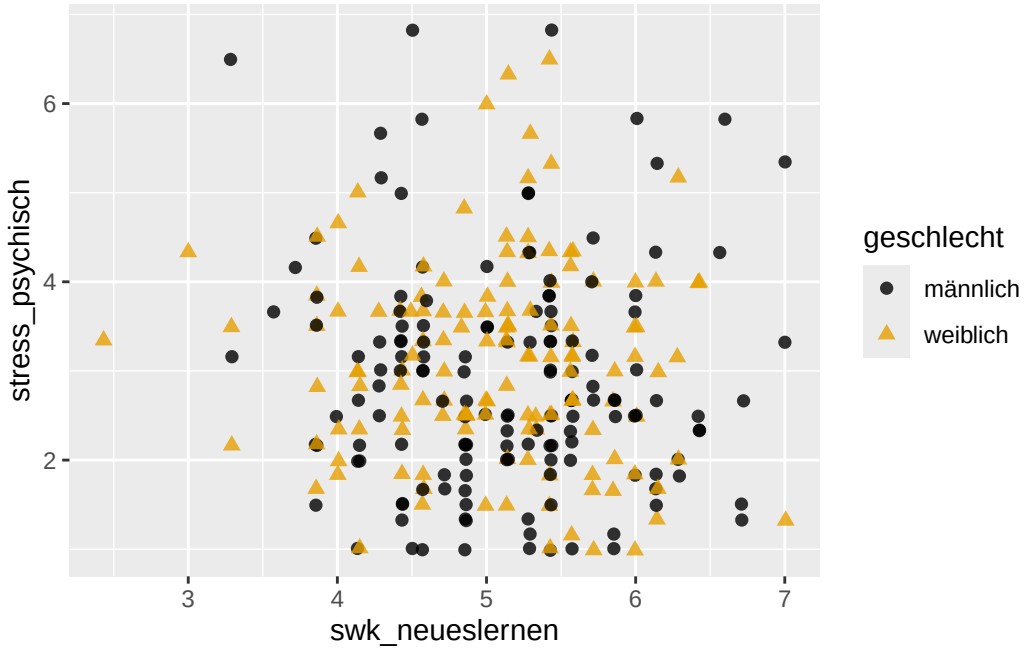
Aufgabe 2

Stellen Sie nun auch die Zusammenhänge zwischen den sechs Selbstwirksamkeitsskalen und der psychischen Symptomatik jeweils mit einem Scatterplot dar. Nutzen Sie dabei `geschlecht`

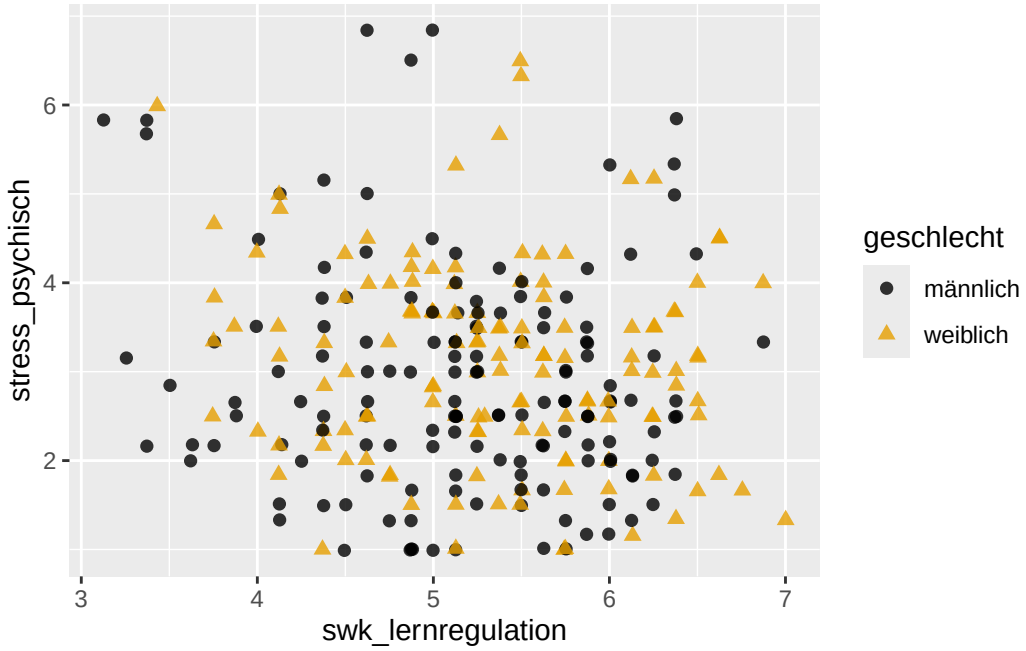
als Gruppierungsvariable, und zwar wie oben für das `color`- und für das `shape`-Argument. Insgesamt müssen dafür sechs Scatterplots erstellt werden.

