Einführung in die Datenwissenschaft mit R

Christian Glahn

Inhaltsverzeichnis

Vorwort

Work in Progress

Copyright

Die PDF-Version liegt hier zum Download.

1. Einleitung

⚠ Work in Progress

1.1. Motivation und Ausgangslage

1.2. Base R und Tidy R

R ist eine Programmiersprache, die durch *Funktionsbibliotheken* erweitert wird. Beim Starten von R wird zuerst nur das Basissystem geladen. Das R-Basissystem besteht aus den eingebauten Sprachelementen und Funktionen sowie aus den Bibliotheken `base`, `compiler`, `datasets`, `grDevices`, `graphics`, `grid`, `methods`, `parallel`, `splines`, `stats`, `stats4`, `tcltk`, `tools`, `translations`, and `utils`. Alle Funktionen dieser Bibliotheken stehen damit nach dem Start von R sofort bereit. Diese Funktionen heissen im R-Jargon **Base R**. Dieses Basissystem stellt bereits alle Funktionen für das statistische Programmieren bereit.

Base R verwendent sehr viele *Idiome* als Funktionsnamen, aus denen sich nicht intuitiv erschliesst, was eine Funktion leistet. Ausserdem wurden im Laufe der Entwicklung immer wieder Funktionen dem Basissystem hinzugefügt, die sich nicht konsistent in das bestehende System aus Funktionen integrieren. Als Beispiel sollen die Funktionen `eapply()`, `lapply()`, `sapply()`, `tapply()` und `vapply()` sowie `replicate()` und `rep()` dienen. Bis auf `replicate()` sehen die Funktionsnamen ähnlich aus, werden aber in unterschiedlichen Kontexten verwendet und auf unterschiedliche Weise aufgerufen. `replicate()` und `rep()` haben einen ähnlichen Funktionsnamen und in der Beschreibung dienen beide Funktionen der replikation. Die Funktion `replicate()` ist aber eine Variante der Funktion `lapply()` mit ähnlicher Syntax und `rep()` nicht.

Beim Erlernen von Base R müssen die Kernsyntax der Programmiersprache, die Verwendung der Idiome mit ihren passenden Kontexten und Anwendungen sowie alle Widersprüche erlernt werden. Für Programmierneulinge erscheinen Programme in Base R sehr kyptisch und wenig intuitiv. Selbst erfahrenen R-Entwickler:innen erschliesst sich die Funktionsweise einiger Base R-Programme erst nach dem Studium der zugehörigen Bibliotheksdokumentation.

Mit zunehmender Bedeutung der Datenwissenschaften, wurden die Inkonsistenzen von Base R zum Hindernis für komplexe Anwendungen und Analysen. Ausgehend von einer konsistenten Syntax für die Datenvisualisierung wurden nach und nach R-Bibliotheken für eine

konsistente und koherente Datentransformation und -Auswertung bereitgestellt. Diese Bibliotheken stellen Daten und Datenströme in das Zentrum der Programmierung. Durch selbsterklärende Funktionsnamen, das zusammenfassen in Funktionsgruppen und einheitliche Logik für Funktionsaufrufe bilden diese Bibliotheken einen *R-Dialekt*, der als **tidy R** bezeichnet wird. R-Programme sind auch für unerfahrende R-Interessierte deutlich intuitiver zu verstehen, wenn sie mit den tidy R Konzepten entwickelt wurden, als vergleichbare Base R Varianten. Durch den datenzentrierten Zugang lässt sich tidy R wesentlich leichter erlernen als Base R.

Den Kern von tidy R bildet die Bibliothek `tidyverse`, die die wichtigsten Funktionen für die Datentransformation und die Datenvisualisierung zusammenfasst. Sie besteht aus den Unterbibliotheken `dplyr`, `forcats`, `ggplot2`, `purrr`, `readr`, `tibble` und `tidyr`.

Wichtige ergänzende Funktionen für die Statistik und das statistische Modellieren werden durch die Bibliotheken `rstatix` und `tidymodels` bereitgestellt.

In diesem Buch werden alle Konzepte datenzentrisch mit dem tidy R Ansatz erarbeitet. Base R Konzepte, Operatoren und Funktionen werden nur verwendet, wenn diese nicht im Widerspruch zu tidy R stehen. In diesen Fällen werden diese nicht gesondert als Base R hervorgehoben.

1.3. Organisation dieses Buchs

2. Tool Chain

2.1. R installieren

Die Installation von R ist einfach. Auf der R-Project Webseite kann das Installationspaket für das jeweilige Betriebssystem heruntergeladen werden. Die Installation erfolgt wie gewohnt über den Installer.

MacOS

Viele R-Bibliotheken benötigen zusätzliche Komponenten, damit sie funktionieren. Diese Komponenten müssen zusätzlich *kompiliert* werden. Unter MacOS benötigt R dafür die App **XCode** und die **XCode Command Line Tools**.
Beide Komponenten stehen unter MacOS kostenlos zur Verfügung. XCode wird wie gewohnt über Apple's AppStore installiert. Nach der Installation muss XCode einmal gestartet werden, um die Lizenzbedingungen zu akzeptieren. Anschliessend sollten die notwendigen Ergänzungen für die Entwicklung unter MacOS installiert werden.
Nach erfolgreicher Installation erscheint eine Abfrage, zum Starten eines neuen Projekts (Abbildung 2.1).

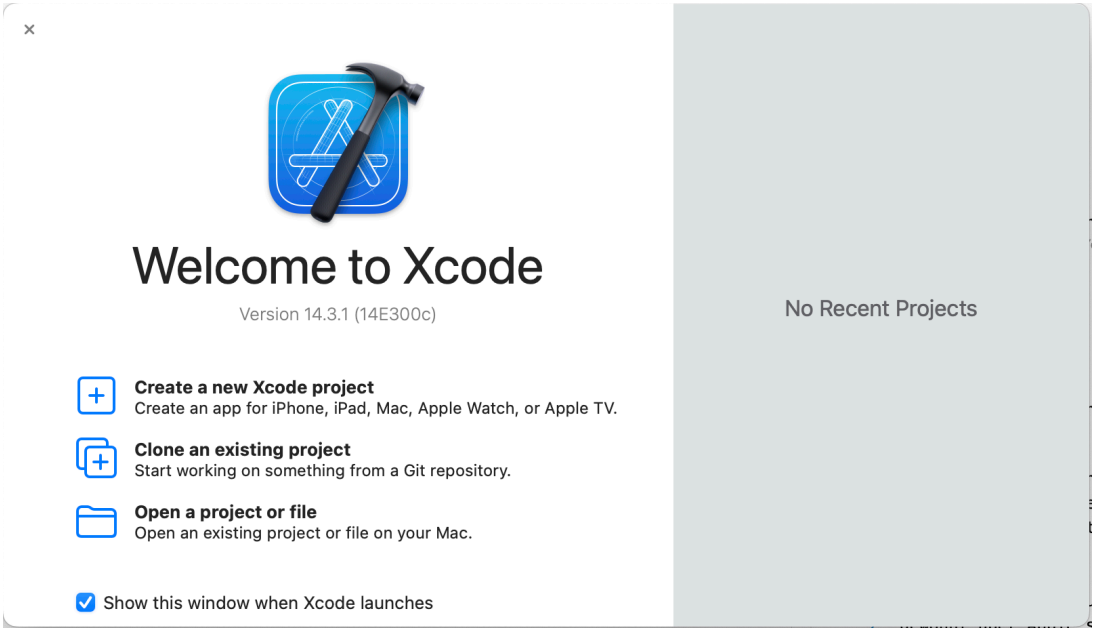


Abbildung 2.1.: XCode Start Dialog

Damit ist die Installation von *XCode* abgeschlossen. Nun folgt die Installation der Kommandozeilenwerkzeuge. Dazu muss ein Terminal geöffnet werden.

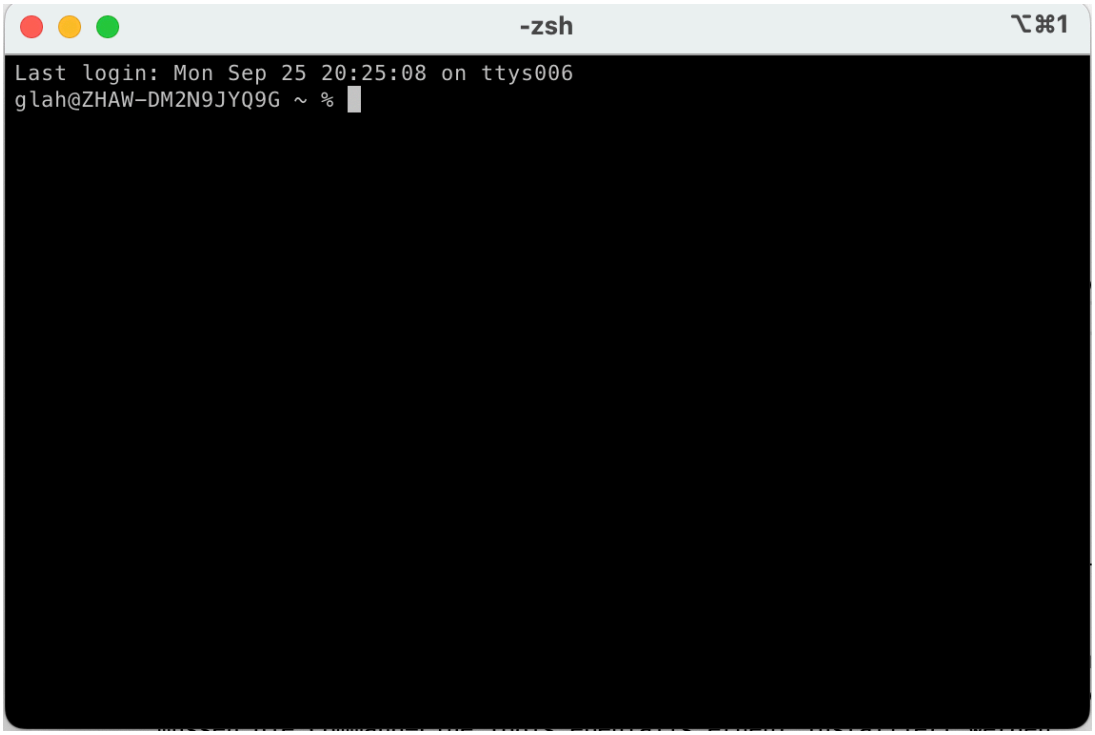


Abbildung 2.2.: MacOS Terminal

Im Terminal muss das folgende Kommando eingegeben und anschliessend mit der Eingabe-Taste abgeschlossen werden.

```
xcode-select --install
```

Anschliessend folgen mehrere Abfragen zur Installation der XCode-Command-Line Komponenten. Nach der Installation kann das Terminal und XCode wieder geschlossen werden.
XCode wird regelmässig grösseren Änderungen unterzogen. Diese Änderungen erfolgen oft im April, Juni und September. Nach einem Update von XCode müssen die Command-Line Tools ebenfalls erneut installiert werden. Ausserdem ist es notwendig, dass die Lizenzbedingungen erneut akzeptiert werden, sonst lassen sich R-Bibliotheken nicht mehr kompilieren.

2.1.1. Überprüfen der Installation

Nach erfolgreicher Installation sollte R mit den Werkzeugen der Laufzeitumgebung auf dem Rechner vorhanden sein. Die Installation lässt sich mithilfe des Terminals (MacOS) oder

der Powershell (Windows) überprüfen.

> **MacOS vs. Windows**
>
> Unter MacOS muss der folgende Befehl eingegeben und mit der Eingabe-Taste abgeschlossen werden.
>
> ```
> Rscript -e 'sessionInfo()'
> ```
>
> In der Windows Powershell muss der Befehl wie folgt aussehen:
>
> ```
> RSCRIPT.EXE -e 'sessionInfo()'
> ```

Bei erfolgreicher Installation erscheint eine Meldung im Terminal, die der folgen Meldung ähnelt. Die Funktion `sessionInfo()` zeigt die Versionsinformation der aktuellen R-Installation an.

```
R version 4.3.1 (2023-06-16)
Platform: aarch64-apple-darwin22.4.0 (64-bit)
Running under: macOS Ventura 13.5.2

Matrix products: default
BLAS:   /opt/homebrew/Cellar/openblas/0.3.23/lib/libopenblasp-r0.3.23.dylib
LAPACK: /opt/homebrew/Cellar/r/4.3.1/lib/R/lib/libRlapack.dylib;  LAPACK version 3.11.0

locale:
[1] de_DE.UTF-8/de_DE.UTF-8/de_DE.UTF-8/C/de_DE.UTF-8/de_DE.UTF-8

time zone: Europe/Zurich
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
[1] compiler_4.3.1
```

2.1.2. Erste Schritte

R ist eine *interaktive Sprache* und besteht im Kern aus einer sog. Laufzeitumgebung. Diese Umgebung übersetzt R-Syntax in Maschinensprache und führt vollständige *Ausdrücke* direkt aus. Innerhalb der Laufzeitumgebung steht ein einfacher Zeileneditor zur Verfügung, mit dem Eingaben erstellt und manipuliert werden können.

> ⚠ MacOS vs Windows
>
> Die R-Laufzeitumgebung wird auf MacOS-Systeme im Terminal mit dem Kommando `R` und Eingabetaste gestartet.
> Auf Windows-Systemen wird die Laufzeitumgebung in der Powershell mit dem Kommando `R.EXE` aufgerufen.

R verwendet für alle Operationen **Funktionen**. Die meisten R-Funktionen haben einen Namen und werden mit runden Klammern aufgerufen. Ein Beispiel ist die Funktion `quit()`, mit der die R-Laufzeitumgebung verlassen wird. Funktionen werden in die Laufzeitumgebung eingegeben und durch das Drücken der Eingabetaste ausgeführt.

```
quit()
```

Dieser Funktionsaufruf führt zu der Abfrage, ob die aktuelle Arbeitsumgebung gesichert werden soll.

```
Save workspace image? [y/n/c]:
```

Weil nichts geändert wurde, kann diese Frage mit `n` für *No* beantwortet und mit einem Druck auf die Eingabetaste übergeben werden. Anschliessend wird die Laufzeitumgebung geschlossen und kehrt auf die Kommandozeile des Betriebssystems zurück.

Neben der interaktiven Laufzeitumgebung wird R mit dem Programm `Rscript` ausgeliefert. Mit `Rscript` können Dateien mit R-Code zusammenhängend ausgeführt werden.

Definition 2.1. Ein **R-Script** ist eine Datei, die nur R-Code enthält. Ein R-Script hat per *Konvention* die Dateiendung `.r`

> ⚠ MacOS vs. Windows
>
> Das Programm `Rscript` heisst unter Windows `RSCRIPT.EXE`.

Beispiel 2.1 (Ausführen eines R-Scripts im MacOS Terminal)**.**

```
Rscript my-rscript.r
```

`Rscript` kann ausserdem einzelne Code-Zeilen ausführen, ohne in die interaktive Laufzeitumgebung wechseln zu müssen. Diese Funktion ist praktisch, um eine einfache Operation auszuführen, wie z.B. eine Bibliothek zu installieren (s. Kapitel 2.3).

2.2. Grafische Oberflächen für R

R hat keine eigene grafische Benutzeroberfläche und ist auf eine externe Entwicklungsumgebung angewiesen. Eine solche Entwicklungsumgebung muss zusätzlich zu R installiert werden, damit die Programmierung und die Analyse vereinfacht wird. Die am häufigsten eingesetzten Entwicklungsumgebungen für R sind:

- RStudio
- Jupyter Notebooks
- Visual Studio Code

> **Hinweis**
>
> In diesem Buch wird Visual Studio Code für alle Beispiele mit Benutzeroberfläche verwendet. Die Bedienung von RStudio oder JupyterLab unterscheidet für die Arbeit in diesem Buch sich nur marginal von Visual Studio Code.

Eine R-Entwicklungsumgebung ist unabhängig von der R-Laufzeitumgebung, die die Programmiersprache bereitstellt. Es ist also möglich, R-Programme in der einen Umgebung zu entwickeln und später in einer anderen weiterzubearbeiten und auszuführen.

Die Grundkomponenten einer Entwicklungsumgebung sind immer gleich (Abbildung 2.3):

- Code-Editor, mit dem Dokumentation und analytische Funktionen geschrieben werden.
- Datei-Browser, über den alle Dateien eines Projekts verwaltet werden können.

- Laufzeit-Console, über die die R-Laufzeitumgebung zugänglich ist.
- Datenbetrachter, zur Auswertung von generierten Datenstrukturen.
- Visualisierungsbetrachter, zur Anzeige von Datenvisualisierungen.

Neben diesen Komponenten existieren oft zusätzliche Werkzeuge und Ansichten.

- Werkzeuge zur Versionierung von Code und Daten.
- Dokumentation für R und ergänzende Bibliotheken.
- Installationsunterstützung von Bibliotheken.

2.2.1. RStudio

RStudio ist eine integrierte Analyseumgebung, die speziell für die Entwicklung von R-Programmen und R-Analysen entwickelt wurde. Das System ist auf R-spezifische Arbeitsabläufe zur Datenanalyse ausgerichtet und unterstützt neben R auch die Programmiersprache Python.

RStudio verwendet eine spezielle Version von Markdown, um R-Code-Fragmente auszuführen und die Ergebnisse in das Dokument einzubinden. Dieses Format heisst R-Markdown.

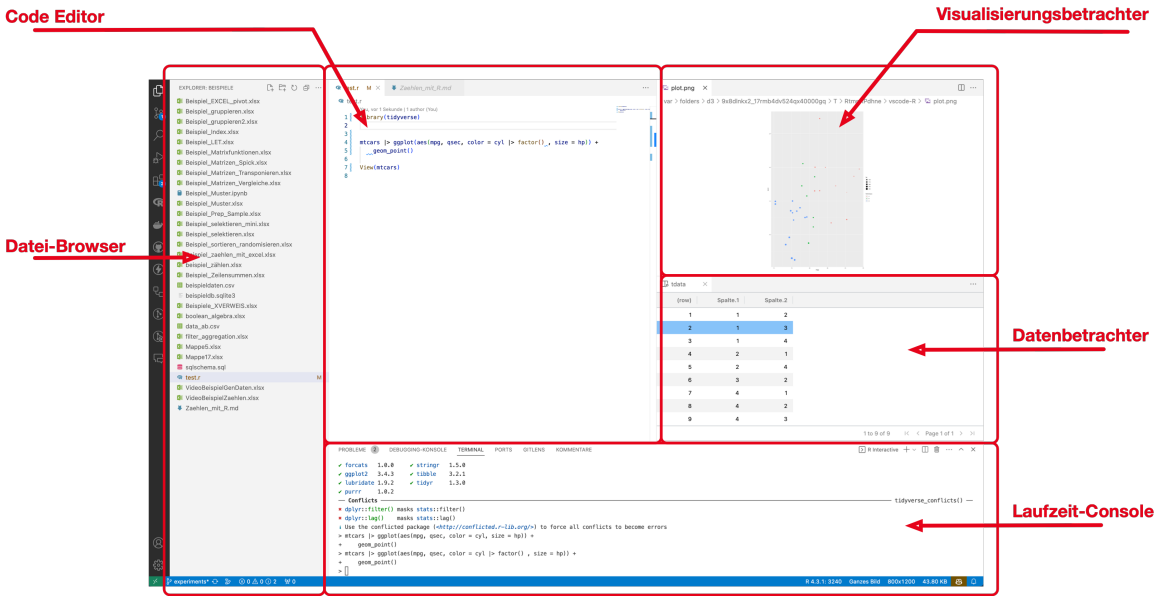


Abbildung 2.3.: Komponenten einer datenzentrischen Entwicklungsumgebung am Beispiel von Visual Studio Code

Von RStudio existiert auch eine Web-basierte Version, welche online Zusammenarbeit und online Publikationen unterstützt.

2.2.2. JupyterLab

JupyterLab ist eine Web-basierte Analyseumgebung, die ursprünglich für die Programmiersprache Python entwickelt wurde. JupyterLab wurde speziell für Datendokumente entwickelt und unterstützt ausschliesslich Jupyter Notebooks als Austauschformat für Analysen.

JupyterLab unterstützt neben Python viele andere Programmiersprachen. Dazu gehört auch R. JupyterLab integriert Programmiersprachen durch spezielle *Kernel*, die Code-Fragmente auswerten und die Ergebnisse in ein Datendokument einbinden.

In der Praxis werden Jupyter Notebooks und JupyterLab oft eingesetzt, wenn sehr umfangreiche Daten analysiert werden sollen, die nicht ohne weiteres über das Internet übertragen werden können oder dürfen.

2.2.3. Visual Studio Code

Visual Studio Code ist ein kostenloser Code-Editor mit vielen Erweiterungen für fast alle Programmiersprachen und Arbeitsumgebungen. Die Erweiterung für R ist ebenfalls kostenlos und kann über den Extension Manager installiert werden.