Lösung

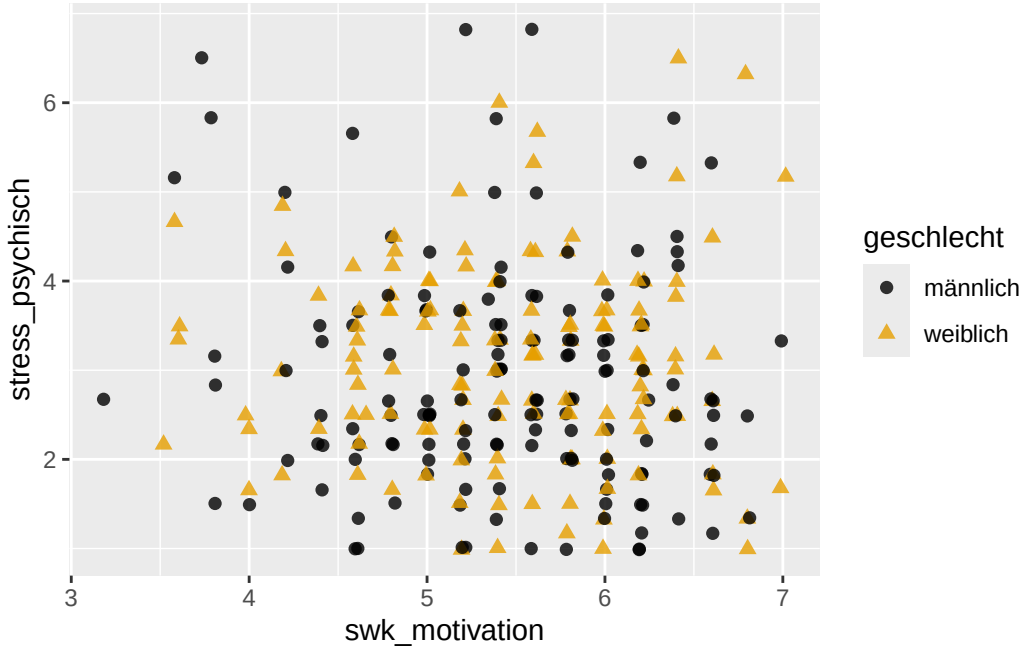
```
p <- selbstwirksamkeit_wide |>
  ggplot(mapping = aes(
    color = geschlecht,
    shape = geschlecht
  ))

p + geom_jitter(aes(x = swk_neueslernen, y = stress_psychisch),
                alpha = 0.8, size = 2) +
  scale_color_manual(values = c("#000000", "#E69F00"))
```



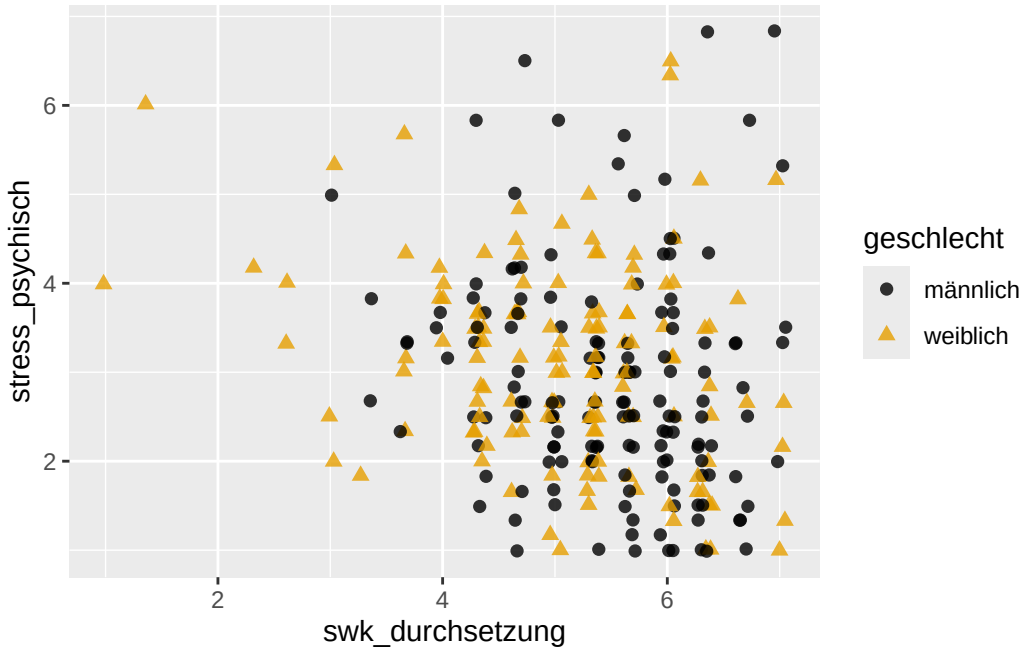

```
p + geom_jitter(aes(x = swk_lernregulation, y = stress_psychisch),
                alpha = 0.8, size = 2) +
  scale_color_manual(values = c("#000000", "#E69F00"))
```



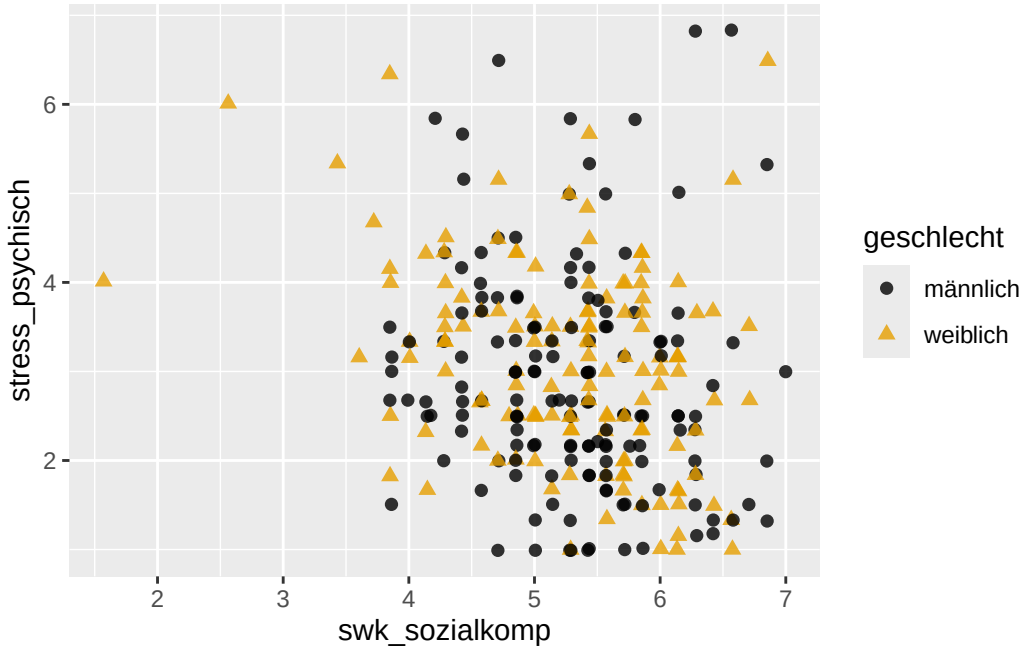

```
p + geom_jitter(aes(x = swk_motivation, y = stress_psychisch),
                alpha = 0.8, size = 2) +
  scale_color_manual(values = c("#000000", "#E69F00"))
```



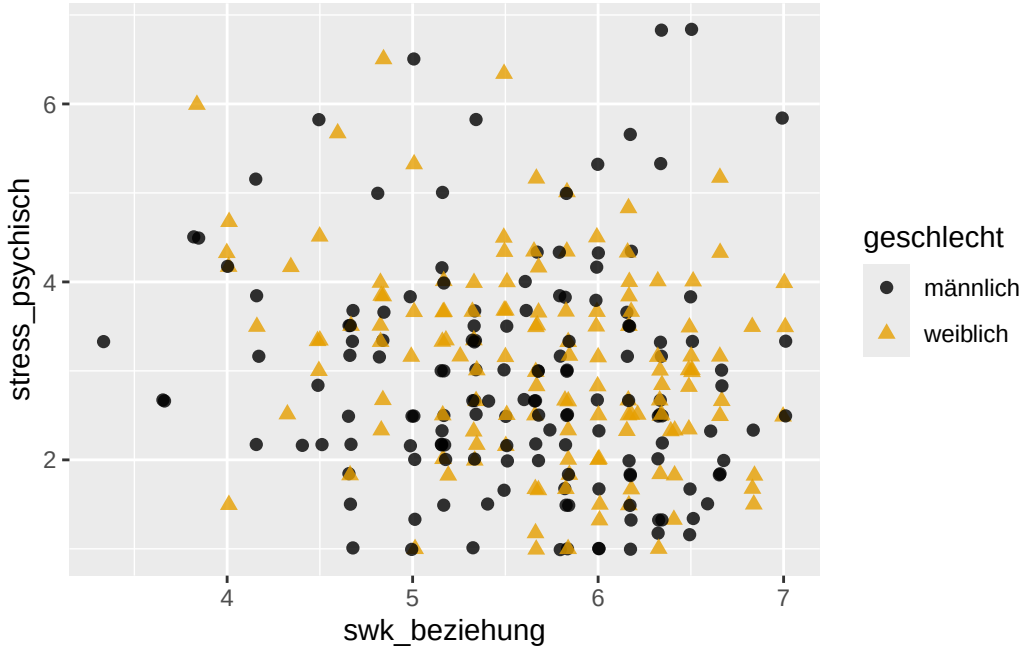
```
p + geom_jitter(aes(x = swk_durchsetzung, y = stress_psychisch),
                alpha = 0.8, size = 2) +
  scale_color_manual(values = c("#000000", "#E69F00"))
```




```
p + geom_jitter(aes(x = swk_sozialkomp, y = stress_psychisch),
                alpha = 0.8, size = 2) +
  scale_color_manual(values = c("#000000", "#E69F00"))
```




```
p + geom_jitter(aes(x = swk_beziehung, y = stress_psychisch),
                alpha = 0.8, size = 2) +
  scale_color_manual(values = c("#000000", "#E69F00"))
```



Tendenziell sieht es so aus, dass höhere Werte in der Selbstwirksamkeit mit niedrigerem psychischen Stress einhergehen (bei beiden Geschlechtern). Es wäre noch interessant, diese Beobachtung über die entsprechenden bivariaten Pearson-Korrelationskoeffizienten zu bestätigen.
Um nur die Korrelationen der sechs Selbstwirksamkeitsskalen mit der Variablen `stress_psychisch` zu erhalten (und nicht die Korrelationen aller Variablen untereinander), können wir die Funktion `cor(x, y)` nutzen, wobei `x` in diesem Fall ein Dataframe mit den sechs `swk_`-Variablen sein muss und `y` ein Dataframe, der nur die Variable `stress_psychisch` enthält. Um um die Korrelationen getrennt für männliche und weibliche Jugendliche zu erhalten, müssen wir diese Dataframes getrennt nach Geschlecht erstellen:

```
# Dataframes für Korrelationen für männliche Jugendliche
selbstwirksamkeit_maennlich <- selbstwirksamkeit_wide |>
  filter(geschlecht == "männlich") |>
  select(starts_with("swk"))

stress_psychisch_maennlich <- selbstwirksamkeit_wide |>
  filter(geschlecht == "männlich") |>
  select(stress_psychisch)

# Dataframes für Korrelationen für weibliche Jugendliche
selbstwirksamkeit_weiblich <- selbstwirksamkeit_wide |>
  filter(geschlecht == "weiblich") |>
  select(starts_with("swk"))

stress_psychisch_weiblich <- selbstwirksamkeit_wide |>
  filter(geschlecht == "weiblich") |>
  select(stress_psychisch)
```

```
# Korrelationen berechnen
cor(x = selbstwirksamkeit_maennlich, y = stress_psychisch_maennlich)
```

```
                  stress_psychisch
swk_neueslernen         -0.05253377
swk_lernregulation      -0.12120519
swk_motivation          -0.10186382
swk_durchsetzung        -0.13215266
swk_sozialkomp          -0.12918169
swk_beziehung           -0.11275072
```

```r
cor(x = selbstwirksamkeit_weiblich, y = stress_psychisch_weiblich)
```

```
                   stress_psychisch
swk_neueslernen         -0.053960724
swk_lernregulation      -0.151537381
swk_motivation          -0.006650242
swk_durchsetzung        -0.258573830
swk_sozialkomp          -0.301637007
swk_beziehung           -0.266703463
```