Im Gegensatz zu R-Studio ist Visual Studio Code in erster Linie ein Code-Editor und bietet für R eine vergleichsweise einfache Entwicklungsumgebung. Der grösste Unterschied zwischen Visual Studio Code und RStudio oder JupyterLab ist der wenig differenzierte Variablen-Inspektor.

In Visual Studio Code lassen sich u.a. auch R-Markdown-Dokumente und Jupyter Notebooks bearbeiten.

2.3. R-Bibliotheken installieren

R verfügt über einen sehr grossen Fundus an Lösungen für das statistische Rechnen. Diese Lösungen werden als Bibliotheken bereitgestellt und über das *Comprehensive R Archive Network* (CRAN) geteilt. CRAN ist ein integraler Bestandteil von R. Weil R jedoch über sehr viele Bibliotheken verfügt, werden diese nicht mit R ausgeliefert, sondern müssen bei Bedarf installiert werden. Hierzu liefert R die Funktion `install.packages()` mit. Diese Funktion teilt R mit, eine Bibliothek mit einem bestimmten Namen zu installieren.

Beispiel 2.2 (Funktions-Schema von `install.packages()`)**.**

```
install.packages(package_name)
```

In diesem Buch werden neben den R-Basisfunktionen fast ausschliesslich die Funktionen der `tidyverse`-Bibliothek behandelt. Die `tidyverse`-Bibliothek erweitert die R-Syntax um moderne Sprachkonzepte und vereinheitlicht viele Funktionen für Standardaufgaben.

> **Hinweis**
>
> Streng genommen ist die `tidyverse`-Bibliothek eine R-Bibliothek im engeren Sinn. Vielmehr vereint sie die häufig zusammen eingesetzten Biblliotheken `ggplot2` (Kapitel 18), `dplyr` (Kapitel 14), `tidyr` (Kapitel 16), `readr` (Kapitel 7), `stringr` (Kapitel 9), `forcats` (Kapitel 10), `lubridate` und `purrr` (Kapitel 8) sowie etliche weitere Module für die tägliche Arbeit mit Daten.

Weil die `tidyverse`-Bibliothek eine zentrale Bedeutung im R-Umfeld hat, ist es an dieser Stelle sinnvoll, die `tidyverse`-Bibliothek mithilfe von `Rscript` zu installieren.

Beispiel 2.3 (Installieren der tidyverse-Bibliotheken unter MacOS)**.**

```
Rscript -e 'install.packages("tidyverse")'
```

Bei der ersten Installation einer Bibliothek fragt R nach einem CRAN-Mirror. Hier sollte ein geografisch nahe Quelle gewählt werden, um die Ladezeiten zu verringern.

Mit Visual Studio Code können R-Bibliotheken auch mit der Arbeitsumgebung installiert werden. Dazu wird die R-Erweiterung geöffnet und im Bereich `Help Pages` die Option

`Install CRAN Package` gewählt. Anschliessend wird der gewünschte Bibliotheksname in der interaktiven Suche eingegeben und mit der Eingabetaste ausgewählt (Abbildung 2.4).

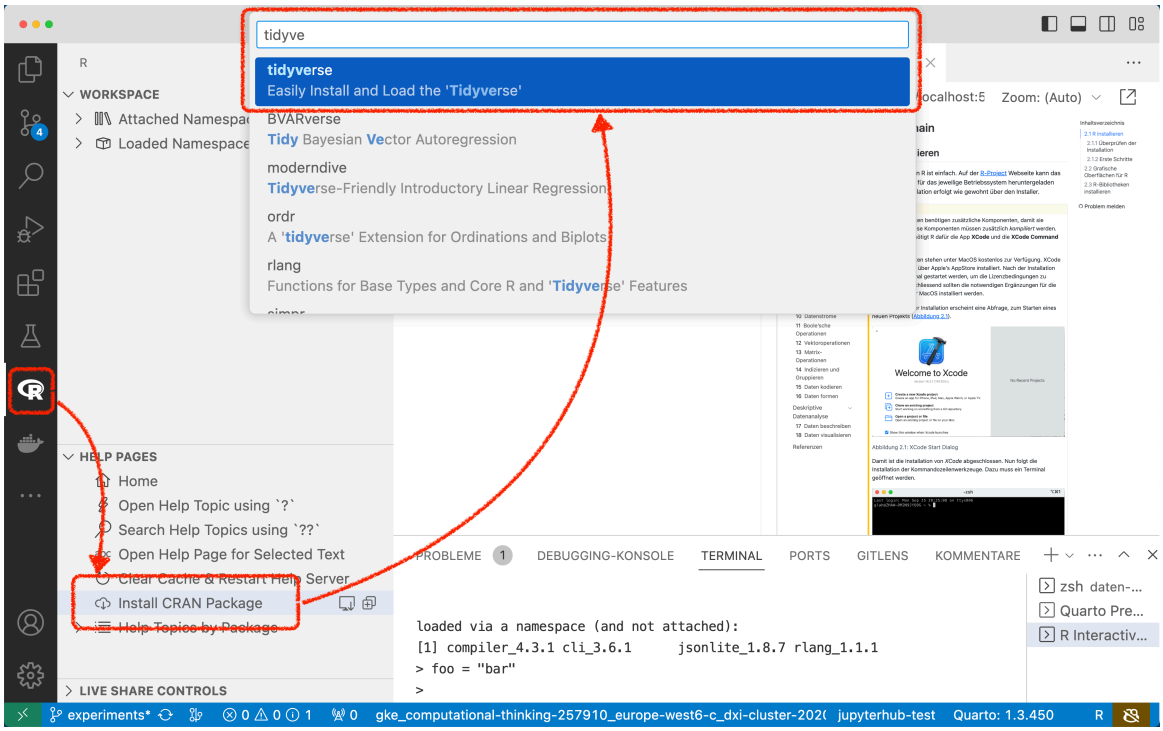


Abbildung 2.4.: Installation der `tidyverse`-Bibliothek in Visual Studio Code

3. Hilfe bekommen

R-Funktionen sind in der Regel gut dokumentiert. Neben der eigentlichen Funktionsbeschreibung finden sich viele ausführliche Problemlösungsstrategien in Form sog. *Vignettes*. Zusätzlich finden sich für die `tidyverse`-Bibliotheken sog. *Cheat Sheets*, die einen schnellen Überblick über die Kernfunktionen erlauben.

> **Praxis**
>
> Nutzen Sie die Dokumentation regelmässig, um die richtigen Funktionen für Ihre Problemstellungen auszusuchen. Die verschiedenen Teile der R-Dokumentation helfen Ihnen die Konzepte und Techniken für die Arbeit mit R zu vertiefen.

3.1. `help()`

Die `help()`-Funktion ist der erste Anlaufpunkt, um mehr über eine Funktion zu erfahren.

R-Funktionen sind in der Regel sehr ausführlich dokumentiert. Falls Sie Details über die Arbeitsweise einer Funktion erfahren möchten, können Sie die Dokumentation einer Funktion mit der `help()`-Funktion abrufen. Dazu rufen Sie diese Funktion wie jede andere R-Funktion auf.

Die `help()`-Funktion ist Teil von `Base R` und ist in jeder Umgebung verfügbar.

Die Funktion erwartet den gewünschten Funktionsnamen. `help()` kann der Funktionsname direkt oder als Zeichenkette als Parameter übergeben werden. D.h. die beiden folgenden Operationen haben den gleichen Effekt und zeigen die Dokumentation der Funktion `read.csv` an.

Beispiel 3.1 (Hilfe anzeigen)**.**

```
help(read.csv)
help("read.csv")
```

In Visual Studio Code ist es nicht notwendig, die `help()`-Funktion aufzurufen, weil die Hilfe direkt in die Arbeitsumgebung integriert ist. In R-Scirpten reicht es, den Mauszeiger über eine Funktion zu bewegen. Visual Studio Code zeigt dann die Hilfe direkt im Editor an (Abbildung 3.1). Diese Darstellung wird als *Inline-Hilfe* bezeichnet.

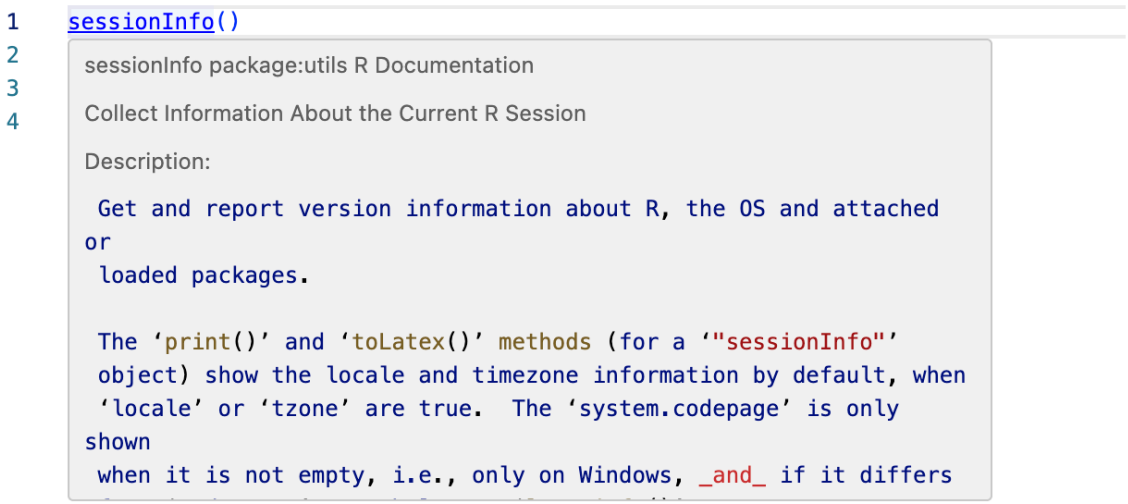


Abbildung 3.1.: Inline Anzeige einer R-Funktionsdokumentation in Visual Studio Code

Neben der *Inline-Hilfe* lassen sich alle Funktionen der installierten R-Bibliotheken auch über den Abschnitt `Help Pages` der R-Erweiterung zugreifen. Dort findet sich unter dem letzten Punkt `Help Topics by Packages` die Dokumentation für alle auf dem Computer installierten Bibliotheken. Der erste Unterpunkt für jede Bibliothek ist der Index, der alle Dokumente für eine Bibliothek auflistet (Abbildung 3.2). Nach dem Installieren einer Bibliothek sollte diese Seite aufgerufen werden, um sich mit der installierten Version vertraut zu machen.

> **!** Achtung
>
> Im Internet finden sich viele Materialien zur Verwendung einzelner Bibliotheken. Oft beziehen sich diese Materialien auf ältere Versionen der jeweiligen Bibliothek. Damit ist nicht sichergestellt, dass die beschriebenen Techniken der richtigen Vorgehensweise entsprechen. **Deshalb sollte immer die offizielle Dokumentation der installierten Bibliotheken zur Überprüfung der beschriebenen Methoden herangezogen werden.**

3.1.1. Aufbau von Funktionsdokumentationen

Die meisten R-Bibliotheken folgen einer Konvention zur systematischen Dokumentation von Funktionen. Jede Funktionsdokumentation besteht aus den folgenden Teilen:

1. Beispielen für den Aufruf der Funktion
2. Beschreibung aller Funktionsparameter
3. Einer detaillierten Funktionsbeschreibung
4. Beispielen

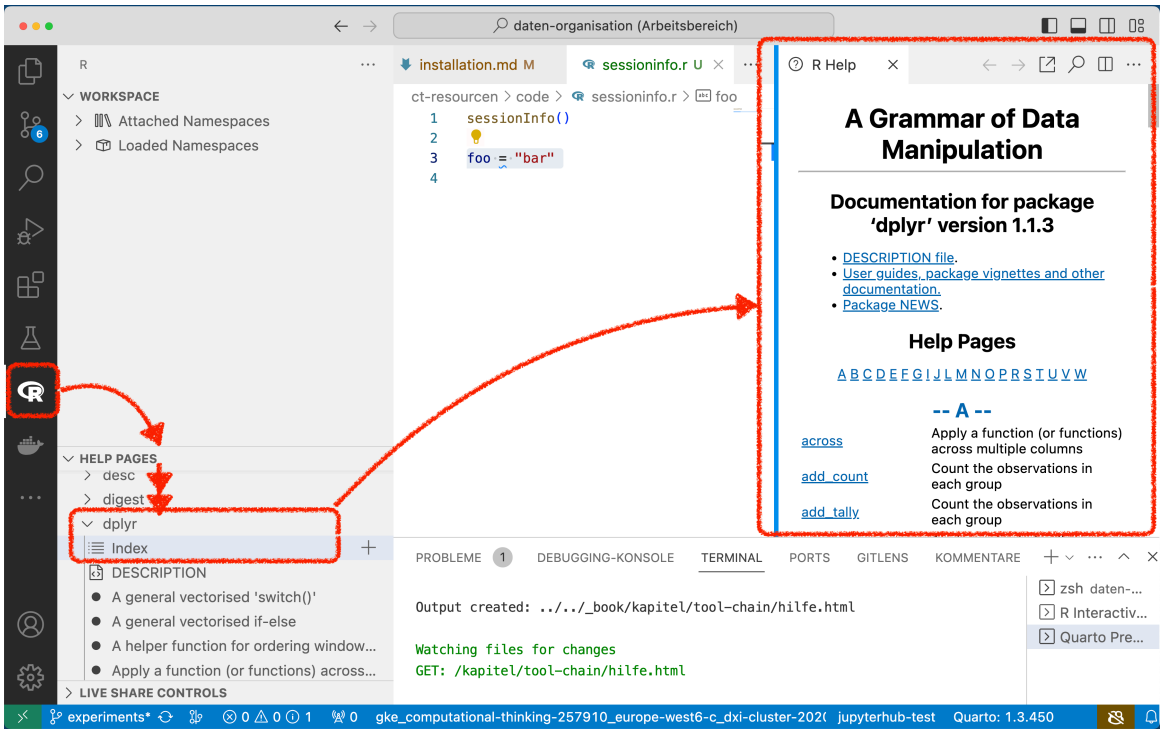


Abbildung 3.2.: Index der Dokumentation für die Bibliothek `dplyr`

Die Beispiele zeigen typische Aufrufe der jeweiligen Funktion und finden sich **immer** *am Ende* der Dokumentation. Es lohnt sich häufig zuerst die Beispiele anzusehen und danach die Funktionsdetails zu lesen.

3.2. Vignettes

Viele R-Bibliotheken haben komplexe Anwendungen. Diese Anwendungen werden in sogenannten *Vignettes* beschrieben. Eine *Vignette* ist eine ausführliche Beschreibung einer Funktion oder des Zusammenspiels mehrerer Funktionen mit nachvollziehbaren Beispielen.

Sie können sich die verfügbaren Vignettes für eine Bibliothek mit der Operation `vignette(package = bibliotheksname)` anzeigen lassen. Wenn Sie z.B. alle Vignettes für die dplyr Bibliothek anzeigen lassen möchten, dann geben Sie `vignette(package = "dplyr")` ein. Das Ergebnis ist die Liste der verfügbaren Vignettes für diese Bibliothek.

Wenn Sie das gesuchte Thema gefunden haben, dann können Sie sich die Vignette mit dem folgenden Befehl anzeigen lassen: `vignette(thema, package = bibliotheksname)`

In Visual Studio Code sind alle Vignettes einer Bibliothek (Abbildung 3.3) über deren Dokumentationsindex (Abbildung 3.2) erreichbar. Dadurch lassen sich Anleitungen oft leichter finden.

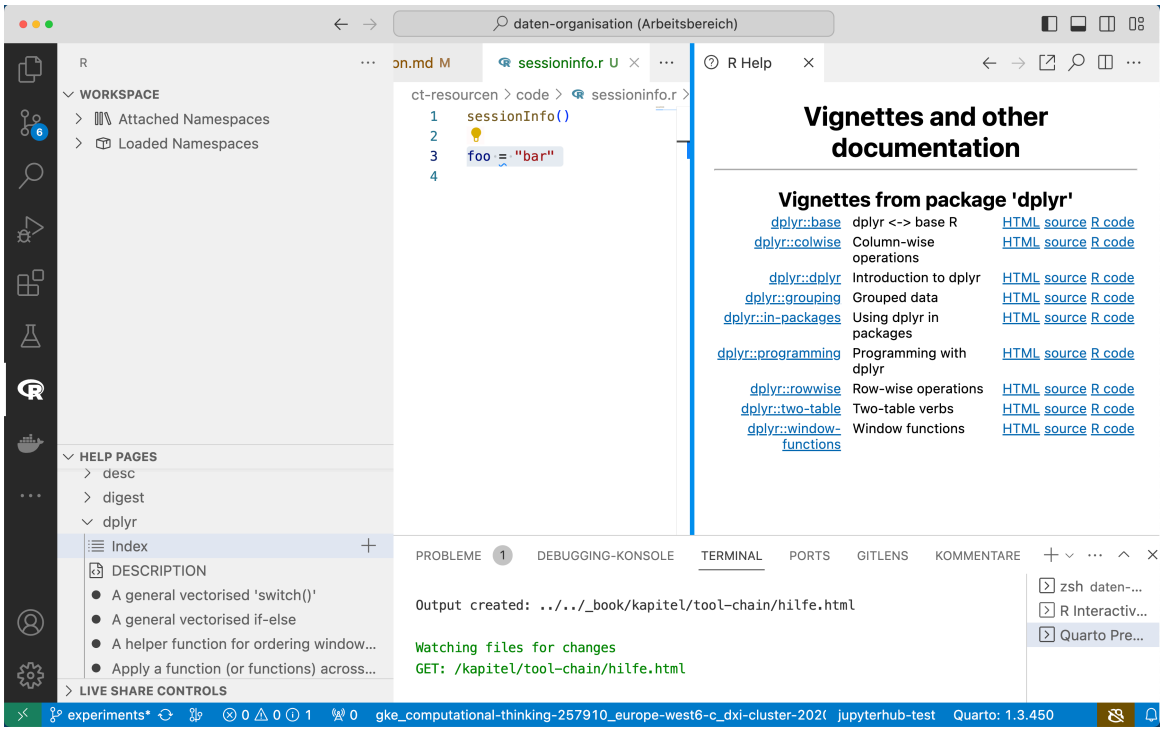


Abbildung 3.3.: Index der `dplyr` Anleitungen

3.3. Cheat Sheets

Die *tidyverse*-Bibliotheken bieten zusätzlich *Spickzettel* für die wichtigsten Funktionen und Techniken für eine Bibliothek auf zwei Seiten. Diese Spickzettel werden auch als *Cheat Sheets* bezeichnet. Sie können diese Cheat Sheets doppelseitig ausdrucken und als Schnellreferenz verwenden.

Im Git-Repository rstudio/cheatsheets finden sich Spickzettel und Kurzreferenzen viele R-Bibliotheken.

> **!** Achtung
>
> Die Spickzettel sind **nicht** Teil der Dokumenation einer Bibliothek und werden nicht mit ihr gepflegt.
> **Ein Spickzettel ersetzt nicht die Dokumentation!** Gelegentlich verweisen Spickzettel auf stark veraltete Praktiken. Es ist also immer ein Vergleich mit der offiziellen Dokumentation notwendig.

Die folgenden Spickzettel unterstützen die Arbeit mit diesem Buch:

- Datenimport
- Datenvisualisierung (ggplot2)
- Datentransformation (dplyr)

- Datenbereinigung (tidyr)
- Vektorfunktionen (purrr)
- Zeichenketten (stringr)
- Faktoren (forcats)
- Datumswerte (lubridate)

4. R-Sprachelemente

Jede Programmiersprache besteht aus festgelegten *Symbolen* und *Operatoren*. Diese beiden Elemente bilden zusammen die **Syntax** einer Programmiersprache mit der *Operationen* erstellt werden können. Syntaktische Kern von R ist vergleichsweise kompakt und übersichtlich.

4.1. Syntaktische Symbole

Syntaktische **Symbole** sind alle Sprachelemente, die für sich allein stehen. Symbole können Werte, Schlüsselworte oder Bezeichner sein. Symbole werden durch Operatoren oder durch Leerzeichen voneinander getrennt.

4.1.1. Werte

Werte können direkt angegeben werden. Dabei legt der Datentyp eines Werts (Kapitel 6) fest, wie dieser eingegeben werden muss.

- Zahlen werden als Ziffernfolge direkt eingegeben (z.B. `123.45`).
- Wahrheitswerte werden in Grossbuchstaben eingegeben (z.B. `WAHR`).
- Zeichenketten werden in Anführungszeichen eingegeben (z.B. `"Daten und Information"`)

4.1.2. Schlüsselworte

R kennt verschiedene Schlüsselworte, die als eigene Symbole festgelegt sind und nicht verändert werden können. Jedes der folgenden Schlüsselworte hat eine Bedeutung für R und kann nur in dieser Bedeutung verwendet und nicht umdefiniert werden.