Während die Korrelationen der drei *akademischen* Selbstwirksamkeitsbereiche `swk_neueslernen`, `swk_lernregulation` und `swk_motivation` mit `stress_psychisch` sehr schwach sind und sich kaum zwischen den Geschlechtern unterscheiden, sind die Korrelationen der drei *sozialen* Selbstwirksamkeitsbereiche `swk_durchsetzung`, `swk_sozialkomp` und `swk_beziehung` mit `stress_psychisch` und vor allem bei den weiblichen Jugendlichen etwas stärker negativ. Insbesondere bei Mädchen gehen hohe soziale Selbstwirksamkeitsüberzeugungen daher (rein deskriptiv, auf Signifikanztests verzichten wir hier) mit niedrigeren Stresssymptomen einher.

Aufgabe 3

Nun sollen dieselben Streudiagramme wie oben erstellt werden, diesmal aber so, dass für jeden Selbstwirksamkeitsbereich ein eigener **Teilplot** erstellt wird. Ausserdem sollen jetzt für Jungen und Mädchen jeweils **getrennte Plots** erstellt werden. Das ist mithilfe der Funktion `facet_grid(bereich ~ geschlecht)` möglich.

Dafür wird ein (messwiederholter) Faktor `bereich` benötigt, der anzeigt, zu welchem Selbstwirksamkeitsbereich ein bestimmter Wert einer *einzigen* Variable `swk` gehört.

a) Führen Sie eine entsprechende wide-to-long Transformation des Datensatzes `selbstwirksamkeit_wide` durch (Ergebnis: `selbstwirksamkeit_long`).

Lösung

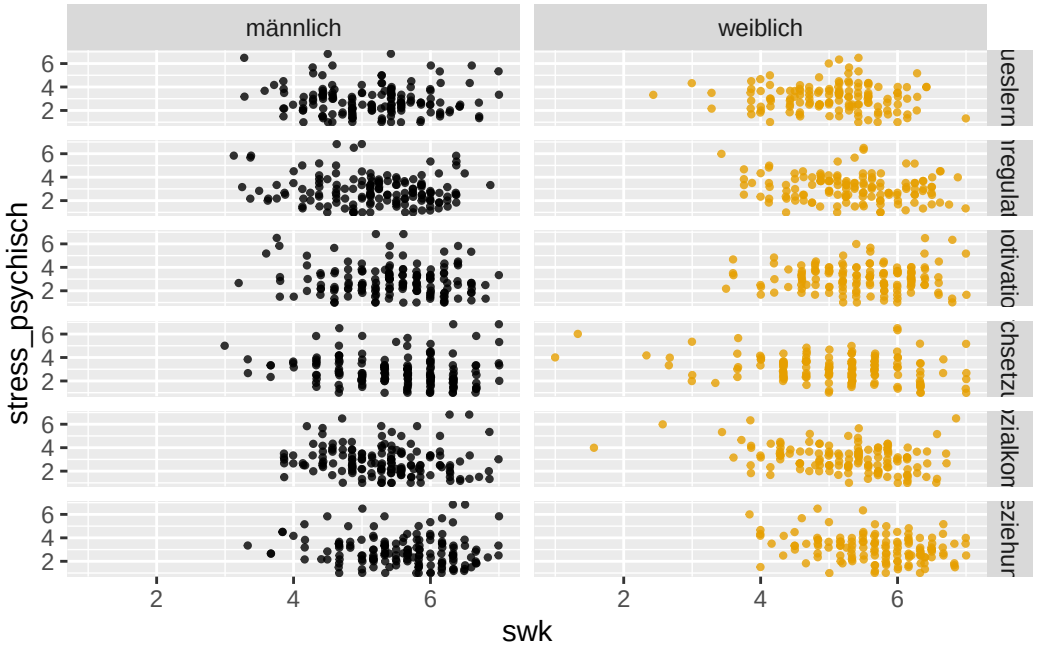
```
selbstwirksamkeit_long <- selbstwirksamkeit_wide |>
  pivot_longer(!c(ID, geschlecht, stress_psychisch),
               names_to = "bereich", values_to = "swk") |>
# Präfix swk_ von den SWK-Bereichen entfernen und `bereich` in Faktor
# konvertieren (mit Faktorstufen in der Reihenfolge der SWK-Bereiche oben,
# das ist wichtig für die Reihenfolge der Teilplots beim Plotten mit facet_grid())
  mutate(
    bereich = str_replace(bereich, "swk_", ""),
    bereich = factor(bereich, levels = c("neueslernen", "lernregulation",
                                         "motivation", "durchsetzung",
                                         "sozialkomp", "beziehung"))
  )

selbstwirksamkeit_long
```

```
# A tibble: 1,698 x 5
      ID geschlecht stress_psychisch bereich          swk
   <dbl> <fct>                 <dbl> <fct>          <dbl>
 1     1 weiblich               1.67 neueslernen     4.57
 2     1 weiblich               1.67 lernregulation  5.5
 3     1 weiblich               1.67 motivation      4.8
 4     1 weiblich               1.67 durchsetzung    5.33
 5     1 weiblich               1.67 sozialkomp      5.14
 6     1 weiblich               1.67 beziehung       6.17
 7     2 männlich               3.5  neueslernen     5
 8     2 männlich               3.5  lernregulation  4
 9     2 männlich               3.5  motivation      4.6
10     2 männlich               3.5  durchsetzung    5
# i 1,688 more rows
```

b) Jetzt kann geplottet werden. Um es noch ein bisschen schöner zu machen, wollen wir unterschiedliche Farben für die Plots von Jungen und Mädchen verwenden:

Lösung

```
p <- selbstwirksamkeit_long |>
  ggplot(aes(x = swk, y = stress_psychisch)) +
  facet_grid(bereich ~ geschlecht)

p +
  geom_jitter(alpha = 0.8, size = 0.8, aes(color = geschlecht)) +
  scale_color_manual(values = c("#000000", "#E69F00"), guide = "none")
```




```
# guide = "none" entfernt die in diesem Fall nicht benötigte Legende
```

Aufgabe 4

Im einführenden Beispiel ganz am Anfang dieses Kapitels haben wir die Verteilung von `stress_psychisch` zwischen Jungen und Mädchen verglichen, u.a. mit einem Box-Plot (`geom_boxplot`) und einem Violin-Plot (`geom_violin`). Oft will man aber nicht die ganze Verteilung einer Variablen vergleichend darstellen, sondern nur die Mittelwerte. Um es noch etwas interessanter zu machen, wollen wir jetzt nicht nur die Mittelwerte von `stress_psychisch` zwischen den Geschlechtern vergleichend darstellen, sondern auch diejenigen von `stress_somatisch` (Skala zu somatischen Stresssymptomen wie z.B. Kopfschmerzen und Schlafstörungen).

Führen Sie folgende Schritte durch:

a) Bilden Sie aus `beispieldaten` einen Subdatensatz, der nur die Variablen `ID`, `geschlecht`, `stress_psychisch` und `stress_somatisch` enthält und entfernen Sie fehlende Werte. Nennen Sie den resultierenden Dataframe `stress_wide`.

Lösung

```
stress_wide <- beispieldaten |>
  select(ID, geschlecht, stress_psychisch, stress_somatisch) |>
  drop_na()
```

b) Führen Sie wie in Aufgabe 3 eine wide-to-long Transformation durch und bilden Sie damit einen Dataframe `stress_long` mit einem messwiederholten Faktor `stressart` (psychisch vs. somatisch) und der (Outcome-)Variable `stress` (die die Ausprägungen der Variablen `stress_psychisch` und `stress_somatisch` enthält). Berechnen Sie anschliessend mittels `group_by()` und `summarize()` die Mittelwerte von `stress` getrennt nach `stressart` und `geschlecht` und speichern Sie diese in einem Dataframe `stress_means` ab.

Lösung

```
stress_long <- stress_wide |>
  pivot_longer(!c(ID, geschlecht),
               names_to = "stressart", values_to = "stress") |>
# Präfix stress_ von den beiden Stressarten entfernen und `stressart` in
# Faktor konvertieren (die Faktorstufen müssen nicht definiert werden,
# da die alphabetische default-Reihenfolge verwendet werden kann)
  mutate(
    stressart = str_replace(stressart, "stress_", ""),
    stressart = as.factor(stressart))

stress_long # anschauen
```

```
# A tibble: 566 x 4
      ID geschlecht stressart stress
   <dbl> <fct>      <fct>      <dbl>
 1     1 weiblich   psychisch   1.67
 2     1 weiblich   somatisch   3
 3     2 männlich   psychisch   3.5
 4     2 männlich   somatisch   3.83
 5    10 weiblich   psychisch   3.67
 6    10 weiblich   somatisch   3.33
```

```
 7    11 weiblich   psychisch   1.5
 8    11 weiblich   somatisch   3.5
 9    12 weiblich   psychisch   2.5
10    12 weiblich   somatisch   1.33
# i 556 more rows
```

```
# Mittelwerte für Plot berechnen
stress_means <- stress_long |>
  group_by(stressart, geschlecht) |>
  summarize(stress = mean(stress))

stress_means # Mittelwerte anschauen
```

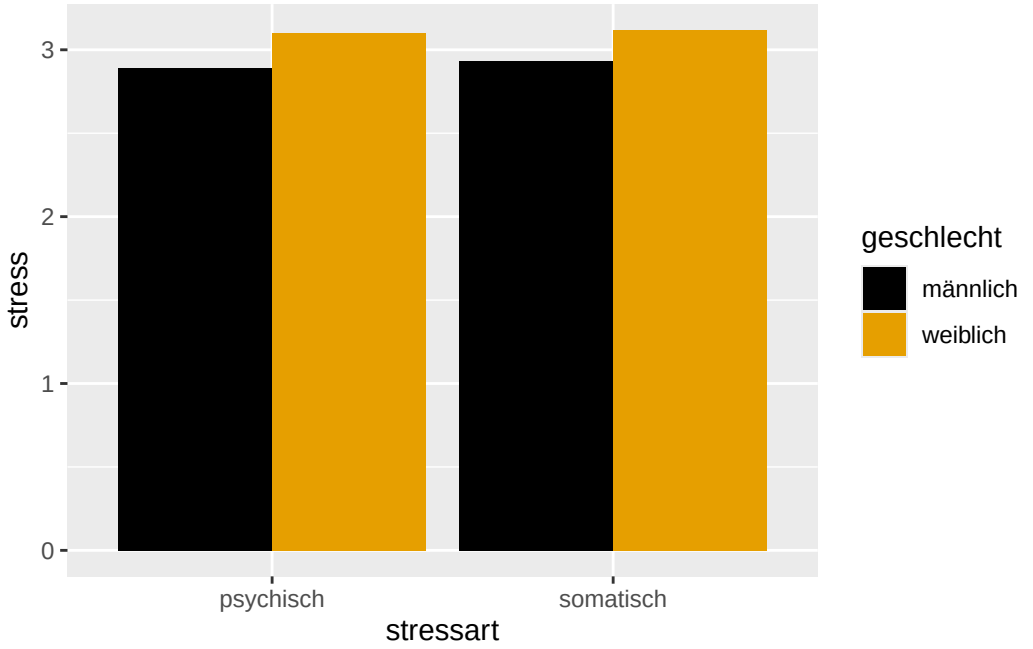
```
# A tibble: 4 x 3
# Groups:   stressart [2]
  stressart geschlecht stress
  <fct>     <fct>       <dbl>
1 psychisch männlich     2.89
2 psychisch weiblich     3.10
3 somatisch männlich     2.93
4 somatisch weiblich     3.12
```