Funktionen (Kapitel 8)

- `function`

Entscheidungen (Kapitel 11) zur bedingten Ausführung von Operationen.

- `if`
- `else`

Schleifen zur wiederholten Ausführung von Operationen.

- `repeat`
- `while`
- `for`
- `next`
- `break`

> **!** Keine Schleifen in der Paxis
>
> R-Schleifen sind im Vergleich zu vergleichbaren Funktionen ***äusserst ineffizient***. Deshalb haben Schleifen in der praktischen Anwendung von R **keine Bedeutung** mehr. Stattdessen werden ausschliesslich spezielle Funktionen über Datenstrukturen verwendet. Entsprechend finden sich diese Schlüsselworte sehr selten in R-Skripten.

4.1.3. Bezeichner

Ist ein Symbol kein Schlüsselwort und kein Wert, dann wird das Symbol als **Bezeichner** behandelt. Ein Bezeichner ist ein Platzhalter für einen Wert (Kapitel 6) und kann wie ein Wert in Operationen verwendet werden.

Grundsätzlich können Bezeichner beliebige Zeichen enthalten. R erkennt einfache Bezeichner mit zwei Regeln.

- Beginnt mit einem ASCII-Buchstaben (A-Z, a-z) oder einem Unterstrich.
- Enthält nur ASCII-Buchstaben (A-Z, a-z), arabische Ziffern (0-9) und Unterstriche.

Weicht ein Bezeichner von diesen Regeln ab, dann muss dieser als solcher durch Backticks (`) gekennzeichnet werden. Beispiel 4.1 zeigt einen Bezeichner, der den deutschen Umlaut (ü) und ein Leerzeichen enthält. Weil `ü` kein ASCII-Buchstabe ist und das Leerzeichen normalerweise Symbole trennt, muss der ganze Bezeichner als Symbol markiert werden.

Beispiel 4.1 (markierter Bezeichner)**.**

```
`merkwürdiger Bezeichner`
```

Wird der Bezeichner aus Beispiel 4.1 nicht markiert, erzeugt R die Fehlermeldung `Fehler: unerwartetes Symbol in "merkwürdiger Bezeichner"`.

4.2. Operationen

Definition 4.1. Eine **Operation** ist ein syntaktisches Konstrukt, das von einer Programmiersprache als ausführbar erkannt wird.

Eine Operation kann sich als ein Satz in einer Sprache vorgestellt werden, wobei für Programmiersprachen nur **ganze (vollständige) Sätze** gültig sind und ausgeführt werden.

R fügt Symbole und Operatoren solange zusammen, bis eine **syntaktisch gültige** Operation gefunden wird. Syntaktisch gültig heisst in diesem Zusammenhang, dass alle syntaktischen Elemente für eine Operation gefunden wurden. Erst dann versucht R diese Operation auszuführen. Bei der Ausführung kann festgestellt werden, dass eine Operation nicht ausführbar ist. In diesem Fall erzeugt R eine **Fehlermeldung**.

Die einfachste R-Operation ist die Angabe eines einzelnen Symbols. Wird nur ein Symbol als Operation eingegeben, dann bedeutet das für R, dass die zum Symbol gehörenden Daten **serialisiert** werden sollen. Das bedeutet, dass die Daten des Symbols angezeigt werden. Ist das Symbol ein Wert, dann wird der Wert wiederholt.

Beispiel 4.2 (Wert direkt serialisieren)**.**

```
"Daten und Information"
```

Beispiel 4.2 zeigt die Operation, die die Zeichenkette *Daten und Information* als Ergebnis serialisiert.

4.2.1. Operatoren

Definition 4.2. Ein **Operator** ist ein syntaktisches Element, das Symbole zu einer Operation *verknüpft*.

Die bekanntesten Operatoren sind die arithmetischen Operatoren und die Vergleichsoperatoren.

In R sind vier spezielle Operatoren wichtig:

- Die drei Zuweisungsoperatoren (`<-`, `=` und `->`).
- Der Funktionsaufrufoperator (`()`).

Mit den *Zuweisungsoperatoren* können Werte Namen zugewiesen werden. Bei den Pfeil-Operatoren wird der Wert in Richtung des Pfeils zugewiesen. Beim Gleich-Operator erfolgt die Zuweisung von rechts nach links. (s. Beispiel 4.3)

Definition 4.3. Eine **Deklaration** heisst die (erste) Zuweisung eines Werts an eine Variable.

Beispiel 4.3 (Zuweisung einer Zeichenkette)**.**

```
buchtitel1 = "Daten und Information 1"
buchtitel2 <- "Daten und Information 2"
"Daten und Information 3" -> buchtitel3
```

Es können auch Namen anderen Namen zugewiesen werden. In diesem Fall wird der zugehörige Wert für einen Namen ermittelt und dem neuen Namen zugewiesen (Beispiel 4.4).

Beispiel 4.4 (Zuweisung von Namen an einen anderen Namen.)**.**

```
buchtitel4 = buchttitel1
# buchtitel4 enthält "Daten und Information 1"
buchtitel5 <- buchtitel2
# buchtitel5 enthält "Daten und Information 2"
buchtitel3 -> buchtitel6
# buchtitel6 enthält "Daten und Information 3"
```

Der *Funktionsaufrufoperator* prüft ob der vorangehende Name eine Funktion ist und ruft diese auf. Zwischen den Klammern können Werte oder Namen als Parameter der Funktion übergeben werden. Die Parameter werden durch Kommas getrennt (Beispiel 4.5).

Beispiel 4.5 (Funktionsaufruf der Summefunktion mit Parametern)**.**

```
sum(1,2,3) # 6
```

Der Funktionsaufrufoperator darf nicht mit den aus der Mathematik bekannten Klammern verwechselt werden. Klammern fassen auch in R Teiloperationen zusammen und reihen diese gegenüber einer anderen Teiloperation vor. Der Funktionsaufrufoperator kommt nur zur Anwendung, wenn ein Name auf eine Funktion verweist.

Wird der Funktionsaufrufoperator zusammen mit dem Schlüsselwort `function` verwendet, dann wird eine Funktion mit den angegebenen Parametern erstellt (s. Beispiel 4.6). Diese Operation heisst **Funktionsdeklaration**. Eine deklarierte Funktion muss in R einem Namen zugewiesen werden.

Beispiel 4.6 (Funktionsdeklaration)**.**

```
add3 <- function (eins, zwei, drei)
    sum(eins, zwei, drei)
```

4.2.2. Blöcke

Mehrere Operationen lassen sich in R zu *Blöcken* zusammenfassen. Ein Block wird durch geschweifte Klammern (`{` und `}`) markiert. Geschweifte Klammern bilden also den **Blockoperator**.

Ein Block wird von R als eine Operation behandelt. Damit die Operationen in einem Block von R ausgeführt werden können, müssen alle übergeordneten Blöcke geschlossen werden.

Beispiel 4.7 zeigt die Deklaration einer Funktion, die aus zwei Operationen besteht. Damit beide Operationen zu einer Funktion zusammengefasst werden können, müssen diese in einen Block gefasst werden.

Beispiel 4.7 (Funktionsdeklaration mit Block)**.**

```
add3mod6 <- function (eins, zwei, drei) {
    sum(eins, zwei, drei) -> sechs
    sechs %% 6
}
```

Teil I.

Datenquellen

5. Dokumentation

Traditionell wurde die Auswertung von Daten und die Dokumentation in getrennten Arbeitsschritten durchgeführt. Dazu wurden während der Auswertung mussten alle notwendigen Materialien für Berichte und Publikationen erzeugt werden und anschliessend während der Dokumentation in ein anderes Dokument eingebettet werden. Diese Trennung der Arbeitsschritte erforderte von Autor:innen bei der Dokumentation grosse Sorgfalt, weil die analytische Vorgehensweise aus den erzeugten Materialien nicht mehr nachvollziehbar war und nicht erzeugte Materialien oft ohne konkrete Belege berichtet wurde.

Mit zunehmender Bedeutung der Datenwissenschaften für industrielle Anwendungen, wurde das Bedürfnis nach Werkzeugen zur schnellen Erstellung von daten-basierten Berichten immer grösser, weil erstens regelmässige Berichte mit gleichen analytischen Methoden erstellt werden müssen und zweitens die aktuellen wissenschaftlichen Standards für jede Aussage in einem analytischen Bericht Belege in der Datenbasis erfordern. Dieses Bedürfnis ist nicht neu und einzelne Speziallösungen lassen sich bis in die Frühzeit der Digitalisierung zurückverfolgen. Jedoch wurden erst seit Mitte der 2010er-Jahre Werkzeuge zur datenbasierten Dokumentation entwickelt und systematisch eingesetzt. Diese Entwicklungen münden in den aktuell verwendeten *Datendokumenten*.

Definition 5.1. Ein **Datendokument** ist ein Dokument, dass Datentransformationen, -Visualisierungen und -Auswertungen in die Dokumentation einbindet.

Datendokumente eignen sich besonders für Labor- oder Projektberichte, weil die Auswertung direkt in den Bericht einfliesst. Datendokumente verbinden sog. beschreibenden Text mit Code-Fragmenten, so dass die Ausgabe der Code-Fragmente direkt Teil des Berichts wird.

Datendokumente verbinden beschreibende und formatierte Textblöcke mit Code-Blöcken. Die Formatierung der Textblöcke erfolgt in der Regel mithilfe von Markdown. Markdown hat gegenüber anderen Textformatierungen den Vorteil, dass es ähnlich wie Programm-Code leicht erlernbar und leicht versionierbar ist.

Datendokumente werden in der Regel als Web-Seiten oder als PDF-Dokumente bereitgestellt. Dieses Bereitstellen ist die **Präsentation** eines Datendokuments.

Bei Datendokumenten werden zwei leicht unterschiedliche Ansätze verfolgt.

- ***R-Markdown*** (Grolemund, 2014) verwendet ausschliesslich Markdown als Dokumentenformat. Die Ergebnisse von Code-Blöcken sind nicht Bestandteil des eigentlichen Dokuments, sondern werden erst für die Präsentation erzeugt. Die Codezellen bilden gemeinsam ein Programm bzw. Script, dass immer in der Reihenfolge der Code-Zellen

ausgeführt wird. Ein R-Markdown-Dokument ist immer ein gültiges Markdown Dokument und kann mit jedem Markdown-fähigen Betrachter (z.B. GitHub) angezeigt werden.

- ***Jupyter Notebooks*** (Jupyter Development Team, 2015) unterscheiden zwischen *Textzellen* und *Code-Zellen*. Textzellen enthalten nur in Markdown formatierte Textelemente. Werden Markdown-Code-Blöcke in Textzellen verwendet, dann werden diese nicht ausgeführt. Code-Zellen enthalten nur ausführbaren Code. Wird dieser ausgeführt werden, dann werden die Ergebnisse als Teil des Dokuments gespeichert. Die Code-Zellen eines Jupyter-Notebooks können in beliebiger Reihenfolge ausgeführt werden. Deshalb wird zusätzlich für ausgeführte Code-Zellen eine Nummer festgehalten, aus der die Reihenfolge der Ausführung hervorgeht. Jupyter Notebooks verwenden ein spezielles Datenformat und benötigen spezielle Betrachter-Software.

> **i** Hinweis
>
> Der Begriff *R-Markdown* wird hier als Referenz für den verfolgten Ansatz verwendet. Datendokumente in R-Markdown sind nicht auf R beschränkt, sondern können beliebige andere Programmiersprachen in Code-Blöcken verwenden.

> **i** Hinweis
>
> Weil *Jupyter Notebooks* die Ergebnisse von Code-Zellen im Dokument speichern, ändert sich das Dokument, selbst wenn keine inhaltlichen Änderungen vorgenommen wurden. Diese Eingenschaft von Jupyter Notebooks erschwert die Versionierung ein wenig.

Für Laborberichte können Datendokumente direkt verwendet werden. Für die Publikation von Projektberichten sind diese Systeme nicht unmittelbar geeignet, sondern müssen durch zusätzliche Tools ergänzt werden, um die zusätzlichen Elemente dieser Berichte erzeugen zu können. Dazu gehören Bild- und Tabellenbeschriftungen, Querverweise, Quellenverweise und das Literaturverzeichnis sowie Inhaltsverzeichnisse und Indizes.

R-Markdown-Dokumente können mit der Software `quarto` (Posit Software PBC, 2023) in ein geeignetes Präsentationsformat gebracht werden.

Für *Jupyter Notebooks* übernimmt diese Funktion das Werkzeug `Jupyter Books` (The Jupyter Book Community, 2023).

5.1. Mathematische Formeln in Datendokumenten

In Datendokumenten lassen sich mathematische Formeln darstellen. Diese Formeln werden im sog. **LaTeX-Math-Mode** eingegeben. Diese Formeln werden bei der Präsentation in die korrekte mathematische Darstellung überführt.

Der LaTeX-Math-Mode ist eine Formelbeschreibungssprache (American Mathematical Society & LATEX Project, 2020; Høgholm & Madsen, 2022), mit der die exotischen und in

nicht mathematischen Texten wenig bis nie verwendeten Symbole und deren Anordnung gezielt erzeugt werden können. Der LaTeX-Math-Mode wird von vielen Systemen zur Darstellung von Formeln verwendet. Deshalb lohnt sich eine Auseinandersetzung mit den Grundkonzepten dieser Technik.

Der LaTeX-Math-Mode kennt zwei Modi: den **inline Modus**, wenn eine Formel wie oben in den Fliesstext eingebettet ist, und den **Gleichungsmodus**, wenn eine Formel wie eine Abbildung hervorgehoben und beschriftet wird. Der inline Modus wird durch ein einfaches Dollar-Zeichen (`$`) oder mit der Zeichenfolge Backslash-runde Klammer (`\(` und `\)`) eingeleitet und abgeschlossen. Der Gleichungsmodus wird durch ein doppeltes Dollar-Zeichen (`$$`) eingeleitet und abgeschlossen. Die Formelbeschreibung ist unabhängig vom Modus, wobei die Darstellung dem zur Verfügung stehenden Platz berücksichtigt.

Mit dem LaTeX-Math-Mode wird die Formel mit ihren Bestandteilen und ihren Beziehungen beschrieben. Die folgenden Grundregeln sind für den Math-Mode wichtig:

- Zahlen und Buchstaben und andere Sonderzeichen auf der Tastatur werden als solche dargestellt.
- Sonderzeichen, Operatoren und besondere Formatierungen werden durch einen Back-Slash (\) eingeleitet, der von einem Schlüsselwort gefolgt wird.
- zusammengehörende Teilausdrücke werden durch geschweifte Klammern zusammengefasst (`{}`). Teilausdrücke, die nur aus einem Symbolbestehen müssen nicht in Klammern gesetzt werden.
- Das Dach (`^`) bedeutet den folgenden Teilausdruck hochstellen.
- Der Unterstrich (`_`) bedeutet den folgenden Teilausdruck tiefstellen.

Beispiel 5.1 (LaTeX-Math-Mode)**.** Die Formel $\sum_{i=1}^{n} (\frac{x_i}{2})^2$ lässt sich mit einer normalen Tastatur nicht eingeben. Deshalb wird der Math-Mode zur Formelbeschreibung verwendet. Im inline Modus wird die Formeldarstellung so gewählt, dass die Formel ungefähr in die aktuelle Textzeile passt. Im Gleichungsmodus wird die gleiche Formel möglichst übersichtlich angezeigt.

$$\sum_{i=1}^{n} (\frac{x_i}{2})^2$$

Die Formel wird wie folgt im Math-Mode beschrieben:

```
\sum_{i=1}^{n}{(\frac{x_i}{2})^2}
```

Die Formel beginnt mit dem grossen griechischen Sigma (Σ) für das Summensymbol. Das wird durch `\sum` erzeugt. Eine Summe besteht aus drei Teilausdrücken:

1. Dem Initialwert *unter* dem Summenzeichen.
2. Dem Endwert *über* dem Summenzeichen.
3. Dem Summenterm hinter dem Summenzeichen.

Entsprechend wird der Initialwert mit `_` tiefgestellt, der Endwert mit `^` hochgestellt und der Summenterm wird hinter die Summe gefügt.

Der Summenterm wird durch eine Potenz und einem Bruch gebildet. Die Potenz wird durch das Hochstellen gekennzeichnet. Die runden Klammern werden als einfache runde Klammern eingegeben

Der Bruch ist eine besondere Darstellung und benötigt das Schlüsselwort `frac` für *fraction* (dt. Fraktur/Bruch). Ein Bruch besteht immer aus zwei Teilausdrücken, die nacheinander in geschweiften Klammern angegeben werden müssen. Der Zähler besteht aus dem Teilausdruck x_i. Entsprechend muss das i gegenüber dem x tiefgestellt werden. Weil es sich jeweils um einzelne Symbole handelt, müssen die Teilausdrücke nicht geklammert werden.

6. Datentypen

R ist eine **vektororientierte Programmiersprache**. Das bedeutet zum einen, dass alle skalaren Werte von R wie ein-dimensionale Vektoren behandelt werden. Zum anderen ist die Syntax von R auf die Arbeit mit Vektoren optimiert, so dass sich entsprechende Aufgaben in R leichter ausdrücken lassen als in anderen (nicht-vektororientierten) Programmiersprachen.

Die Idee der Vektororientierung hat Auswirkungen auf die Datentypen, denn die fundamentalen Datentypen legen in R nur die *zulässigen Wertebereiche* fest. Die Werte selbst werden von R immer als Datenstruktur behandelt, auch wenn diese nicht explizit als solche gekennzeichnet wurden. Bei der praktischen Arbeit tritt diese Besonderheit meist nicht in den Vordergrund und wird in der R-Syntax nicht hervorgehoben. Die entsprechenden Sonderfälle werden im Kapitel 12 behandelt.

6.1. Fundamentale Datentypen

6.1.1. Undefinierte Werte

R kennt zwei voneinander verschiedene Werte für undefinierte Werte.

- `NULL`
- `NA`

Weder `NULL` noch `NA` sind in R gleichwertig mit dem Wert `0`. Die beiden Werte sind ausserdem nicht gleich und haben eine leicht unterschiedliche Bedeutung.

`NULL` bedeutet, dass kein Wert vorhanden ist *und* kein Datentyp bekannt ist. Dieses Symbol ist für die Programmierung von Bedeutung und zeigt für eine Variable (Kapitel 8) an, dass diese auf keinen Wert im Speicher des Computers verweist. Das Symbol `NULL` ist ein eigener Datentyp.

`NA` (für *not available*) bedeutet, dass ein Wert fehlt, obwohl ein Wert erwartet wird. In diesem Fall ist der Datentyp bekannt. Dieser Wert bezieht sich auf die Daten und zeigt fehlende Werte an. Der Wert `NA` hat einen beliebigen Datentyp ausser `NULL`. `NA` ist ein Platzhalter für fehlende Werte in Daten, der immer ausserhalb des gültigen Wertebereichs der Daten liegt. Damit müssen fehlende Werte in R nicht durch einen alternativen Wert ersetzt werden.

Beim Zählen von Werten werden `NA`-Werte mitgezählt. Für mathematische Operationen, wie der Addition oder der Multiplikation, führt ein Operand mit Wert `NA` als Operand

immer zum Ergebnis `NA`. Deshalb müssen `NA` Werte vor allen anderen Operationen behandelt werden.

6.1.2. Zahlen

R unterscheidet drei Arten von Zahlen:

- Gleitkomma (`numeric()`)
- Ganzzahlen (`integer()`)
- Komplexe Zahlen (`complex()`)

Standardmässig werden alle Zahlenwerte als **Gleichkommazahlen** erstellt. Gleitkommazahlen können sowohl direkt oder in wissenschaftlicher Notation eingegeben werden. Bei der wissenschaftlichen Notation **muss** immer der Nachkommaanteil angegeben werden.

Beispiel 6.1 (Zahlenbeispiele in R)**.**

```
5
5.374
3.92e+2
1.0e-5
```

Das kann mit der Funktion `is.numeric()` überprüft werden. Diese Funktion liefert Wahr zurück, wenn ein Wert eine Gleitkommazahl ist oder als solche behandelt werden kann.

Die Werte anderer Datentypen lassen sich mit der Funktion `as.numeric()` in Zahlen umwandeln. Lässt sich ein Wert nicht eine Zahl umwandeln, dann wird der Wert `NA` mit einer entsprechenden Warnung erzeugt.

Gelegentlich müssen **ganzzahlige Werte** sichergestellt werden. Diese Werte werden aus Gleitkommazahlen mit der Funktion `as.integer()` erzeugt. Wird eine Gleitkommazahl in eine Ganzzahl umgewandelt, dann wird nur der ganzzahlige Teil behalten. Der Nachkommateil wird gestrichen und nicht gerundet. Grundsätzlich sind alle Längen und alle Indizes in R automatisch Ganzzahlen und müssen nicht umgewandelt werden.