Wir können hier schon sehen, dass jugendliche Mädchen leicht höhere Stresswerte aufweisen als Jungen (rein deskriptiv), und zwar sowohl in Bezug auf psychische als auch auf somatische Stresssymptome. Insgesamt sind die Mittelwerte um den Wert `3` herum aber als eher niedrig zu interpretieren (glücklicherweise!), da die Stresssymptome auf einer Skala von `1` (sehr geringe Auspägung) bis `7` (sehr starke Ausprägung) gemessen wurden.

c) Plotten Sie diese Mittelwerte mittels eines Balkendiagramms (Bar-Plot: `geom_bar()`) mit den beiden Stressarten auf der X-Achse, den Stress-Werten auf der Y-Achse und mit `geschlecht` als (farbliche) Gruppierungsvariable. Plotten Sie diesmal direkt (ohne vorherige Definition eines Plotobjekts).

Lösung

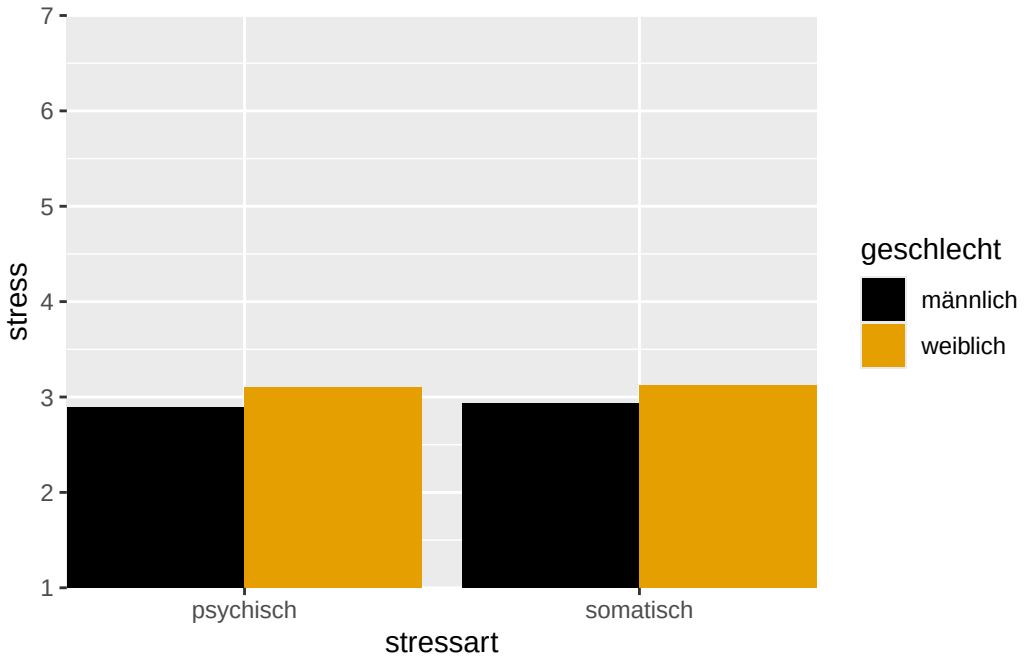
```
stress_means |>
  ggplot(aes(x = stressart, y = stress, fill = geschlecht)) +
  geom_bar(stat = "identity", position = "dodge") +
  scale_fill_manual(values = c("#000000", "#E69F00"))
```



Es fällt auf, dass Y-Achse hier bei `0` beginnt und bei knapp über `3` endet (ca. dort wo die Balken enden). Da der Wert `0` auf der Stress-Skala von `1` bis `7` gar nicht existiert, wäre es sinnvoll, die Y-Achse erst beim Wert `1` beginnen zu lassen, ausserdem könnte man den ganzen möglichen Wertebereich der Stressskala auf der Y-Achse anzeigen lassen, um einen zutreffenderen Gesamteindruck von der Höhe der Stresssymptome und der Geschlechtsunterschiede zu erhalten. Das ist möglich mit der bisher nicht besprochenen `ggplot2`-Funktion `coord_cartesian()`, wie im Folgenden gezeigt wird.

Vertiefung

```
stress_means |>
  ggplot(aes(x = stressart, y = stress, fill = geschlecht)) +
  geom_bar(stat = "identity", position = "dodge") +
  scale_fill_manual(values = c("#000000", "#E69F00")) +
  coord_cartesian(ylim = c(1, 7), expand = FALSE)
```



In `coord_cartesian()` bestimmt das Argument `ylim = c(1, 7)` Anfang und Ende der Y-Achse, und `expand = FALSE` macht, dass die Achse *genau* beim Wert `1` beginnt und *genau* beim Wert `7` endet (und nicht leicht darüber hinaus erweitert wird wie mit `expand = TRUE`).