Komplexe Zahlen sind spezielle Zahlen, die über dem Zahlenraum der reellen bzw. Gleitkommazahlen hinausgehen. Eine Komplexe Zahl besteht aus einem reellen und einem *imaginären* Zahlenanteil. Der imaginäre Zahlenanteil ist als ein Vielfaches der Zahl $i = \sqrt{-1}$ definiert. In R wird diese Beziehung der beiden Teile als Summe dargestellt.

Beispiel 6.2 (Darstellung einer komplexen Zahl)**.**

```
3+7i
```

Alternativ können komplexe Zahlen mit der Funktion `complex()` erzeugt werden. Dazu werden bei beiden Zahlenanteile getrennt über die Parameter `real` und `imaginary` angegeben, wobei beim imaginären Teil das nachfolgende `i` durch die Funktion eingefügt wird.

Beispiel 6.3 (Erzeugen eines Vektors von komplexen Zahlen mit `complex()`).

```
complex(real = c(2,8), imaginary = c(3, 7))
```

```
[1] 2+3i 8+7i
```

Für reelle Zahlen gilt, dass der imaginäre Anteil gleich 0 ist. Mit der Funktion `as.complex()` lässt sich jede Zahl in eine komplexe Zahl umwandeln. Wird eine so erstelle komplexe Zahl mit der ursprünglichen reellen Zahl verglichen, dann zeigt R korrekt an, dass die beiden Werte gleich sind.

> Praxis
>
> In der Praxis wird die Umwandlung von reellen in komplexe Zahlen R überlassen.

Beispiel 6.4 (Umwandeln einer reelen Zahl in eine komplexe Zahl).

```
as.complex(2.2)
```

```
2.2+0i
```

Neben den klassischen Zahlen verfügt R über das Konzept **Unendlich**, das in vielen Programmiersprachen fehlt. Ein unendlicher Wert wird durch das Symbol `Inf` dargestellt. Weil sowohl positive als auch negative unendliche Werte existieren steht `Inf` für *positiv unendlich* und `-Inf` für *negativ unendlich*.

6.1.3. Zeichenketten

Zeichenketten heissen in R *Character-Strings* (`character()`). Zeichenketten müssen immer in Anführungszeichen eingefasst werden. Als Anführungszeichen dürfen das einfache Anführungszeichen (`'`) oder das doppelte Anführungszeichen (`"`) verwendet werden.

Beispiel 6.5 (Gleichwertige Zeichenketten mit einfachen und doppelten Anführungszeichen).

```
'Daten und Information'
"Daten und Information"
```

> Praxis
>
> Die Lesbarkeit von R-Code wird dadurch verbessert, dass konsequent nur ein Symbol für die Kennzeichnung von Zeichenketten verwendet wird. Die Wahl welches der bei-

den Zeichen benutzt wird, wird von persönlichen Vorlieben geleitet. In diesem Buch wird konsequent das doppelte Anführungszeichen (") eingesetzt, weil dadurch leere Zeichenketten direkt als solche erkennbar sind und dieses Symbol auch in anderen Programmiersprachen und Dateiformaten zur Kennzeichnung von Zeichenketten benutzt wird.

Beispiel 6.6 (Leere Zeichenketten mit einfachen und doppelten Anführungszeichen)**.**

```
''
""
```

Um zu überprüfen, ob ein Wert eine Zeichenkette ist, wird die Funktion `is.character()` verwendet. Diese Funktion ergibt Wahr, wenn der Wert eine Zeichenkette ist und in allen anderen Fällen Falsch.

Um einen anderen Wert in eine Zeichenkette umzuwandeln bzw. zu *serialisieren*, wird die Funktion `as.character()` eingesetzt.

Beispiel 6.7 (Eine Zahl in eine Zeichenkette serialisieren)**.**

```
as.character(123.4)
# ergibt "123.4"
```

6.1.4. Wahrheitswerte

R kennt die beiden Wahrheitswerte *Wahr* und *Falsch* bzw. die englischen Begriffe **True** und **False**. In R müssen Wahrheitswerte (`logical()`) immer in Grossbuchstaben geschrieben werden. Der Wert *Wahr* wird also `TRUE` und der Wert *Falsch* wird `FALSE` geschrieben.

Beide Wahrheitswerte dürfen mit dem Anfangsbuchstaben abgekürzt werden. Auch dieser Buchstabe muss gross geschrieben werden. Die Werte `T` und `TRUE` sowie `F` und `FALSE` sind deshalb immer gleich.

Die Werte `TRUE` und `FALSE` dürfen nicht in Anführungszeichen eingefasst werden, denn sonst werden sie als Zeichenketten und nicht als Wahrheitswerte interpretiert.

Normalerweise müssen Wahrheitswerte nicht mit `is.logical()` geprüft oder mit `as.logical()` aus anderen Datentypen erzeugt werden.

6.1.5. Faktoren

Ein besonderer Datentyp von R sind **Faktoren** (`factor()`).

Definition 6.1. Ein **Faktor** ist ein *diskreter Datentyp* mit einem *festen Wertebereich* mit einer ganzzahligen Ordnung.

Die angezeigten Werte eines Faktors können als Zahlen, Zeichenketten oder Wahrheitswerte dargestellt werden. Jeder Faktor hat einen definierten Wertebereich, wobei die Werte dieses Wertebereichs diskret sind und eine ganzzahlige Ordnung haben. Mit Faktoren können nominal- und ordinalskalierte Datentypen abgebildet werden. Die Ordnung eines Faktor wird für die visuelle Darstellung der Werte verwendet und für Vergleiche von Werten berücksichtigt.

Ein Faktor ist in R ein eigener fundamentaler Datentyp und kann nicht als Datentyp der dargestellten Werte verwendet werden.

Die *Ordnungswerte* eines Faktors lassen sich in Ganzzahlen ausgeben und entsprechend weiter verarbeiten. In diesem Fall muss der Faktor mit der Funktion `as.integer()` in Zahlen umgewandelt werden.

Der *Wertebereich* eines Faktors kann mit der Funktion `levels()` abgefragt werden.

Der Wertebereich eines Faktors ist standardmässig Alphanumerisch geordnet. Für ordinalskalierte Datentypen kann diese Reihung angepasst werden (s. Kapitel 10).

> ⚠ Warnung
>
> Der tatsächliche Wertebereich eines Faktors umfasst immer nur die vorhandenen Werte in den Daten. Die nicht vorkommenden Werte werden von R aus dem Wertebereich eines Faktors entfernt.

6.2. Datenstrukturen

Die zentralen Datenstrukturen von R sind Vektoren, Listen, Matrizen und Data-Frames.

6.2.1. Vektoren

Vektoren sind Datenstrukturen, bei denen alle Elemente vom gleichen Datentyp sind. Alle Werte mit fundamentalen Datentyp sind in R grundsätzlich Vektoren mit einem Element. Diese Eigenschaft mit der Funktion `is.vektor()` überprüft werden. Diese FUnktion hat als Ergebnis Wahr, wenn die Eingabe ein Vektor ist und in allen anderen Fällen Falsch. Wird der Funktion ein Wert übergeben, dann gibt die Funktion Wahr (bzw. `TRUE`) zurück.

Beispiel 6.8 (Ein Wert ist ein Vektor)**.**

```
is.vector(1)
# ergibt TRUE
```

Um Werte zu längeren Vektoren zu Verknüpfen wird die Verbindenfunktion `c()` benutzt. Diese Funktion verbindet Vektoren so dass die Werte der Eingabevektoren im Ergebnisvektore nacheinander in der Reihenfolge der Eingabe vorliegen.

Beispiel 6.9 (Erzeugen eines Vektors aus einzelnen Werten)**.**

```
c(1, 2, 3)
# ergibt {1, 2, 3}
```

Die `c()`-Funktion hat immer einen Vektor als Ergebnis. Werden Vektoren als Eingabe verwendet, dann werden die Vektoren zu *einem* neuen Vektor zusammengefügt.

Beispiel 6.10 (Erzeugen eines Vektors aus mehreren Vektoren)**.**

```
c(c(3, 4), c(1, 2), c(5, 6))
# ergibt {3, 4, 1, 2, 5, 6}
```

Versucht man Vektoren aus Werten mit unterschiedlichen Datentypen zu erstellen, dann werden alle Werte in den allgemeinsten auftredenten Datentyp umgewandelt. Dabei gilt die Reihenfolge vom allgemeinsten Datentyp zum speziellsten Datentyp: Zeichenkette, Zahl, Wahrheitswert.

Beispiel 6.11 (Erzeugen eines Vektors mit unerschiedlichen Datentypen)**.**

```
c(1, TRUE, "Daten und Information")
# ergibt {"1", "TRUE", "Daten und Information"}

c(1, TRUE, 2 FALSE)
# ergibt {1, 1, 2, 0}
```

Alle Vektoren haben eine Länge, die mit der Funktion `length()` ermittelt wird.

Beispiel 6.12 (Länge eines Vektor bestimmen)**.**

```
length(c(1,3,7))
# ergibt 3
```

Damit einzelne Werte eines Vektors angesprochen werden können, müssen eckige Klammern (`[]`) verwendet werden.

Beispiel 6.13 (Ein Vektorelement über dessen Index ansprechen)**.**

```
c(1, 7, 13) -> vektor123

vektor123[2] # ergibt 7
```

> Praxis
>
> In der Regel werden die einzelnen Elemente von Vektoren nicht über den Index angesprochen. Im Gegensatz zu anderen Programmiersprachen hat die Verwendung des Vektorindex in R keine grosse Bedeutung.

Vektoren sind eingeschränkt auf *eine* Dimension. Um komplexere Datenstrukturen zu erhalten, müssen weitere Datenstrukturen hinzugezogen werden.

6.2.2. Listen

Listen ähneln Vektoren in vielen Punkten. Der zentrale Unterschied zwischen Listen und Vektoren ist, dass die Elemente von Listen einen *beliebigen Datentyp* haben dürfen. Listen werden in R mit der Funktion `list()` erzeugt.

Beispiel 6.14 (Erzeugen einer Liste aus einzelnen Werten)**.**

```
list(1, TRUE, "Daten und Information")
# erzeugt {1, TRUE, "Daten und Information"}
```

Die Funktion `is.list()` überprüft, ob ein Wert eine Liste ist. Die Funktion `length()` liefert die Anzahl der Elemente in einer Liste.

Im Vergleich zur Funktion `c()` fügt die Funktion `list()` Listen nicht zusammen, sondern behandelt alle Eingabewerte als eigene Elemente. D.h. der Datentyp jedes Werts bleibt erhalten, wenn dieser einer Liste hinzugefügt wird. Diese Eigenschaft lässt sich ausnutzen, um geschachtelte Datenstrukturen zu erzeugen. Beispiel 6.15 zeigt die Erstellung einer geschachtelten Liste mit zwei unterschiedlich langen Vektoren.

Beispiel 6.15 (Geschachtelte Liste mit zwei Vektoren)**.**

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

Der Listen verwenden zur Indizierung doppelte eckige Klammern (`[[]]`). Nur so lassen sich die Werte der Liste korrekt referenzieren.

Beispiel 6.16.

```
list(1, TRUE, "Daten und Information") -> gemischteListe

gemischteListe[[3]]
# ergibt "Daten und Information"
```

> **Warnung**
>
> Der Vektorindex kann auch für Listeneinträge verwendet werden. In diesem Fall wird eine Liste zurückgegeben, die nur die ausgewählten Listenelemente enthält. Das ist normalerweise nicht gewünscht.

6.2.2.1. Benannte Listen

Listeneinträge können *Namen* haben. Diese können später zum Referenzieren der Listeneinträge verwendet werden. Auf diese Weise lassen sich objektartige Strukturen erzeugen. Benannte Einträge lassen sich über die Position des Werts oder dem Namen als Index ansprechen. Wird ein Name als Index verwendet, dann **muss** dieser als Zeichenkette angegeben werden. Häufiger findet sich jedoch die Dollar-Referenzierung in R-Skripten. Die Dollar-Referenzierung verwendet das Dollar-Zeichen (`$`) um auf einen Namen zuzugreifen. Damit diese Referenzierung funktioniert, **muss** der Name ein gültiges R-Symbol sein, dass nicht markiert werden muss. Alle anderen Namen müssen als Listenindex verwendet werden. Beispiel 6.17 zeigt die verschiedenen Zugriffsarten für benannte Listen.

Beispiel 6.17 (Verwendung benannter Listen)**.**

```
list(modul = "Daten und Information",
     semester = 1) ->
        benannteListe

benannteListe[[2]] # ergibt 1
benannteListe[["semester"]] # ergibt 1
benannteListe$semester # ergibt 1
```

> **Hinweis**
>
> In R müssen nicht alle Listeneinträge benannt sein. Es ist normal, dass sowohl benannte als auch unbenannte Listenelemente vorhanden sind. Unbenannte Listeneinträge können nur über ihre Position angesprochen werden.

6.2.3. Matrizen

Als vektororientierte Sprache ist die Matrix ein wichtiges Konstrukt der Datenstrukturierung. Alle Werte einer Matrix müssen vom Datentyp Zahl sein. Man kann sich eine Matrix als Vektor von gleichlangen Vektoren vorstellen. Wegen der besonderen Eigenschaften von R-Vektoren ist diese Art der Schachtelung jedoch nicht möglich.

Eine Matrix wird in R immer aus Vektoren erzeugt. Dafür gibt es vier Wege.

Eine Matrix wird aus einem Vektor über die Zeilenzahl m erstellt. Dabei werden immer m aufeinanderfolgende Werte eines Vektors in eine Spalte geschrieben.

Beispiel 6.18 (eine m-Matrix aus einem Vektor erstellen)**.**

```
matrix(c(1,2,3,4,5,6), nrow = 2)
```

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

Eine Matrix wird aus einem Vektor über die Spaltenzahl n erstellt. Dazu wird der Vektor in n grosse Blöcke aufeinanderfolgender Werte gegliedert, die jeweils in eine Spalte geschrieben werden.

Beispiel 6.19 (eine n-Matrix aus einem Vektor erstellen)**.**

```
matrix(c(1,2,3,4,5,6), ncol = 2)
```

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

Eine Matrix wird über das Kreuzprodukt (Kapitel 13) aus zwei Vektoren erstellt.

Beispiel 6.20.

```
c(1, 2, 3) %*% t(c(3, 4, 5))
```

```
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    8   10
[3,]    9   12   15
```

> **Warnung**
>
> Wird lässt sich der Ausgangsvektor nicht in die angegebene Zeilen- oder Spaltenzahl gliedern, dann werden die Vektorwerte solange wiederholt, bis die Matrix aufgefüllt wurde und eine Warnrmeldung ausgegeben. Dadurch ist das Ergebnis nicht immer klar nachvollziehbar.

Eine Matrix wird über das äussere Matrixprodukt (Kapitel 13) aus zwei Vektoren erstetllt.

Beispiel 6.21 (eine -Matrix aus Vektoren erstellen)**.**

```
c(1, 2, 3) %o% c(3, 4, 5)
```

```
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    8   10
[3,]    9   12   15
```

> **Praxis**
>
> Das Kreuzprodukt sollte nicht zum Erzeugen von Matrizen verwendet, sondern ausschliesslich als mathematische Operation behandelt werden. Das äussere Produkt ist flexibler und einfacher anzuwenden.

Mit der Funktion `is.matrix()` kann überprüft werden, ob eine Datenstruktur eine Matrix ist. Erfüllt eine tabellarische Struktur die Kriterien für eine Matrix, dann kann diese Struktur mit der Funktion `as.matrix()` in eine Matrix umgewandelt werden.

Die R-Matrix ähnelt der Struktur von Vektoren.

- Eine Matrix hat eine Länge. Diese Länge entspricht der Gesamtzahl der Elemente der Matrix.
- Wird eine Matrix in der Funktion `c()` als Wert übergeben, dann wird die Matrix zuerst in einen Vektor umgewandelt.

Weil die Anzahl der Spalten und Zeilen nicht über die Länge bestimmt werden kann, muss zu diesem Zweck die Funktion `dim()` verwendet werden. `dim()` gibt die Anzahl der Zeilen und der Spalten in dieser Reihenfolge aus.

Die Werte in einer R-Matrix können über den Zeilen- und Spaltenindex abgefragt werden. Dazu werden wie bei den Vektoren einfache eckige Klammern verwendet. Der Zeilen- und Spaltenindex werden dabei durch ein Komma voneinander getrennt.

Beispiel 6.22 (Einen Wert über den Matrix-Index zugreifen)**.**

```r
c(1, 2, 3) %o% c(3, 4, 5) -> matrix123
matrix123[2,2] # ergibt 8
```

Wird beim Matrix-Index der Zeilen- oder der Spaltenindex weggelassen, wird eine ganze Zeile bzw. Spalte als Vektor ausgewählt.

> **Praxis**
>
> Ähnlich wie bei Vektoren, ist es in der Praxis nur sehr selten notwendig, auf die Werte einer Matrix zuzugreifen.

6.2.4. Data-Frames

Ein Data-Frame ist eine *benannte geschachtelte Liste* mit *Vektoren gleicher Länge*. Diese Datenstruktur ist die Basis für Datentabellen. Anders als eine normale geschachtelte Liste stellt ein Date-Frame zusätzlich sicher, dass alle Vektoren *immer* die gleiche Länge haben.

Data-Frames werden normalerweise mit den Funktionen `tibble()` oder `tribble()` erstellt. Diese Funktionen sind nur in **tidy R** verfügbar. Diese Funktionen erzeugen eine effizienteren und damit schnelle Version eines Data-Frames als Base R.

> **Praxis**
>
> Data-Frames werden normalerweise beim Einlesen der Daten mit der korrekten Datenstruktur automatisch erzeugt. Händisches Erstellen von Date-Frames ist dann nicht mehr notwendig.

Weil Data-Frames spezielle Listen sind, können sie genau gleich wie Listen behandelt werden. Es lassen sich damit die gleichen Zugriffe, wie bei Listen umsetzen.

> **Hinweis**
>
> Vektoren in Data-Frames können Listen als Datentyp haben, was bei normalen Vektore nicht möglich ist. Diese besondere Eigenschaft wird im Kapitel 16 ausgenutzt.

> **Praxis**
>
> Die Behandlung von Data-Frames als geschachtelte Listen ist inzwischen unüblich. Stattdessen sollten die effizienteren und leichter zu merkenden Transformationsschritte eingesetzt werden.

7. Importieren und Exportieren

7.1. Daten importieren

Das Einlesen von Datendateien ist ein zentraler Bestandteil von R, weil es die Voraussetzung für die statistische Programmierung bildet. Diese Funktionen gehen jedoch nicht sehr sparsam mit dem Arbeitsspeicher unseres Computers um, sodass sehr grosse Datenmengen immer wieder zu Problemen führen.

Die `readr`-Bibliothek ersetzt die Base R-Funktionen zum Einlesen von Dateien durch flexiblere und effizientere Funktionen. Diese Funktionen können mit grösseren Datenmengen umgehen und schonen den verfügbaren Arbeitsspeicher. Deshalb sind die `readr`-Funktionen den jeweiligen Gegenstücken von Base R vorzuziehen.

7.1.1. Dateitypen

Für den Austausch von Stichproben stehen verschiedene Dateiformate zur Verfügung. Diese Dateiformate unterscheiden sich durch die Strategie, mit der die Werte in den einzelnen Tabellenzellen unterschieden werden.

Die wichtigsten Formate sind:

- Tabulator getrennte Werte (TSV, tabulator-separated values)
- Komma getrennte Werte (CSV, comma-separated values)
- Excel Tabellen (via `readxl`-Bibliothek)
- Fixformat Tabellen (FWF, fixed-width format)
- R-Datendateien (RDS, R-data structure)

Diese Dateien können wir mit den folgenden Funktionen einlesen.

Format	tidy R	Base R
txt (ganze Datei)	`read_file()`	`readChar()` + `file.info()`
txt (zeilenweise)	`read_lines()`	`readLines()` + `file()`
csv (mit `,` als Trennzeichen)	`read_csv()`	`read.csv()`
csv (mit `;` als Trennzeichen)	`read_csv2()`	`read.csv2()`
tsv	`read_tsv()`	`read.table()`
xls (Excel Arbeitsmappen mit `readxl`)	`read_excel()`	-
FWF	`read_fwf()`	-
RDS	`read_rds()`	`readRDS()`

Die Base R Funktionen `read.table()`, `read.csv` und `read.csv2()` importieren Zeichenketten als Faktoren (s. Kapitel 6). Damit können diese Werte nicht direkt als Zeichenketten behandelt werden. Diese automatische Behandlung entfällt bei den jeweiligen tidy R Varianten. Dadurch lassen sich Daten intuitiver bearbeiten.

> **Achtung**
>
> In der Schweiz kann das CSV-Format zu Verwirrung führen, weil sehr häufig das Semikolon als Spaltentrennzeichen und der Punkt als Dezimaltrennzeichen verwendet werden. Die Ursache für diese Situation sind CSV-Dateien, die aus Excel exportiert wurden.
> Die normalerweise für dieses Format empfohlene Funktion `read_csv2()` behandelt Dezimalzahlen fälschlich als Ganzzahlen. Um dieses Problem zu beheben, sollte das Dezimaltrennzeichen laut Dokumentation wie folgt angepasst werden:
>
> ```
> read_csv2(datei_name, locale = locale(decimal_mark = "."))
> ```

Bei der modernen `read_` Variante können wir uns leicht an der Dateiendung orientieren, um die richtige `read_`-Funktion auszuwählen.

Wenn eine Datei eingelesen wird, dann gibt die jeweilige `read_`-Funktion neben den Daten auch zurück, wie die Datei eingelesen wurde. Enthält die eingelesene Datei die erwarteten Spaltenüberschriften, dann wurde das richtige Dateiformat ausgewählt.

7.1.2. Dateien mit einer Spalte

CSV-Dateien können mit Komma oder Semikolon als Trennzeichen erstellt werden. Falls eine Datei mit nur **einem** *Datenvektor* importiert werden muss, dann kann R das Spaltentrennzeichen nicht finden. In solchen Fälle **muss** die Datei mit der `read_csv()` oder `read_csv2()`-Funktion noch einmal eingelesen werden.

Für Spalten mit Zeichenketten oder Ganzzahlen wird immer die Funktion `read_csv()` verwendet.

Für Gleitkommazahlen erfolgt die Auswahl auf Grundlage des verwendeten Dezimaltrennzeichens. Wird der Dezimalpunkt verwendet, dann **muss** die Funktion `read_csv()` benutzt werden. Wird das Dezimalkomma verwendet, dann **muss** die Funktion `read_csv2()` eingesetzt werden.

Beispiel 7.1 (Datei mit einer Spalte importieren)**.** Mit dem Aufruf `read_csv("beispieldaten.csv")` werden Daten mit einem Komma als Trennzeichen und mit Dezimalpunkt eingelesen.

Mit dem Aufruf `read_csv2("beispieldaten.csv")` werden Daten mit einem Semikolon als Trennzeichen und mit Dezimalkomma eingelesen.

In beiden Fällen nutzen wir dieses Verhalten aus, um eine Stichprobe mit nur einer Spalte einzulesen.

7.1.3. Excel Arbeitsmappen

> **Praxis**
>
> Liegen Daten in einer Excel Arbeitsmappe vor, dann muss diese Arbeitsmappe **nicht** in ein anderes Dateiformat umgewandelt werden, damit die Daten in R importiert werden können.

In Excel werden Daten in *Arbeitsmappen* organisiert. Es ist also möglich, mehr als eine Tabelle und darauf basierende Operationen in einer Datei zu speichern. Damit Daten aus Arbeitsmappen in R importiert werden können, müssen die Struktur der Arbeitsmappe bekannt sein.

Eine Excel Arbeitsmappe ist eine Datei, die üblicherweise auf `.xlsx` endet. Die Dateiendung signalisiert uns *meistens* die interne Organisation einer Datei. *Interne Organisation einer Datei* bedeutet, in welcher Folge die Daten in einer Datei auf der Festplatte abgelegt sind.

Nur das Dateiformat von `.xlsx`-Dateien unterstützt alle Funktionen von Excel und kann von R korrekt eingelesen werden

Das Dateiformat wird in Excel im `Speichern-Unter`-Dialog festgelegt. Dieser Dialog erscheint in der Regel, wenn eine neue Arbeitsmappe das erste Mal gespeichert wird. Wenn im Start-Dialog von Excel einfach eine neue Arbeitsmappe erstellt wird, dann erzeugt Excel *automatisch* eine Arbeitsmappe im Excel-Format.

> **Merke**
>
> Excel-Dateien sind Dateien mit der Endung `.xlsx` oder `.xls`und werden als **Excel Arbeitsmappen** bezeichnet. Nur Dateien mit dieser Endung können in R als Excel-Datei importiert werden.

Excel Arbeitsmappen haben vier zentrale Strukturelemente:

1. Arbeitsblätter
2. Adressbereiche
3. Zellenwerte
4. Zellenformeln

Jedes Arbeitsblatt einer Arbeitsmappe hat einen *eindeutigen* Namen.

Die Adressbereiche sind in Zeilen und Spalten gegliedert. Wir finden Daten daher immer auf einem bestimmten Arbeitsblatt in einem bestimmten Adressbereich. Die konkrete Position der Daten in der Arbeitsmappe legen die Autoren willkürlich fest.

Jede Zelle eines Arbeitsblatts hat *immer* zwei *gleichzeitige* Zustände, die immer in einer Excel Arbeitsmappe gespeichert werden:

1. Jede Zelle hat einen Wert.
2. Jede Zelle hat eine Operation.

Aus diesen Strukturelementen ergeben sich zwei Konsequenzen:

1. Ein Arbeitsblatt kann mehr als eine Tabelle mit Daten enthalten.
2. Die Daten müssen nicht am Anfang (d.h. in der ersten Zeile und ersten Spalte) eines Arbeitsblatts beginnen.

Um mit den Daten in Excel Arbeitsmappen arbeiten zu können, müssen bekannt sein, auf welchem Arbeitsblatt und in welchem Adressbereich die Daten stehen.

> **Merke**
>
> Tabellen sind **keine** Strukturelemente von Excel Arbeitsmappen, die in R zugänglich sind.

> **Achtung**
>
> Wenn Excel Arbeitsmappen mit Excel geöffnet werden, dann berechnet Excel alle Operationen auf *allen* Arbeitsblättern neu. Damit werden die Werte in der Arbeitsmappe verändert.
> Es kommt also vor, dass sich eine Arbeitsmappe ändert, ohne dass eine Interaktion vorgenommen wurde. In diesen Fällen fragt Excel beim Schliessen der Arbeitsmappe, ob die Änderungen gespeichert werden sollen.
> Wird eine Excel Arbeitsmappe in R (oder in einer anderen Programmiersprache) geöffnet, dann wird nur die Arbeitsmappe geöffnet ***ohne*** die Operationen neu zu berechnen.

Mit den Funktionen der `readxl`-Bibliothek können wir Excel Arbeitsmappen nach R importieren. Dabei sind zwei Funktionen von besonderer Bedeutung:

- `excel_sheets(dateiname)` und
- `read_excel(dateiname, sheet)`

Mit der Funktion `excel_sheets()` können die vorhandenen Arbeitsblätter erkannt werden. Das Ergebnis dieser Funktion ist die Liste der Arbeitsblattnamen in einer Arbeitsmappe. Diese Funktion sollte vor dem Import von Daten zur Kontrolle der Arbeitsblattnamen verwendet werden.

Die Funktion `read_excel()` erlaubt es einzelne Arbeitsblätter zu importieren. Wenn kein Arbeitsblattname für den Parameter `sheet` übergeben wird, dann nimmt die Funktion das aktive oder das erste Arbeitsblatt in der Arbeitsmappe.

Mit den `readxl`-Funktionen können keine Formeln aus den Zellen ausgelesen werden.

Beispiel 7.2 (Excel-Arbeitsmappe importieren)**.**

```
library(readxl)

Arbeitsblaetter = excel_sheets("Bestellungen_2.xlsx")
# Das Arbeitsblatt "Daten" sollte vorhanden sein.

Daten = read_excel("Bestellungen_2.xlsx", "Daten")
```

Die Funktion `read_excel()` importiert alle Daten auf einem Arbeitsblatt. Enthält nur ein bestimmter Bereich auf einem Arbeitsblatt die Daten von Interesse, dann muss dieser Bereich als Excel-Bereichsadresse angegeben werden.

> **Warnung**
>
> `read_excel()` kann nur mit Excels Arbeitsblattadressen umgehen. Tabellenadressen oder die Gatter-Notation beherrscht die Funktion nicht.

7.2. Daten exportieren

R unterstützt den Export strukturierter Daten in Textdateien. Beim Exportieren kommen für die Formate TSV und CSV werden die entsprechenden Funktionen `write_tsv()`, `write_csv()` und `write_csv2()` benutzt. Für speziellformatierte Dateien kann die Funktion `write_delim()` eingesetzt werden.

Alle Export-Funktionen erwarten eine Datenstruktur als ersten Parameter und einen Dateinamen als zweiten Parameter. Der Dateiname legt fest, wohin das Ergebnis der Funktion auf dem Computer geschrieben werden soll.

Die Import- und Export-Funktionen lassen sich zu einfachen Konvertierungsprogrammen verknüpfen. Beispiel 7.3 korrigert von Excel exportierte CSV-Dateien in ein gültiges CSV-Format.

Beispiel 7.3 ("Schweizer" CSV-Format korrigieren)**.**

```
library(readr)

write_csv(
    read_csv2("Bestellungen_2.csv"),
    "Bestellungen_korrigiert.csv"
)
```

oder eleganter mit Funktionsverkettung:

```
read_csv2("Bestellungen_2.csv") |>  write_csv("Bestellungen_korrigiert.csv" )
```

> **Warnung**
>
> R kann Excel Arbeitsmappen **nicht exportieren**. Die `readr`-Funktionen `write_excel_csv()` und `write_excel_csv2()` exportieren CSV-Dateien mit einer zusätzlichen Markierung am Dateianfang. **Diese Funktionen sollten nur verwendet werden, wenn eine CSV-Datei nur mit Excel importiert werden soll und nicht für die Archivierung oder Weiterverarbeitung gedacht ist.**
> Die zusätzliche Markierung wird als **Byte Order Mark** (BOM) bezeichnet und muss das UTF8-Symbol `FEFF` sein. Dieses Symbol ist ein Leerzeichen ohne Länge und wird deshalb nie dargestellt. Excel bzw. Power Query verwenden das BOM, um UTF8-kodierte Dateien zu identifizieren.

7.3. JSON-Daten

JSON ist ein Datenformat, dass von vielen sog. *Web-Diensten* zum Austausch von Datenstrukturen eingesetzt wird. R kann dieses Datenformat mit der `tidyverse`-Bibliothek `jsonlite` importieren und auch exportieren. `jsonlite` stellt zwei Funktionen für den regelmässigen Einsatz bereit:

- `fromJSON()`
- `toJSON()`

Die beiden Funktionen `fromJSON()` und `toJSON()` unterstützen das Parsen von und Serialisieren zu Zeichenketten im JSON-Format.

Um Daten aus einer Textdatei im JSON-Format zu importieren, kann die Funktion `read_json()` verwendet werden.

Beispiel 7.4 (JSON Daten aus einer Datei importieren)**.**

```
library(jsonlite)

Daten = read_json("beispiele/daten.json", simplifyVektor = TRUE)
```

Mit der Funktion `toJSON()` werden Daten in eine JSON-formatierte Zeichenkette umgewandelt. Diese Zeichenkette kann anschliessend mit `write_file()` in eine Datei geschrieben werden (Beispiel 7.5). Dieser Doppelschritt kann mit der Funktion `write_json()` zusammengefasst werden (Beispiel 7.6).

Beispiel 7.5 (Daten im JSON-Format exportieren)**.**

```
write_file(toJSON(Daten),"neue_daten.json"))
```

Beispiel 7.6 (Daten im JSON-Format exportieren und in eine Datei schreiben)**.**

```
write_json(Daten,"neue_daten.json")
```

> **Hinweis**
>
> Die beiden Funktionen `read_json()` und `write_json()` erlauben das Lesen und Schreiben von Textdateien im JSON-Format. Die Standardeinstellungen sind jedoch nicht identisch mit denen von `fromJSON()` und `toJSON()`, so dass der Import und Export mit diesen Funktionen komplexer ist, als mit der oben beschrieben Technik.

7.4. YAML-Daten

YAML ist eine Verallgemeinerung des JSON-Formats. Mit dem Ziel, dass Menschen komplexe Datenstrukturen leichter eingeben und lesen können. In R wird das Format von der Bibliothek `yaml` unterstützt. Diese Bibliothek gehört nicht zum `tidyverse` und muss separat installiert werden.

Die `yaml`-Bibliothek stellt vier Funktionen bereit:

- `yaml.load()` zum *Parsen* einer YAML-formatierte Zeichenkette in einer Datenstruktur.
- `as.yaml()` zum *Serialisieren* einer Datenstruktur in eine YAML-Zeichenkette.
- `read_yaml()` zum Importieren von YAML-Daten aus einer Datei.
- `write_yaml()` zum Schreiben einer Datenstruktur in eine YAML-formatierte Datei.

Der YAML-Parser `yaml.load()` erzeugt immer eine geschachtelte Datenstruktur aus benannten Listen, unbenannten Listen und Vektoren erzeugt wird. Der Parser erkennt automatisch, ob eine YAML-Liste ein Vektor oder eine Liste ist. YAML-Objekte werden immer in benannte Listen umgewandelt.

Beispiel 7.7 (YAML-Daten mit `read_yaml()` importieren)**.**

```
library(yaml)

yamlDaten = read_yaml("daten.yml")
```

Beispiel 7.8 (YAML-Daten mit `read_yaml()` exportieren)**.**

```
yamlDaten |> write_yaml("kopie_der_daten.yml")
```

7.5. Festkodierte Daten

R unterstützt den Import von **festkodierten Daten** nicht direkt. Festkodierte Daten benötigen einen eigenen Parser, der die Datenfelder extrahiert. Die prinzipielle Vorgehensweise ähnelt dem Import und Export von JSON-Daten. Dazu werden die Daten als unstrukturierte Textdaten mit der Funktion `read_file()` eingelesen. Anschliessend werden die Datenfelder mit Zeichenketten-Operationen (Kapitel 9) einzeln extrahiert. Beim Exportieren müssen die Daten zuerst serialisiert werden und anschliessend mit der Funktion `write_file()` in die entsprechende Datei geschrieben werden.

Teil II.

Mathematik der Daten

8. Variablen, Funktionen und Operatoren

8.1. Variablen

Variablen sind spezielle R Symbole (s. Kapitel 4) mit denen Werte für die spätere Verwendung markiert werden. Variablen sind also **Bezeichner**, welche die eigentlichen Werte **substituieren**.

Damit eine Variable einen Wert substituieren kann, muss der Wert der Variablen *zugewiesen* werden. Ein Wert kann dabei ein einzelner Wert eines fundamentalen Datentyps oder eine komplexe Datenstruktur sein.

Bei der ersten Zuweisung wird eine Variable *deklariert* (Definition 4.3).

Beispiel 8.1 (Den Wert 1 der Variable `var1` zuweisen)**.**

```
var1 = 1
```

Variablen müssen in einem Geltungsbereich *eindeutig* sein. Wird nämlich einer Variable mehrfach zugewiesen, dann ist der Wert einer Variablen der Wert der letzten Zuweisung.

Der **Geltungsbereich** (engl. Scope) einer Variablen wird durch Funktionskörper definiert. R kennt dabei drei Arten von Geltungsbereichen. In diesem Zusammenhang spricht man von äusseren (engl. *outer scope*) und inneren Geltungsbereichen (engl. *inner scope*).

Grundsätzlich können alle Variablen in einem Geltungsbereich verwendet werden, die in einem der äusseren Geltungsbereiche deklariert und zugewiesen wurden. Variablen der inneren Geltungsbereiche sind in den äusseren Geltungsbereichen *nicht verfügbar*.

Der **globale Geltungsbereich** gilt für alle Variablen, die ausserhalb einer Funktion oder einer Bibliothek erzeugt werden.

Der **Funktionsgeltungsbereich** ist auf den Funktionsköper einer Funktion beschränkt.

Der **Modulgeltungsbereich** ist der globale Geltungsbereich einer Funktionsbibliothek. Variablen dieses Geltungsbereichs sind im globalen Geltungsbereich eines R-Scripts nicht erreichbar. In der Praxis spielt dieser Geltungsbereich eine untergeordnete Rolle

> **Warnung**
>
> Die letzte Zuweisung ist nicht zwingend die Zuweisung, die als letztes im Code erscheint.

8.2. Funktionen

In R bilden Funktionen die Grundlage für die Datenverarbeitung. Fast alle Spracheelemente sind als Funktionen umgesetzt.

8.2.1. Identätsfunktion

R hat eine explizite Identitätsfunktion `identity()`. Diese Funktion setzt die Identität für einen Parameter um. Die Funktion wird zur allgemeinen Identitätsfunktion, indem die Parameterliste als Liste übergeben.

Beispiel 8.2 (Mehrparametrische Indentität)**.**

```
identity(list(1, "Daten und Information", TRUE))
```

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

[[2]]
[1] "Daten und Information"

[[3]]
[1] TRUE
```

8.2.2. Transformationen

Transformationen verändern die Werte eines Vektors, wobei das Ergebnis *immer* ein Ergebnis der Länge des ursprünglichen Vektors hat.

In R sind alle arithmetischen Operatoren Transformatoren (s. Beispiel 8.3)

Beispiel 8.3 (Additive Transformation eines Zahlenvektors (Skalarprodukt))**.**

```
c(1, 9 , 7, 3, 5) * 2
```

```
[1]  2 18 14  6 10
```

Eine besondere Gruppe innerhalb der Transformationen sind die Matrizen-Produkte (Kapitel 13) sowie das Umformen von Vektoren (Kapitel 16). Diese Transformationen erzeugen aus mehreren Vektoren *Vektorfelder*, wobei die ursprünglichen Werte erhalten bleiben bzw. durch eine Multiplikation umgeformt werden.

8.2.3. Aggregatoren

Aggregationen fassen Werte eines Vektors zusammen. Das Ergebnis von Aggregatoren ist oft ein einzelner Wert, kann aber auch mehrere Werte umfassen. In solchen Fällen hat das Aggregat höchstens die Länge des Eingabevektors.

Typische Aggregationen sind die Summe (`sum()`, Beispiel 8.4) und andere statistische Kennzahlen (Kapitel 17).

Beispiel 8.4 (Summenaggregation)**.**

```
c(1, 2, 3, 4) |> sum()
```

```
[1] 10
```

8.2.4. Transformationsverben

Die `tidyverse`-Bibliothek kennt sog. **Transformationsverben**. Das sind spezielle Funktionen für die Arbeit mit Datenrahmen. Diese Funktionen vereinheitlichen den Umgang mit Daten, indem sie die Komplexität der notwendigen Arbeitsschritte verbergen. Gleichzeitig machen sie die Intention einer Datentransformation sichtbar.

Tabelle 8.1.: Die wichtigsten Transformationsverben der `tidyverse` Bibliothek{#tbl-transformationsverben}

Funktion	Beschreibung
`mutate()`	Führt eine oder mehrere Transformationen über einen Datenrahmen durch.
`arrange()`	Ordnet die Datensätze eines Datenrahmens neu an (Kapitel 11).
`summarise()`	Führt eine oder mehrere Aggregationen über einen Datenrahmen durch.
`count()`	Zählt die Anzahl der Datensätze, optional auch entlang gemeinsamer Eigenschaften (Kapitel 12).
`rename()`	Benennt Vektoren um.
`select()`	Wählt Vektoren in einem Datenrahmen aus.
`filter()`	Aggregiert einen Datenrahmen mithilfe von logischen Ausdrücken (Kapitel 11)
`group_by()/ungroup()`	Gruppiert einen Datenrahmen mithilfe gemeinsamer Eigenschaften (Kapitel 14)
`pivot_longer()/ pivot_wider()`	Überführt einen Datenrahmen von der Breitform in die Langform, bzw. umgekehrt (Kapitel 16).
`nest()/unnest()`	Bettet zusammengehörende Teile eines Datenrahmens als Unterstruktur in einen Datenrahmen ein, bzw. bettet diese Teile aus (Kapitel 16).

Die beiden wichtigsten Funktionen sind `mutate()` für Transformationen und `summarise()` für Aggregationen. Das Ergebnis dieser Funktionen ist immer ein neuer Datenrahmen, welcher die transformierten Daten enthält.

8.2.5. Generatoren

Generatoren bilden eine besondere Funktionsgruppe in R. Mit ihnen lassen sich Werte erzeugen. In dieser Gruppe gibt es zwei Arten:

1. Sequenzgeneratoren
2. Zufallsgeneratoren

8.2.5.1. Sequenzgeneratoren

Sequenzgeneratoren erzeugen Sequenzen von Werten. Sequenzgeneratoren werden in R zur systematischen Erzeugung von Vektoren verwendet.

Die beiden wichtigsten Funktionen dieser Gruppe sind die Funktionen `seq()` und `rep()`.

Die Funktion `seq()` erzeugt eine Sequenz von Werten mit fester Schrittweite. In R wird eine Sequenz über einen Startwert sowie die Länge, die Schrittweite *oder* den Endwert definiert.

Beispiel 8.5 (Sequenzen mit `seq()` erzeugen)**.**

```
seq(5)
```

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

```
seq(2, 5)
```

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

```
seq(2, len = 5)
```

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

```
seq(2, len = 3, by = 2)
```

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

Sequenzen mit der Schrittweite `0` wiederholen den Startwert für die erforderliche Länge. Solche Sequenzen werden relativ häufig verwendet, weshalb Beispiel 8.6 etwas umständlich ist.

Beispiel 8.6 (Einsvektor der Länge 4 mit `seq()` erzeugen)**.**