Aufgabe 5

Jetzt sollen auch noch für die sechs Selbstwirksamkeitsskalen Mittelwertsvergleiche zwischen Jungen und Mädchen geplottet werden. Gehen Sie dabei genauso wie in Aufgabe 4 vor, aber benutzen Sie statt eines Bar-Plots ein Liniendiagramm mit zusätzlichen Punkten.

Hinweis

Beachten Sie, dass `geom_line()` ein zusätzliches `group`-Argument in `aes()` benötigt, um zu wissen, welche Punkte mit Linien verbunden werden sollen. Das allgemeine `color = geschlecht` der `ggplot()` Funktion genügt hier nicht (wird aber trotzdem benötigt, um Linien und Punkte mit nach Geschlecht unterschiedlichen Farben zu erhalten).

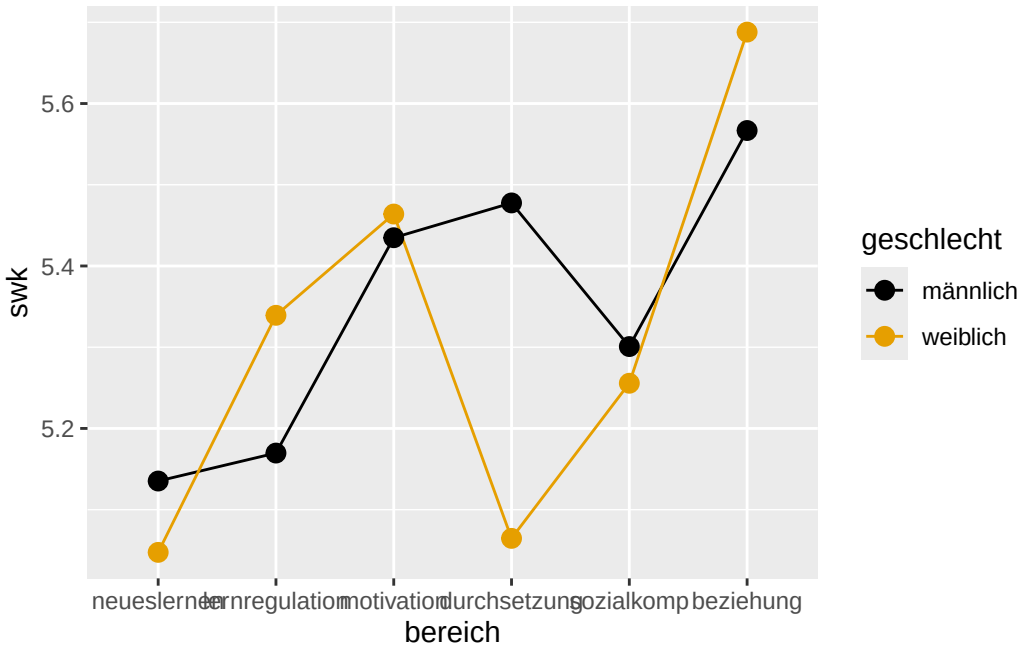
Benutzen Sie den in Aufgabe 3 gebildeten Datensatz `selbstwirksamkeit_long` zunächst, um die zu plottenden Mittelwerte zu berechnen.

Zum Abschluss soll der Plot noch (weiter) verschönert werden: er soll einen weissen Hintergrund bekommen sowie einen Titel und bessere (bzw. korrektere) Achsen- und Legendenbeschriftungen.

Plotten Sie wieder direkt (ohne vorherige Definition eines Plotobjekts).

Lösung

```
swk_means <- selbstwirksamkeit_long |>
  group_by(bereich, geschlecht) |>
  summarize(swk = mean(swk))

swk_means |>
  ggplot(aes(x = bereich, y = swk, color = geschlecht)) +
  geom_line(aes(group = geschlecht)) +
  geom_point(size = 3) +
  scale_color_manual(values = c("#000000", "#E69F00"))
```