```
seq(1, len = 4, by = 0)
```

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

Diese Operation kann mit R vereinfacht werden. Hierfür diehnt die Funktion `rep()` (für *replicate*). Die Funktion `rep()` erfordert den Wiederholungswert sowie die Anzahl der Wiederholungen (Beispiel 8.7).

Beispiel 8.7 (Einsvektor der Länge 4 mit `rep()` erzeugen)**.**

```
rep(1, 4)
```

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

8.2.5.2. Zufallsgeneratoren

Definition 8.1. Zufallszahlen sind numerische Werte, die zufällig ausgewählt werden.

Zufallszahlen werden in normalen Computer-Programmen relativ selten verwendet.

> **i** Merke
>
> Die ***Anwendungsgebiete*** von Zufallszahlen sind die **Krypografie** und die **Simulation**.

In den Datenwissenschaften sind Simulationen ein wichtiges Werkzeug für analytische und prediktive Modelle, insbesondere für die künstliche Intelligenz.

R bietet die Funktion `runif()` zur Erzeugung von gleichmässig verteilten Zufallszahlen. Die Funktion generiert zufällig reelle Werte im Intervall von $]0, 1[$. In diesem Zusammenhang bedeutet *gleichmässig verteilt*, dass alle Werte mit gleicher Wahrscheinlichkeit erzeugt werden.

> **i** Merke
>
> *Gleichmässig verteilte* Werte werden auch als **uniforme** oder **uniform-verteilte** Werte bezeichnet.

Beispiel 8.8 (Fünf Zufallswerte mit `runif()` erzeugen)**.**

```r
runif(5)
```

```
[1] 0.7999100 0.0587859 0.6590794 0.9931456 0.8063347
```

Aus diesen Werten lassen sich beliebige Zufallsvektoren erzeugen. Um Werte in anderen Intervallen zu erzeugen, können der Funktion eigene Intervallgrenzen übergeben werden.

Werden uniformverteilte Zufallswerte gegenübergestellt, dann sind diese Werte gleichmässig im Intervall verteilt (Abbildung 8.1).

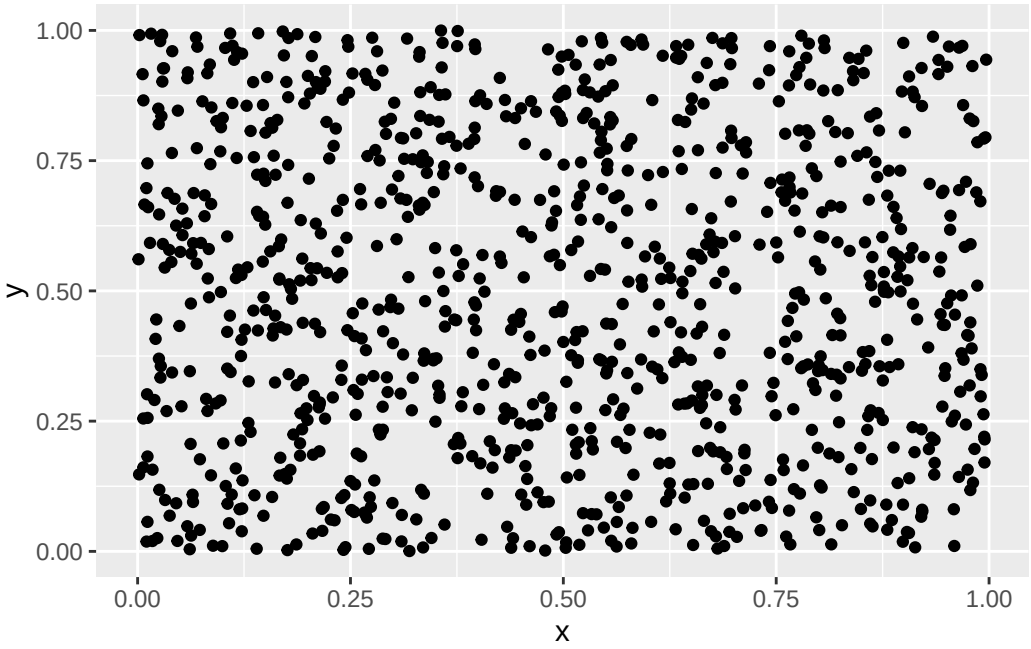


Abbildung 8.1.: Visualisierung von zwei Vektoren mit Zufallswerten

> **Merke**
>
> Ganzzahlen dürfen nicht durch Runden , sondern müssen durch Entfernen des Nachkommaanteils mit `trunc()` erzeugt werden.

Beispiel 8.9 (Zufällige Ganzzahlen in einem Intervall erzeugen)**.**

```r
runif(10, min = -10, max = 10) |> trunc()
```

```
 [1] -8  0  7 -5  7 -8  0 -5 -3 -5
```

Neben den uniformverteilten Werten, stellt R Funktionen zum Erzeugen von Zufallswerten mit anderen Verteilungen an. Die Grundlage für diese Zufallswerte bilden die *statistischen Verteilungen*, wie z.B. die Normalverteilung (mit `rnorm()`), die F-Verteilung (mit `rf()`), die Binomialverteilung (mit `rbinom()`) oder die χ^2-Verteilung (mit `rchisq()`).

In diesen Verteilungen sind nicht alle Werte gleich wahrscheinlich. Dadurch erscheinen in einer Visualisierung die Werte geklumpt (Abbildung 8.2).

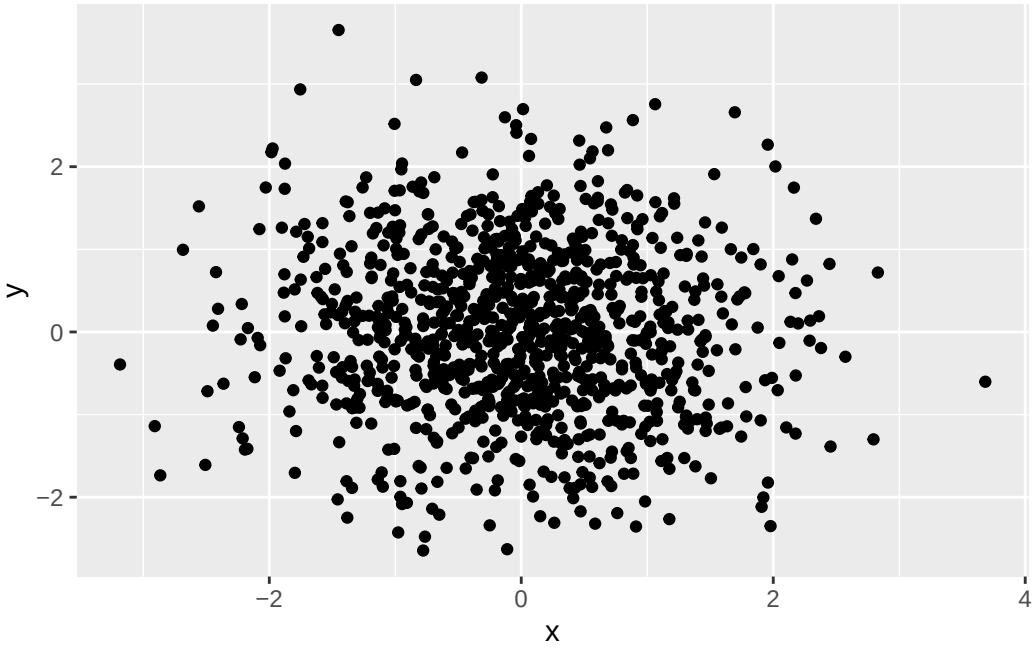


Abbildung 8.2.: Visualisierung von zwei Vektoren mit normalverteilten Zufallswerten

8.3. Operatoren

Alle R-Operatoren sind Funktionen. R kennt 29 vordefinierte Operatoren, die zwei Werte verknüpfen. Zu diesen Operatoren gehören die auch die arithmetischen Operatoren für die Grundrechenarten.

Tabelle 8.2.: Liste der Base R Operatoren

Operator	Beschreibung	Art
`+`	Plus, sowohl unär als auch binär	arithmetisch
`-`	Minus, sowohl unär als auch binär	arithmetisch
`*`	Multiplikation, binär	arithmetisch
`/`	Division, binär	arithmetisch
`^`	Potenz, binär	arithmetisch
`%%`	Modulo, binär	arithmetisch

Operator	Beschreibung	Art
`%/%`	Ganzzahldivision, binär	arithmetisch
`%*%`	Matrixprodukt, binär	arithmetisch, Matrix
`%o%`	äusseres Produkt, binär	arithmetisch, Matrix
`%x%`	Kronecker-Produkt, binär	arithmetisch, Matrix
`<`	Kleiner als, binär	logisch
`>`	Grösser als, binär	logisch
`==`	Gleich, binär	logisch
`!=`	Ungleich, binär	logisch
`>=`	Grösser oder gleich, binär	logisch
`<=`	Kleiner oder gleich, binär	logisch
`%in%`	Existenzoperator, binär	logisch
`!`	unäres Nicht	logisch
`&`	Und, binär, vektorisiert	logisch
`&&`	Und, binär, nicht vektorisiert	logisch
`|`	Oder, binär, vektorisiert	logisch
`||`	Oder, binär, nicht vektorisiert	logisch
`<-`, `<<-`, `=`	linksgerichtete Zuweisung, binär	Zuweisung
`->`, `->>`	rechtsgerichtete Zuweisung, binär	Zuweisung
`[`	Indexzugriff (Vektoren), binär	Index
`$`, `[[`	Listenzugriff, binär	Index
`~`	funktionale Abhängigkeit, sowohl unär als auch binär	Funktionen
`:`	Sequenz (in Modellen: Interaktion), binär	Funktionen
`|>`	Funktionsverkettung	
`?`	Hilfe	spezial

Hinweis

Im R-Umfeld wird oft von **Modellen** geschrieben und gesprochen. *Modelle* sind *spezielle Funktionen*, die *Beziehungen zwischen Daten* beschreiben, ohne eine mathematisch exakte Beziehung vorzugeben. Modelle werden in der *Statistik* und *Stochastik* eingesetzt, wenn die exakten Beziehungen zwischen Daten unbekannt sind.

Beispiel 8.10 (Exakte *lineare* Beziehung zwischen Daten)**.**

```
f = function (x, c) 2 * x + 3 * c
```

Beispiel 8.11 (Beziehung zwischen Daten mit Interaktion als Modell)**.**

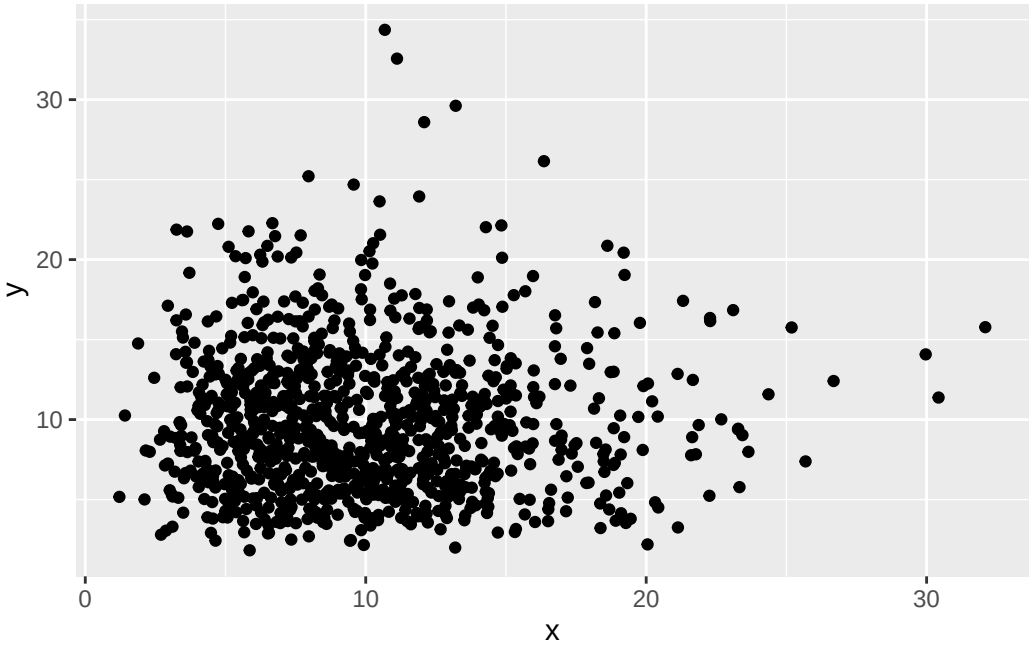


Abbildung 8.3.: Visualisierung von zwei Vektoren mit χ^2-verteilten Zufallswerten

```
f = y ~ x : c
```

Hinter jedem Operator steht eine Funktion, die mit den beiden Operanden als Parameter ausgeführt wird, um das Ergebnis des Operators zu bestimmen. Daraus folgt, dass jeder Operator auch als Funktionsbezeichner verwendet werden kann. In diesem Fall muss R mitgeteilt werden, dass der Operator nun als Funktionsbezeichner verwendet werden soll. Der Operator muss also mit Backticks als Bezeichner markiert werden.

Beispiel 8.12 (+-Operator als Funktionsbezeichner)**.**

```
`+`(1, 2)
```

```
[1] 3
```

8.3.1. Zuweisung

R kennt zwei Zuweisungsoperatoren: `<-` und `->`. Die Zuweisung erfolgt in Richtung des Pfeils. Daneben wird der `=`-Operator ebenfalls als (inoffizieller) Zuweisungsoperator unterstützt.

Ein Zuweisungsoperator erwartet immer einen Bezeichner und eine Operation als Parameter. Das Ergebnis der Operation wird als Wert dem Bezeichner zugewiesen.

Weil nicht immer klar ist, ob `<-` oder `=` verwendet werden soll, lautet die offizielle Kommunikation, dass für Variablenzuweisungen der `<-`-Operator verwendet werden sollte. Das einfache Gleich (`=`) weist einen Wert einem Funktionsparameter zu. Gerade in **tidy R** ist dieser Unterschied nur schwer nachvollziehbar, weil bestimmte Parameter wie Variablen behandelt werden.

> **Hinweis**
>
> In diesem Buch wird für die *linksgerichtete Zuweisung* immer das Gleichzeichen (`=`) verwendet, so dass eine Zuweisung eines Werts an eine Variable und an einen Parameter gleichwertig behandelt wird. Dadurch wird die Lesart etwas vereinfacht. Zusätzlich wird die rechtsgerichtete Zuweisung konsequent als Abschluss für einen primären Datenstrom (s. Kapitel 8.4) eingesetzt.

8.3.2. Funktionsausführung

Der Ausführenoperator (`()`) gilt in R offiziell nicht als Operator, weil dieser nicht als Funktion umgesetzt werden kann. Es gibt zwar die Funktion `do.call()`, um eine Funktion auszuführen. Wenn diese Funktion als Ausführungsoperator eingesetzt wird, müsste `do.call()` sich selbst aufrufen, um sich selbst auszuführen. Dieses Problem wird von R dadurch gelöst, dass `(` und `)` als eigene *Symbole* erkannt werden und immer eine Funktionsausführung anzeigen.

8.3.2.1. Hilfeoperator

Der *Hilfeoperator* ist ein besonderer Operator, weil dieser die Interaktion mit der Dokumentation von Funktionen und Konzepten ermöglicht. Der Hilfeoperator wird normalerweise nicht in einem R-Script verwendet und hat keine Bedeutung für die Datenverarbeitung.

Der Hilfeoperator kann direkt mit einem Bezeichner aufgerufen werden. Existiert für den Bezeichner eine Dokumentation, dann wird diese angezeigt.

Beispiel 8.13 (Dokumentation der Funktion `is.character()`)**.**

```
?is.character
```

Wird der Hilfeoperator mit sich selbst aufgerufen, wird der nächste Wert als Suchbegriff gewertete und eine Suche über alle Hilfedokumente auf dem System durchgeführt.

Beispiel 8.14 (Dokumentationssuche nach Operatoren)**.**

```
??operator
```

8.4. Funktionsketten

R unterstützt die spezielle Funktionsverkettung mit dem `|>`- Operator. Dadurch lassen sich Funktionsfolgen direkt in R ausdrücken. In Kombination mit der rechtsgerichteten Zuweisung (`->`) ist es möglich, Datenströme durch eine Funktionskette von einem Ausgangswert zu einem Ergebnis in der natürlichen Reihenfolge aufzuschreiben.

Beispiel 8.15 (Funktionskette mit abschliessender Zuweisung)**.**

```
# library(tidyverse)
iris |>
    filter(Species == "setosa") |>
    arrange(desc(Petal.Length)) ->
        sortierteSetosaWerte
```

Neben der speziellen Funktionsverkettung (`|>`) gibt es einen sehr ähnlichen Verkettungsoperator: `%>%`. Dieser Verkettungsoperator ist Teil der `tidyverse`-Bibliothek und gleicht der speziellen Funktionsverkettung mit dem kleinen Unterschied, dass die Parameterzuweisung für die nachfolgende Funktion zusätzliche Kontrollmöglichkeiten bietet, die der speziellen Funktionsverkettung fehlen.

8.5. Eigene Funktionen erstellen

In R werden Funktionen mit dem `function`-Schlüsselwort erstellt. Eine R-Funktion besteht aus einer Parameterliste und einem Funktionskörper. Die Parameterliste wird in Klammern hinter dem Wort `function` angegeben. Der Funktionskörper kann eine einzelne Operation oder ein Block sein. Das Ergebnis einer Funktion ist das Ergebnis der letzten Operation des Funktionskörpers.

Beispiel 8.16 zeigt eine *Funktionsdeklaration*, die einen `parameter` *akzeptiert*. Die Funktion quadriert diesen Wert und zieht vom Ergebnis `1` ab. An diesen Operationen wird erkannt, dass die Funktion nur Werte vom Datenyp Zahlen als `parameter` akzeptiert.

Parameter sind in R spezielle *Variablen*, mit denen Werte an eine Funktion übergeben werden. Parameter existieren nur *innerhalb* einer Funktion während der Ausführung des Funktionskörpers. Es kommt sehr häufig vor, dass ausserhalb einer Funktion Variablen mit gleichem Bezeichnern vorhanden sind. Ein Parameter überschreibt diese Variablen **nicht**.

Beispiel 8.16 (Eine Funktion deklarieren)**.**

```
function (parameter) {
    parameter ^ 2 - 1
}
```

Damit eine Funktion sinnvoll verwendet werden kann, muss sie zuerst einer Variablen zugewiesen werden. Der Bezeichner einer Funktion sollte möglichst die zentrale Bedeutung einer Funktion beschreiben.

> **Hinweis**
>
> Die Wahl eines guten Funktionsbezeichners hängt vom jeweiligen Geltungsbereich ab. Mathematische Funktionen werden oft mit $f(x)$ oder $g(x)$ usw. geschrieben. In R sind solche Bezeichner ebenfalls zulässig, solange sie **eindeutig** sind. Solche sehr kurzen Funktionsbezeichnern sollten speziell gekennzeichnet und dokumentiert werden.

> **Praxis**
>
> Weil das Schlüsselwort `function` recht lang ist, behindert es gelegentlich das Lesen sehr einfacher Funktionen. R erlaubt die Definition der Parameterliste mit `\()` anstatt `function ()` zu schreiben. Beide Schreibweisen sind gleichbedeutend.
> Beispiel 8.22 zeigt die Anwendung der Abkürzung mit `\()`.
> Grundsätzlich sollte das `function`-Schlüsselwort der Abkürzung vorgezogen werden, wenn eine Funktion einem Bezeichner zugewiesen wird.

Beispiel 8.17 weist der Funktion aus Beispiel 8.16 den Bezeichner `quadrat_minus_eins` zu. Dieser Bezeichner kann anschliessend als Funktion verwendet werden (s. Beispiel 8.18).

Beispiel 8.17 (Eine Funktion mit Bezeichner deklarieren)**.**

```
quadrat_minus_eins = function (parameter) {
    parameter ^ 2 - 1
}
```

Beispiel 8.18 (Eine selbstdeklarierte Funktion aufrufen)**.**

```
quadrat_minus_eins(2)
```

```
[1] 3
```

8.5.1. Parameter und Variablen

Ein Parameter ist ein Platzhalter für einen Wert, der einer Funktion beim Funktionsaufruf übergeben wird. Parameter werden für eine spezielle Form der Variablenzuweisung eingesetzt.

Im Funktionskörper verhält sich ein Parameter wie eine Variable. Einem Parameter können also in einem Funktionskörper neue Werte zugewiesen werden. Neben Parametern können Funktionskörper zusätzliche Variablen benötigen. Der Geltungsbereich dieser Variablen sind auf den Funktionskörper beschränkt.

8.5.2. Datentypen überprüfen

Wird der neuen Funktion ein falscher Datentyp als Parameter übergeben, dann können die Rs Fehlermeldungen sehr verwirrend sein. Es ist daher ein guter Stil, Parameter die bestimmte Datentypen erfordern direkt zu Begin des Funktionskörpers zu prüfen (s. Beispiel 8.19).

Beispiel 8.19 (Eine Funktion mit Typenprüfung deklarieren)**.**

```
quadrat_minus_eins = function (parameter) {
    stopifnot(is.numeric(parameter))
    parameter ^ 2 - 1
}
```

8.5.3. Nebeneffekte

> ! Wichtig
>
> **Nebeneffekte** sind in (fast) immer unerwünscht. Die in diesem Abschnitt werden die beiden speziellen Zuweisungsoperatoren `<<-` und `->>` vorgestellt, die gezielt **Nebeneffekte** erzeugen.
> Dieser Abschnitt beschreibt einen Sonderfall der Variablen- oder Funktionsdeklaration in **speziellen *Closures*** (s.u.), der in R **sehr selten** vorkommt. Die meisten Algorithmen lassen sich *nebeneffektsfrei* Programmieren, weshalb die beiden speziellen Zuweisungsoperatoren normalerweise nicht verwendet werden.

Der Funktionskörper bildet einen abgegrenzten Geltungsbereich für Variablen. Alle normalen Zuweisungen gelten nur für den Funktionskörper, selbst wenn eine Variable oder ein Parameter ursprünglich in einem äusseren Geltungsbereich deklariert wurde.

Beispiel 8.20 (Geltungsbereich von Variablen in Funktionen)**.**

```
# Deklarationen
var1 = 1
f = function (x) {
    var1 = x + var1
    var1
}

# Anwendung
f(2)
```

```
[1] 3
```

```
var1
```

```
[1] 1
```

In ***seltenen Fällen*** ist es notwendig, eine Variable eines äusseren Geltungsbereichs in einer Funktion einen neuen Wert zuzuweisen. Hier kommen die speziellen Zuweisungen `<<-` und `->>` zum Einsatz. Wird anstelle einer normalen Zuweisung die spezielle Zuweisung verwendet, dann wird einer Variablen oder einem Parameter eines äusseren Geltungsbereich ein neuer Wert zugewiesen.

Definition 8.2. Ändert eine Funktion eine Variable eines äusseren Geltungsbereichs, dann ist diese Änderung ein **Nebeneffekt** der Funktion.

Beispiel 8.21 (Funktion mit Nebeneffekt)**.**

```
# Deklarationen
var1 = 1
f = function (x) {
    x + var1 ->> var1
    var1
}

# Anwendung
f(2)
```

```
[1] 3
```

```
var1
```

```
[1] 3
```

Praxis

In R sollten ausschliesslich *Closures* Nebeneffekte haben, wenn eine Closure eine Variable einer generierenden Funktion ändern muss. ***Dieser Fall tritt sehr selten ein!***

Wichtig

Variablen mit globalem Geltungsbereich sollten **nie** durch Nebeneffekte geändert werden.

> **Hinweis**
>
> **Objektorientierte Sprachen**, wie Python oder Java, verwenden Nebeneffekte als zentrales Programmierprinzip.
> **Streng-funktionale Sprachen**, wie Excel, sind ***nebeneffektfrei***.

8.6. Funktionen als Werte

> **Merke**
>
> Eine Funktion ist für R ein Wert wie eine Zahl oder eine Zeichenkette.

Im Fall von Funktionen ist der Wert einer Funktion die Funktionsdeklaration. Entsprechend ist es möglich Funktionen zu überschreiben.

Wird nur der Bezeichner einer Funktion eingegeben, gibt R die Funktionsdefinition wie jeden anderen Wert direkt aus.

8.6.1. Callbacks

In R werden am häufigsten Funktionen als Parameter an eine andere Funktion übergeben. Im Gegensatz zur Funktionsverkettung ist dabei die Funktion selbst und nicht ihr Ergebnis der Wert des Parameters. Eine Funktion höherer Ordnung implementiert oft den generischen Teil eines Algorithmus und delegiert dem Callback spezifische Aufgaben.

Eine typische Anwendung von Callbacks sind Schleifen. Zwar existieren in R die Schleifenkonzepte `while`, `repeat` und `for`, sie kommen in der Praxis jedoch nie zum Einsatz. Stattdessen kommen fast immer Funktionen höherer Ordnung zum Einsatz. Die Schleife wird durch die Funktion höherer Ordnung realisiert. Der Schleifenblock wird als Callback umgesetzt.

> **Hinweis**
>
> Weil R eine vektorbasierte Programmiersprache ist, werden die meisten Operationen automatisch für alle Elemente eines Vektors ausgeführt. Dadurch sind viele Schleifen unnötig, die in anderen Programmiersprachen erforderlich sind.

Die Funktionen der Bibliothek `purrr` oder deren Schwesterbibliothek `furrr` sind der einfachste Weg in R, um das Verhalten von Schleifen funktional umzusetzten. Dabei sind zwei Funktionen zentral:

- `map()` für Operationen, die für jedes Element unabhängig ausgeführt werden können. Diese Operationen sind *immer* Transformationen. Die Funktion hat immer eine Liste als Ergebnis. Falls ein Vektor benötigt wird, kann dieser durch eine Verkettung mit `unlist()` oder mit der Funktion `map_vec()` erzeugt werden (s. Beispiel 8.22).

- `reduce()` für Operationen, die ein oder mehrere Elemente gemeinsam berücksichtigen. Diese Operationen sind *meistens* Aggregationen. (s. Beispiel 8.23)

Beispiel 8.22 (Lineartransformation mit `map()`).

```
map(rbinom(10, 7, .5), \(x) x - 4) |> unlist()
```

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

```
# Alternativ
map_vec(rbinom(10, 7, .5), \(x) x - 4)
```

```
 [1]  0 -2 -1 -1 -1 -1 -2 -2 -3  3
```

Beispiel 8.23 (Berechnung des n-ten Werts der Fibonacci-Reihe mit `reduce()`).

```
n = 6

fib_add = function (a, b) c(a[2], a[1] + a[2])

reduce(seq(2, len = n-1), fib_add, .init = c(0,1))[2]
```

```
[1] 8
```

8.6.2. Closures

Closures sind Funktionen, die von anderen Funktionen erzeugt und als Ergebnis zurückgegeben werden. Eine Closure bleibt mit der Ausführung der erzeugenden Funktion auch nach Rückgabe verbunden. Dadurch ergeben sich Anwendungen, mit denen sich Funktionsaufrufe vereinfachen lassen.

Beispiel 8.24 (Closure zum systematischen Quadrieren oder Kubieren).

```
potenz_factory = function(e) function (x) x ^ e

quadrieren = potenz_factory(2)
kubieren = potenz_factory(3)

quadrieren(c(2, 3, 4))
```

```
[1]  4  9 16
```

```
kubieren(c(1, 2, 3))
```

```
[1]  1  8 27
```

> **Praxis**
>
> Closures werden in R meistens in Verbindung mit Callbacks verwendet. Die generierende Funktion konfiguriert die Callbacks mit ihren Parametern, so dass diese in den Callbacks verwendet werden können. Ein solches Vorgehen ist immer dann sinnvoll, wenn sehr ähnliche Logik in mehreren Callbacks vorkommt und abstrahiert werden kann.

Beispiel 8.25 (Closure mit Callback verbinden)**.**

```
potenz_reduzierer = function(e) function (p, x) p + x ^ e

reduce(c(1,2,3), potenz_reduzierer(1)) # Summe
```

```
[1] 6
```

```
reduce(c(1,2,3), potenz_reduzierer(2)) # Quadrat-Summe
```

```
[1] 14
```

```
reduce(c(1,2,3), potenz_reduzierer(3)) # Kubik-Summe
```

```
[1] 36
```

8.7. Bibliotheken

Oft ist es nicht notwendig eigene Funktionen zu erstellen. Stattdessen kann in vielen Fällen auf Funktionsbibliotheken zurückgegriffen werden, die bereits entsprechende Funktionen bereitstellen.

R wird durch Funktionsbibliotheken erweitert. Eine Funktionsbibliothek stellt hauptsächlich Funktionen und Operationen für bestimme Algorithmen oder Analysemethoden bereit. Eine Funktionsbibliothek wird mit der Funktion `install.packages()` auf einem Rechner installiert.

In einem R-Script lassen sich die Funktionen einer Bibliothek auf zwei Arten nutzen:

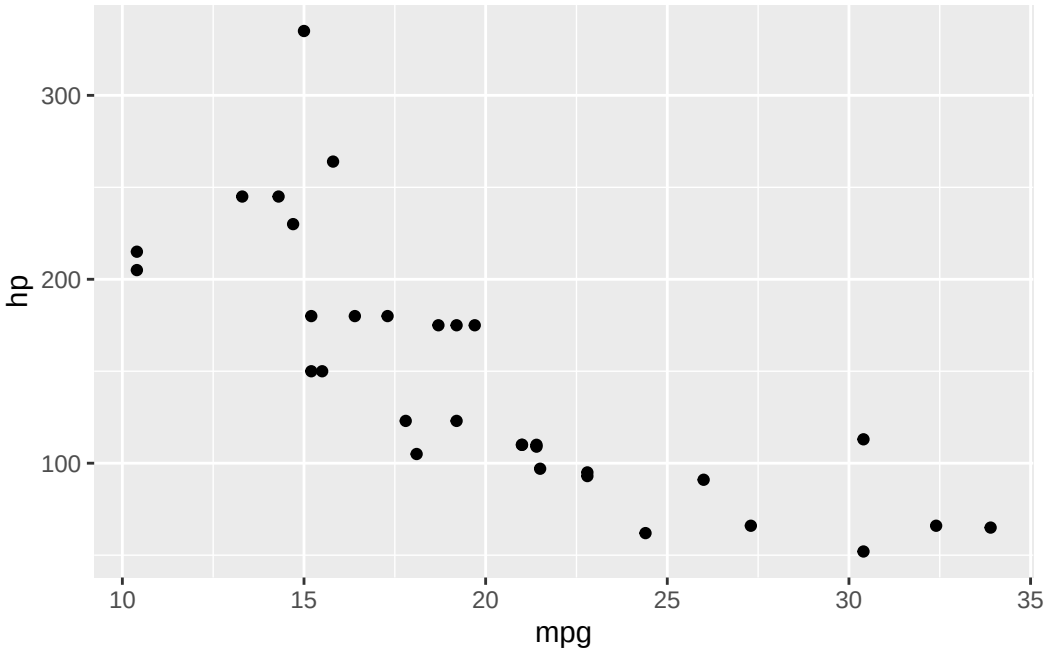
1. Die Bibliothek wird mithilfe der Funktion `library()` in den Code eingebunden.
2. Eine Funktion einer Bibliothek wird direkt angesprochen.

Die erste Option bietet sich an, wenn ein Script viele Funktionen einer Bibliothek aufrufen wird. R läd in diesem Fall *alle Funktionen* der Bibliothek, so dass diese direkt verwendet werden können.

Beispiel 8.26 (Funktionen mit der `library()` Funktion einbinden)**.**

```
library(tidyverse)

mtcars |>
    ggplot(aes(mpg, hp)) +
        geom_point()
```



Die zweite Option ist sinnvoll, wenn nur eine oder zwei Funktionen einer Bibliothek verwendet werden sollen. In diesem Fall muss R nicht die gesamte Bibliothek bereitstellen, sondern läd gezielt nur die gewünschten Funktionen.

Beispiel 8.27 (Eine Funktion direkt ansprechen)**.**

```
mtcars |>
    dplyr::filter(hp > 200)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Duster 360        14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
Cadillac Fleetwood 10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial  14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
Camaro Z28         13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
Ford Pantera L     15.8   8  351 264 4.22 3.170 14.50  0  1    5    4
Maserati Bora      15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
```

> **Hinweis**
>
> R bietet sog. Meta-Bibliotheken an, mit denen mehrere Bibliotheken gemeinsam verwendet werden können. Funktionen können nur nicht über den Namen einer Meta-Bibliothek, sondern immer nur über die Bibliothek, die einer Funktion definiert.

Die `tidyverse`-Bibliothek ist eine solche Meta-Bibliothek. Beispiel 8.28 zeigt wie die Funktion `read_csv()` direkt angesprochen werden kann, wenn die `tidyverse`-Bibliotheken nicht mit `library(tidyverse)` eingebunden wurden. `read_csv()` wird in der Bibliothek `readr` definiert. Entsprechend kann die Funktion nur über `readr::read_csv()` aufgerufen werden.

Beispiel 8.28 (Funktion aus Unterbibliothek direkt ansprechen)**.**

```r
readr::read_csv("meine_daten.csv")
# entspricht:
#   library(tidyverse)
#   read_csv("meine_daten.csv")
```

> **Hinweis**
>
> Die Syntax von R kann durch Module erweitert werden. Diese Form nutzt die Konzepte zur **Metaprogrammierung** von R. Dadurch können neue Programmierkonzepte in die Sprache einfliessen. Die `tidyverse`-Bibliotheken nutzen diese Möglichkeit intensiv. Solche Bibliotheken **müssen** mit der Funktion `library()` eingebunden werden, damit die zusätzliche Syntax bereitgestellt wird.

8.8. Bibliotheken verwalten

Verwendet ein R-Script Funktionsbibliotheken, dann ist dieses Script nur auf Rechnern lauffähig, auf denen die benutzten Bibliotheken auch installiert sind. Solche notewendigen Bibliotheken heissen die **Abhängigkeiten** (engl. *dependencies*) eines Scripts. Weil sich die Abhängikeiten nicht immer leicht erkennen lassen, müssen alle Abhängigkeiten müssen dokumentiert werden.

> **Praxis**
>
> Im Internet gibt es sehr viele Beispiele, die die Funktion `install.packages()` als Teil des Programmcodes darstellen. In konkreten R-Projekten sollte die Funktion `install.packages()` **nie** in einem normalen R-Script aufgerufen werden, weil bei jedem Start des Script geprüft wird, ob eine neue Version der Bibliothek existiert. Diese Technik stellt ein Sicherheitsrisiko dar, weil bei jeder Ausführung des Scripts Installationen unkontrolliert vorgenommen werden können und Schadcode auf die Systeme geschleust werden kann.
> Das Risiko unkontrollierter Installationen wird verringert, indem Installationen von der Programmlogik getrennt und nur kontrolliert durchgeführt werden. Dadurch wird die Installation von Funktionsbibliotheken von ihrer Anwendung getrennt.

Die Dokumentation von Abhängigkeiten wird normalerweise von einem sog. Packetmanagement übernommen. R verfügt über *kein integriertes* Packetmanagement. Dieses wird von der Bibliothek `renv` übernommen. Bevor dieses genutzt werden kann muss `renv` mit `install.packages("renv")` installiert werden.

> **Praxis**
>
> `renv` sollte bei der Installation von R gleich mitinstalliert werden.

`renv` ist ein Packetmanagementsystem für R. Anders als die Funktion `install.packages()` installiert `renv` nicht nur Bibliotheken, sondern dokumentiert auch die Abhängigkeiten eines Projekts in einer Form, dass alle Abhängigkeiten einfach auf dem System installiert werden können. Mit `renv::restore()` lässt sich ein Projekt in einer anderen Umgebung mit allen Abhängigkeiten konfigurieren und ausführen.

Wird eine Bibliothek mit `renv` installiert, dann steht diese Bibliothek nur dem jeweiligen Projekt zur Verfügung. Was auf dem ersten Blick als Nachteil klingt, ist ein grosser Vorteil, wenn unterschiedliche Projekte besonderen Anforderungen an die Versionen einer Bibliothek haben. Auf diese Weise kann jedes Projekt die richtige Version einer Bibliothek verwenden und beeinflusst keine anderen Projekte.

8.8.1. Projektvorbereitung

Ein Projekt wird mit `renv::init()` für die Verwendung des Packetmanagements vorbereitet. Beim ersten Aufruf von `renv` werden die internen Abhängigkeiten von `renv` kontrolliert und notfalls installiert. Das nimmt etwas Zeit in Anspruch.

Das Packetmanagement erfasst automatisch alle Bibliotheken, die systemweit installiert wurden. Dadurch wird sichergestellt, dass alle Bibliotheken berücksichtigt wurden, die im eigenen System installiert sind und deshalb auch im Projekt verwendet werden können. Die Einzige Ausnahme davon ist `renv` selbst.

8.8.2. Bibliotheken installieren

Nach der Initialisierung des Packetmanagements können projektspezifische Bibliotheken mit `renv::install()` installiert werden. War eine Installation erfolgreich, sollte die Bibliothek auf ihre Funktionstüchtigkeit mit einem einfachen Beispiel geprüft und danach mit `renv::snapshot()` als Abhängigkeit dokumentiert werden. Mit einem *Snapshot* wird eine Bibliotheksversion als Abhängigkeit registiert. Im Gegensatz zu `install.packages()` wird ab diesem Zeitpunkt nicht mehr eine beliebige Version der Bibliothek installiert, sondern nur die dokumentierte Version. Dadurch wird sichergestellt, dass der Code auch in anderen Umgebungen wie erwartet funktioniert.

8.8.3. Bibliotheken updaten

Eine Besonderheit von `renv` ist die Möglichkeit, kontrollierte Updates für einzelne oder alle Abhängigkeiten eines Projekts mit `renv::update()` durchzuführen. `renv::update()` installiert die neusten Versionen der Projektbibliotheken.

> **Merke**
>
> Updates sollten nie unklontrolliert akzeptiert werden!

Bevor neue Bibliotheksversionen in das Packetmanagement aufgenommen werden, sollte immer geprüft werden, ob der bestehende Code mit den neuen Versionen immer noch funktioniert. Sollten bei dieser Prüfung Probleme auftreten, dann können die Updates mit `renv::revert()` wieder rückgängig gemacht werden. Gab es keine Probleme, dann können die Updates mit `renv::snapshot()` übernommen werden.

9. Zeichenketten

Bisher haben wir Zeichenketten als atomare Werte behandelt. In diesem Kapitel geht es die Operationen für Zeichenketten.

Eine Zeichenkette hat eine Länge, die der Anzahl der Symbole in der Zeichenkette entspricht und jedes Symbol in einer Zeichenkette kann über dessen Position identifiziert werden.

Wenn Daten als Zeichenketten vorliegen, dann handelt es sich immer um **diskrete Daten**.

Am leichtesten und am einheitlichsten lassen sich Zeichenketten mit der `tidyverse`-Bibliothek `stringr` (Wickham, 2023b) bearbeiten.

Tabelle 9.1.: Die wichtigsten Zeichenketten-Operationen

Name	R
Länge	`str_length()`
Teilketten verbinden	`str_c()`
Position einer Teilzeichenkette finden	`str_locate()`/`str_which()`
Teilkette finden (mit Wahrheitswert als Ergebnis)	`str_detect()`
In Grossbuchstaben umwandeln	`str_to_upper()`
In Kleinbuchstaben umwandeln	`str_to_lower()`
Nur ersten Buchstaben als Grossbuchstabe	`str_to_title()`
Leerzeichen bereinigen	`str_squish()`/`str_trim()`
Nicht-druckbare Zeichen entfernen	`str_replace_all(text, "[\u01-\u07\u0E-\u1f\u80-\u9F]+", "")`
Teilkette extrahieren (linksseitig)	`str_sub(text, 1, x)`
Teilkette extrahieren (rechtsseitig)	`str_sub(text, x, -1)`
Teilkette extrahieren (mittig)	`str_split()`/`str_sub()`
Zeichenkette ersetzen	`str_replace()`/`str_replace_all()`

> **Warnung**
>
> Die Funktion `str()` ist eine Hilfsfunktion, die Information über die interne Struktur einer Variablen bereitstellt. `str()` ist eine Abkürzung für *structure*. Es handelt sich dabei um **keine** Zeichenkettenfunktion.

In R verwenden wir dazu die Funktion `str_replace_all()`. Diese Funktion ersetzt einen Teil einer Zeichenkette durch eine andere Zeichenkette. Wir müssen daher R mitteilen, dass

wir alle Teilzeichenketten löschen möchten, die nicht-druckbare Zeichen enthalten. Das erreichen wir durch den ***regulären Ausdruck*** `"[\u01-\u07\u0E-\u1f\u80-\u9F]+"`. Dieses Suchmuster teilt R mit, welche nicht-druckbaren Zeichen entfernt werden müssen. Das Löschen erreichen wir dadurch, dass wir eine Teilzeichenkette mit der leeren Zeichenkette (s.u.) ersetzen.

9.1. Einzelne Symbole aus einer Zeichenkette extrahieren

In **R** lassen sich die einzelnen Symbole einer Zeichenkette mit der Operation `zeichenkette |> str_extract("")` extrahieren.

> **Warnung**
>
> Wenn Sie die Zeichenketten in einem Zeichenkettenvektor in die einzelnen Symbole zerlegen möchten, dann erhalten Sie für jede Zeichenkette einen eigenen Vektor, der die Symbole der Zeichenkette enthält. R kann diese Vektoren nicht einfach zu einem grossen Vektor zusammensetzen. Daher werden die Ergebnisse als Listen geschützt und zu einem Ergebnisvektor zusammengefasst.
> In einem zweiten Schritt können die extrahierten Symbole mit `unnest()` in der Stichprobe erweitert werden.

Beispiel 9.1 (Zeichenkettenvektor in die einzelnen Symbole zerlegen)**.**