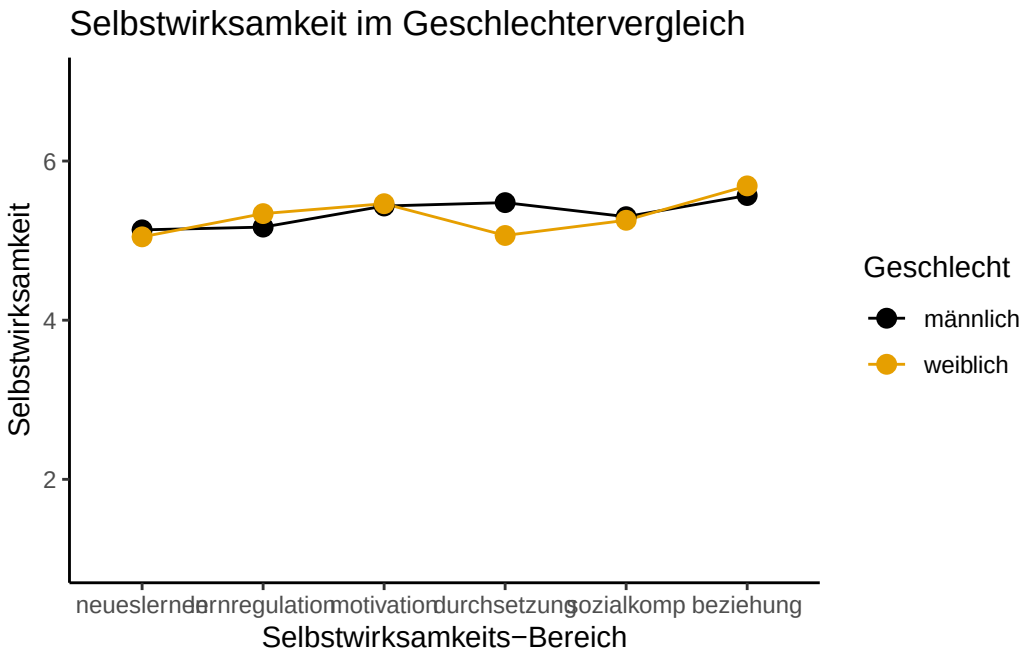


Ohne die `coord_cartesian()`-Funktion wird hier nur derjenige Auschnitt der `swk`-Skala dargestellt, in dem sich die zu plottenden Mittelwerte befinden (diese liegen zwischen `5.05` und `5.69`, s.o.). Dadurch wirkt es hier so, als gäbe es relativ grosse Geschlechtsunterschiede in der Selbstwirksamkeit, je nach Bereich in unterschiedliche Richtungen. Tatsächlich handelt es sich relativ zur Gesamtskala (`1` bis `7`) aber grösstenteils um sehr geringe Unterschiede.

Daher plotten wir noch einmal und geben jetzt mit Hilfe von `coord_cartesian()` an, dass die Y-Achse von `1` bis `7` gehen soll. Diesmal wählen wir `expand = TRUE` (default), damit wir links und rechts ein bisschen Platz für die Darstellung der Punkte bekommen (mit `expand = FALSE` würden die Punkte des ersten Bereichs `neueslernen` und die des letzten Bereichs `beziehung` nicht vollständig geplottet).

Ausserdem jetzt mit den Achsenbeschriftungen etc.:

```r
swk_means |>
  ggplot(aes(x = bereich, y = swk, color = geschlecht)) +
  geom_line(aes(group = geschlecht)) +
  geom_point(size = 3) +
  scale_color_manual(values = c("#000000", "#E69F00")) +
  coord_cartesian(ylim = c(1, 7), expand = TRUE) +
  theme_classic() +
  ggtitle("Selbstwirksamkeit im Geschlechtervergleich") +
  xlab("Selbstwirksamkeits-Bereich") +
  ylab("Selbstwirksamkeit") +
  labs(color = "Geschlecht")
```



In dieser Darstellung kann man jetzt gut erkennen, dass einzig im Bereich der Durchsetzungsfähigkeit (`durchsetzung`) ein etwas grösserer Geschlechtsunterschied besteht (Jungen geben dort eine höhere Selbstwirksamkeitsüberzeugung an als Mädchen). Alle anderen Unterschiede sind sehr gering.

Literatur

Allaire, J. (2022). *quarto: R Interface to Quarto Markdown Publishing System*. https://github.com/quarto-dev/quarto-r

Grolemund, G. (2014). *Hands-on programming with R. Write your own functions and simulations*. O'Reilly. https://rstudio-education.github.io/hopr/

Luhmann, M. (2020). *R für Einsteiger. Einführung in die Statistik-Software für die Sozialwissenschaften*. (5. Aufl.). Beltz. https://www.beltz.de/fachmedien/psychologie/produkte/details/43910-r-fuer-einsteiger.html

Schloerke, B., Cook, D., Larmarange, J., Briatte, F., Marbach, M., Thoen, E., Elberg, A., & Crowley, J. (2021). *GGally: Extension to ggplot2*. https://CRAN.R-project.org/package=GGally

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. https://ggplot2.tidyverse.org

Wickham, H. (2019). *Advanced R* (2. Aufl.). Chapman & Hall. https://adv-r.hadley.nz/

Wickham, H. (2022a). *stringr: Simple, Consistent Wrappers for Common String Operations*. https://CRAN.R-project.org/package=stringr

Wickham, H. (2022b). *tidyverse: Easily Install and Load the Tidyverse*. https://CRAN.R-project.org/package=tidyverse

Wickham, H. (2023). *forcats: Tools for Working with Categorical Variables (Factors)*. https://CRAN.R-project.org/package=forcats

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., … Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, *4*(43), 1686. https://doi.org/10.21105/joss.01686

Wickham, H., & Bryan, J. (2022). *readxl: Read Excel Files*. https://CRAN.R-project.org/package=readxl

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). *R for Data Science* (2. Aufl.). O'Reilly. https://r4ds.hadley.nz/

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., & Dunnington, D. (2022). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. https://CRAN.R-project.org/package=ggplot2

Wickham, H., François, R., Henry, L., Müller, K., & Vaughan, D. (2023). *dplyr: A Grammar of Data Manipulation*. https://CRAN.R-project.org/package=dplyr

Wickham, H., & Grolemund, G. (2017). *R for Data Science* (1. Aufl.). O'Reilly. https://r4ds.had.co.nz/index.html

Wickham, H., Hester, J., & Bryan, J. (2022). *readr: Read Rectangular Text Data*. https://CRAN.R-project.org/package=readr

Wickham, H., Miller, E., & Smith, D. (2022). *haven: Import and Export SPSS, Stata and SAS Files*. https://CRAN.R-project.org/package=haven

Wickham, H., Vaughan, D., & Girlich, M. (2023). *tidyr: Tidy Messy Data*. https://CRAN.R-project.org/package=tidyr

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman; Hall/CRC. https://bookdown.org/yihui/bookdown

Xie, Y. (2023). *bookdown: Authoring Books and Technical Documents with R Markdown*. https://CRAN.R-project.org/package=bookdown