```
tibble(zeichenkette = c("Daten", "und", "Information")) |>
    mutate(
        symbol = zeichenkette |> str_extract_all("")
    ) |>
    unnest(symbol)
```

zeichenkette <chr>	symbol <chr>
Daten	D
Daten	a
Daten	t
Daten	e
Daten	n
und	u
und	n
und	d
Information	I
Information	n
Information	f
Information	o

zeichenkette <chr>	symbol <chr>
Information	r
Information	m
Information	a
Information	t
Information	i
Information	o
Information	n

Bei solchen Operationen sollten Sie die Quelldaten nicht überschreiben. Erstellen Sie immer einen neuen Vektor für extrahierte Symbole und Zeichenketten. So bleibt der Bezug zu den ursprünglichen Werten erhalten.

9.2. Nicht-druckbare Zeichen

Definition 9.1. Als **nicht-druckbare Zeichen** werden Symbole bezeichnet, die bei der Darstellung einer Zeichenkette nicht angezeigt werden. Die nicht-druckbaren Zeichen zählen zur Länge einer Zeichenkette und verändern den Inhalt einer Zeichenkette.

Beispiel: Die Zeichenkette `Hallo` unterscheidet sich von der Zeichenkette `Hal<0x08>lo`.

Excel und R behandeln nicht-druckbare Zeichen unterschiedlich. In Excel werden die nicht-druckbaren Zeichen für die Darstellung und für Vergleiche entfernt, jedoch werden die nicht-druckbaren Zeichen bei der Länge und beim Extrahieren berücksichtigt. In R werden nicht-druckbare Zeichen bei der Darstellung und bei Vergleichen berücksichtigt. In Excel können wir mit der `IDENTISCH()`-Funktion zwei Zeichenketten nach den gleichen Regeln wie in R vergleichen.

Zu den nicht-druckbaren Zeichen gehören auch Leerzeichen, Tabulatoren und Zeilenumbrüche. Wir können diese speziellen nicht-druckbaren Zeichen nur erkennen, wenn sie von druckbaren Zeichen umgeben sind.

Deutlich wird das an den folgenden Zeichenketten:

- `Hallo`
- `Hal<0x07>lo`, wobei das Symbol `0x07` für einen Piepton steht
- `Hal<0x08>lo`, wobei das Symbol `0x08` für einmal Rückwärtslöschen steht.

Diese drei Zeichenketten haben in Excel und R die Längen 5, 6 und 6. Excel stellt alle drei Zeichenketten als “Hallo” dar. Ausserdem werden die Zeichenketten als gleich ausgewertet. R wertet die Zeichenketten aus und stellt nicht-druckbare Zeichen prinzipiell als ein Leerzeichen dar. Das Symbol `0x08` wird von R ausgewertet und deshalb wird das vorangehende Symbol gelöscht. Entsprechend wird in unserem Fall `Halo` angezeigt. Ebenfalls werden alle drei Zeichenketten in R als ungleich ausgewertet.

9.3. Die leere Zeichenkette

Ein besonderer Fall ist die *leere Zeichenkette*. Die leere Zeichenkette wird oft als Platzhalter genutzt. Die leere Zeichenkette ist das *neutrale Element* für die Verknüpfung von Zeichenketten.

Die leere Zeichenkette wird in R immer durch doppelte Anführungszeichen eingerahmt. Soll eine leere Zeichenkette als Wert in einer Zelle eingegeben werden, dann ist ein einfacher Apostroph (‘) einzugeben.

> **Hinweis**
>
> In R dürfen Sie *optional* auch einfache Anführungszeichen als alternative Zeichenkettenmarkierungen verwenden. Weil das einfache Anführungszeichen (’) und der Backtick (`) sehr ähnlich aussehen aber eine andere Bedeutung haben, sollte nur das doppelte Anführungszeichen (") in R verwendet werden.

Beispiel 9.2 (Die leere Zeichenkette)**.**

```
leereZeichenkette = ""
```

9.3.1. Schreibweise ändern

9.4. Zeichenketten trennen

In R gibt es verschiedene Funktionen, die aus einem Wert mehrere Werte erzeugen. Das gilt insbesondere für die Zeichenkettenfunktionen. In diesem Zusammenhang nimmt die Funktion `str_split()` eine besondere Position ein, weil sie relativ oft gebraucht wird. Diese Funktion trennt eine Zeichenkette entlang eines Trennzeichens bzw. eines Trennmusters und gibt die Ergebniswerte zurück.

> **Hinweis**
>
> Wir können uns die Funktion `str_split()` als eine flexiblere Variante von Excels *“Text in Spalten”*-Befehl vorstellen. Die Parameter für diese Funktion sind eine Zeichenkette sowie das Trennmuster. Das Ergebnis ist ein Vektor aus Zeichenketten.

Natürlich wäre es toll, wenn wir `str_split()` zum Umformen einer Stichprobe verwenden könnten. Allerdings führt das Ergebnis zu *Listenvektoren*, mit denen wir nicht leicht arbeiten können. Das illustriert das folgende Beispiel. In diesem Beispiel trennen wir die Zeichenketten im Vektor `text` an den Leerzeichen, sodass wir einzelne Worte erhalten.

Beispiel 9.3 (Zeichenketten mit `str_split()` trennen)**.**

```
library(tidyverse)

texte = tibble(text = c("Daten und Information", "Klimatologie Informatik"))

texte |>
    mutate(
        getrennter_text = text |> str_split(" ")
    )
```

text <chr>	getrennter_text <list>
Daten und Information	Daten, und, Information
Klimatologie Informatik	Klimatologie, Informatik

Im Vektor `getrennter_text` stehen nun Listen mit unterschiedlicher Länge. Wären diese Listen Vektoren, dann könnten wir mit der Funktion `pivot_longer()` die Werte transponieren. Das funktioniert mit eingebetteten Listen leider nicht, weil die Werte nicht über mehrere Vektoren verteilt sind, sondern alle im gleichen Vektor stehen.

Definition 9.2. Enthält ein Vektor Listen mit Werten, dann werden die Listenwerte als **eingebettete** (engl. *nested*) **Werte** bezeichnet.

Um an eingebettete Werte zu gelangen, müssen wir sie zuerst "ausbetten". Dazu verwenden wir die Funktion `unnest()`. Mit dieser Funktion werden eingebette Listen in einen Vektor entpackt.

Beispiel 9.4 (Entpacken von getrennten Zeichenketten mit `unnest()`)**.**

```
texte |>
    mutate(
        getrennter_text = text |> str_split(" ")
    ) |>
    unnest(getrennter_text) -> texte_getrennt

texte_getrennt
```

text <chr>	getrennter_text <chr>
Daten und Information	Daten
Daten und Information	und
Daten und Information	Information
Klimatologie Informatik	Klimatologie
Klimatologie Informatik	Informatik

Beachten Sie hier, dass alle nicht aufgelösten Vektoren für jeden Listeneintrag erweitert werden.

Jetzt können wir mit diesen Werten wie gewohnt weiterarbeiten.

> **Hinweis**
>
> Die Umkehrfunktion von `unnest()` ist die Funktion `nest()`. Beide Funktionen werden ausführlich in Kapitel 16 behandelt.

Das Beispiel 9.4 können wir mit der folgenden Operation zurück in die Listenform bringen. Dabei beachten wir, dass wir den neuen Vektor benennen und diesem die Werte aus dem Ursprungsvektor übergeben müssen.

Beispiel 9.5 (Zeichenketten einbetten)**.**

```
texte_getrennt |>
    nest(getrennter_text = getrennter_text)
```

text <chr>	getrennter_text <list>
Daten und Information	Daten, und, Information
Klimatologie Informatik	Klimatologie, Informatik

9.5. Teilzeichenketten extrahieren

9.6. Suchen und Ersetzen

9.6.1. Position einer Teilzeichenkette finden

9.6.2. Teilzeichenketten austauschen

9.7. Mustererkennung

Definition 9.3. Reguläre Ausdrücke sind Zeichenketten, mit denen *komplexe Suchmuster* für Zeichenketten beschrieben werden.

Ein regulärer Ausdruck beschreibt die Struktur einer gesuchten Zeichenkette. Reguläre Ausdrücke sind eine Standardtechnik zum Suchen-und-Ersetzen.

R verfügt mit regulären Ausdrücken über eine leistungsfähige Mustererkennung für Zeichenketten. Diese Mustererkennung steuern wir über *reguläre Ausdrücke*(Vignette regex) oder *Regulärausdruck* (Sauer, 2019). Reguläre Ausdrücke erlauben es, Muster in Zeichenketten

zu finden und diese Muster durch etwas anderes auszutauschen. In R schreiben wir reguläre Ausdrücke als Zeichenketten mit einer besonderen Musterbeschreibungssprache. Wir können an viele Zeichenkettenfunktionen solche regulären Ausdrücke als Parameter übergeben.

Die wichtigsten Symbole zur Musterbeschreibung mit regulären Ausdrücken sind die folgenden Symbole und Symbolkombinationen:

- `.` - beschreibt das Vorkommen eines beliebigen Symbols
- `\s` - beschreibt alle Symbole die als Leerzeichen gelten
- `\d` - beschreibt alle Ziffern
- `\w` - beschreibt alle Buchstaben unabhängig von der Gross- und Kleinschreibung
- `*` - beschreibt das Auftreten von Sequenzen von 0 oder mehreren der voranstehenden Symbole
- `+` - beschreibt das Auftreten von Sequenzen von 1 oder mehreren der voranstehenden Symbole
- `?` - beschreibt das Auftreten von Sequenzen von 0 oder 1 des voranstehenden Symbols
- `{}` - beschreibt das Auftreten von Sequenzen der angegebenen Länge des voranstehenden Symbols
- `^` - steht für den Anfang der Zeichenkette
- `$` - steht für das Ende der Zeichenkette
- `[]` - "Symbolbereich": Die Symbole zwischen den beiden Klammern beschreiben die möglichen Symbole an der Position in der Zeichenkette
- `()` - Gruppiert eine Teilzeichenkette

9.7.1. Normale Zeichen in Mustern

Normale Buchstaben oder Ziffern haben keine besondere Bedeutung und bedeuten, dass an der entsprechenden Stelle das jeweilige Symbol vorkommen muss.

```
zeichenkettenVektor = c( "Daten und Information", "Datenverarbeitung", "Informatik", "Date

# der reguläre Ausdruck wäre eigentlich "\w\s\w" die zusätzlichen Backslashs
# zeigen R an, dass wir den Backslash in unserem Muster haben möchten.

regulaererAusdruck = "Daten In"

zeichenkettenVektor |> str_detect(regulaererAusdruck)
# erzeugt c(FALSE FALSE FALSE TRUE TRUE)
```

9.7.2. Mustersymbole in R verwenden

Wenn wir ein Symbol in unserem Muster aufnehmen wollen, das normalerweise ein besonderes Symbol für reguläre Ausdrücke ist, dann müssen wir diesem Symbol einen *Backslash* voranstellen. Dieses Voranstellen wird als **"Escaping"** bezeichnet. Weil R reguläre Ausdrücke als Zeichenketten behandelt, müssen wir aufpassen, denn der Backslash ist auch ein

reserviertes Symbol in Zeichenketten. Deshalb ist der zweite Backslash notwendig, um den ersten Backslash vor der Zeichenketteninterpretation zu schützen.

```
zeichenkette = "Daten und Information"

# der reguläre Ausdruck wäre eigentlich "\w\s\w" die zusätzlichen Backslashs
# zeigen R an, dass wir den Backslash in unserem Muster haben möchten.

regulaererAusdruck = "\\w\\s\\w"

zeichenkette |> str_replace(regulaererAusdruck, "p x") # erzeugt "Datep xnd Information"

# Hinweis, um alle Vorkommnisse des Musters auszutauschen, müssen wir
# str_replace_all() verwenden!
```

9.7.3. Multiplikatoren

Die Symbole `*`, `+`, `?` und `{}` werden als *Multiplikatoren* bezeichnet. So können Wiederholungen in Mustern abgebildet werden.

Mit diesen Elementen können wir Zeichenketten beschreiben, ohne die genaue Abfolge der Symbole zu kennen.

Beispiele:

```
"ab"       # erkennt ab
"a?b"      # erkennt b und ab
"a*b"      # erkennt b, ab, aab, aaab, aaaab usw.
"a+b"      # erkennt ab, aab, aaab, aaaab usw.
"a{2}b"    # erkennt aab
"a{2,4}b"  # erkennt aab, aaab und aaaab
"a.b"      # erkennt aab, acb, adb, a3b, a-b usw.
"a.*b"     # erkennt ab, acb, acdb, a-!%b usw.

"a\\sb"    # erkennt "a b" oder "a     b" (Achtung doppelter Backslash!)
"\\w\\d"   # erkennt einen Buchstaben, der von einer Ziffer gefolgt wird  (Achtung doppelt
"a[cd]?b"  # erkennt ab, acb und adb

"ab$"      # erkennt ab nur am Ende der Zeichenkette
"^ab"      # erkennt ab nur am Anfang der Zeichenkette
```

Gelegentlich wollen wir ein Muster bis zu einem bestimmten Symbol in unserer Zeichenkette finden. In diesem Fall können wir einen negierten Symbolbereich angeben.

Beispiel 9.6. Gegeben ist die folgende Zeichenkette:

```
aquaponics = "
   The term aquaponics [7] is coined by combining
    two words: aquaculture, which refers to fish
    farming, and hydroponics-the technique of growing
    plants without soil.[16]"
```

Wir möchten nun die Zeichenkette ab dem Wort `term` und der öffnenden eckigen Klammer der Referenz markieren. D.h. wir wollen nicht ein beliebiges Zeichen und wollen nicht alle Symbole bis auf die öffnende Klammer explizit ausschliessen. Stattdessen können wir einen *negierten* Symbolbereich angeben. In unserem Fall erlauben wir jedes Zeichen, ausser die öffnende eckige Klammer. Weil die eckige Klammer eine besondere Bedeutung für reguläre Ausdrücke hat, müssen wir sie entsprechend mit Backslash "escapen". Unser regulärer Ausdruck muss entsprechend `"termn [^\\[]+\\["`. Der Teil `[^\\[]+` bedeutet dabei, "*alle Symbole ausser der öffnenden eckigen Klammer [*". Die beiden Backslashes sind dabei die notwendige Escape-Sequenz, um die Klammer vom Symbolbereich zu unterscheiden.

Der folgende Code demonstriert diesen regulären Ausdruck.

```
aquaponics |>
    str_match("term [^\\[]+\\[")
```

Das Ergebnis eines regulären Ausdrucks ist normalerweise immer nur der erste Treffer.

```
"term aquaponics ["
```

Um alle Treffer eines Musters zu erhalten, muss für die entsprechende `_all`-Variante der Funktion verwendet werden. Die folgenden Funktionen haben eine solche Funktionsvariante:

Bedeutung	Erster Treffer	Alle Treffer
Finde und extrahiere ein Suchmuster	`str_extract()`	`str_extract_all()`
Finde ein Suchmuster und gebe die Zeichenkette zurück	`str_match()`	`str_match_all()`
Finde ein Suchmuster und gebe die Position des Treffers zurück	`str_locate()`	`str_locate_all()`
Finde und lösche ein Suchmuster	`str_remove()`	`str_remove_all()`
Finde ein Suchmuster und ersetze den Treffer durch eine andere Zeichenkette	`str_replace()`	`str_replace_all()`
Zeige Suchtreffer für ein Suchmuster an	`str_view()`	`str_view_all()`

9.8. Tokens

Die Bibliothek `tidytext` stellt viele hilfreiche Funktionen zur quantitativen Textanalyse bereit. Beim Tokenisieren müssen verschiedene Regeln beachtet werden, damit das richtige Ergebnis erzeugt wird. Diese Regeln müssen wir zum Glück nicht im Detail kennen. Die Bibliothek `tidytext` stellt uns die Funktion `unnest_tokens()` sowie ein paar Hilfsfunktionen bereit. Mit diesen Funktionen können wir Texte leicht in Tokens zerlegen.

```
library(tidyverse)
library(tidytext)

tibble(
    rohtext = read_file("text/marketing_4.txt")
) -> Rohdaten
```

Hier fällt uns eine neue Funktion auf: `read_file()`. Mit dieser Funktion können beliebige Textdateien eingelesen werden. R versucht bei dieser Funktion nicht, strukturierte Daten zu finden. Stattdessen wird der gesamte Inhalt der Datei als Zeichenkette zurückgegeben.

Nun können wir die Daten mit Hilfe von `unnest_tokens()` in Tokens zerlegen. Die Funktion `unnest_tokens()` funktioniert analog zu `str_split()` gefolgt von `unnest()`. Sie erleichtert diese Funktionsfolge, indem sie nicht nur Leerzeichen, sondern auch alle anderen Symbole und Satzzeichen, die keine Worte darstellen aus der ursprünglichen Zeichenkette entfernt.

```
Rohdaten |>
    unnest_tokens(worte, rohtext)  |>
    head()
```

worte<chr>
neues
lehrbuch
für
modernes
marketing
marketing

In diesem Beispiel sehen wir die grundsätzliche Arbeitsweise der Funktion. Wir übergeben ein Stichprobenobjekt der Funktion über die Funktionsverkettung. Anschliessend übergeben wir der Funktion als ersten Parameter den Namen des Zielvektors (hier: `worte`) und als zweiten Parameter den Namen des Quellvektors (hier `rohtext`). Alle Texte im Vektor `rohtext` werden durch die Funktion in Worte zerlegt und anschliessend konsequent in die Kleinschreibung überführt. Abschliessend wird der ursprüngliche Vektor `rohtext` aus der Ergebnisstichprobe entfernt.

9.8.1. Deutsche Gross- und Kleinschreibung

Während die Gross- und Kleinschreibung im Englischen (mit Ausnahme von Eigennamen) nicht signifikant ist, haben deutsche Worte in Gross- und Kleinschreibung eine andere inhaltliche Bedeutung. Falls diese Bedeutung erhalten bleiben soll, kann der optionale Parameter `to_lower = FALSE` übergeben werden. Der Code ändert sich dann wie folgt:

```
Rohdaten |>
    unnest_tokens(worte, rohtext, to_lower = FALSE)  |>
    head()
```

worte<chr>
Neues
Lehrbuch
für
modernes
Marketing
Marketing

9.8.2. Texte in Sätze zerlegen

Wenn wir Texte in Sätze zerlegen wollen, dann verwenden wir die Funktion `unnest_sentences()`.

```
Rohdaten |>
    unnest_sentences(saetze, rohtext, to_lower = FALSE)
```

saetze <chr>
Neues Lehrbuch für modernes Marketing.
Marketing wandelt sich im rasanten Tempo.
Auf Kundenwünsche oder neue Technologien muss nicht nur auf der operativen, sondern auch auf der strategischen Ebene in nahezu Echtzeit reagiert werden.
Welche Instrumente und Frameworks dem Marketing dabei zur Verfügung stehen, zeigt das gerade erschienene Lehrbuch des Instituts für Marketing Management:
“Marketingmanagement.
Building and Running the Business.
Mit Marketing Unternehmen transformieren” Marketing hat in den vergangenen Jahren einen Paradigmenwechsel durchlaufen.

Dieser Code ist übrigens identisch mit dem folgenden Code:

```
Rohdaten |>
    unnest_tokens(saetze, rohtext, to_lower = FALSE, token = "sentences")
```

In diesem Beispiel fällt auf, dass `unnest_sentences()` streng entlang den Satztrennzeichen (`. ! ?`) trennt und in Anführungszeichen eingebettete Sätze ebenfalls trennt. Dieses Verhalten kann nur dadurch beeinflusst werden, dass die Texte im Vorfeld entsprechend vorbereitet werden. In solchen Fällen empfiehlt es sich, eingebettete Satzenden durch ein Semikolon (`;`) zu ersetzen.

In anderen Fällen wollen wir nach dem Trennen die Satzzeichen aus den Sätzen vollständig entfernen. Das erreichen wir mit dem Parameter `strip_punct = TRUE`. Dieser Parameter veranlasst, dass alle Satzzeichen aus den Sätzen entfernt werden. Dazu gehört auch das Zeichen für das Satzende.

9.8.3. Absätze trennen

Damit wir Absätze trennen können, müssen die Rohtexte entsprechend vorbereitet sein. Absätze werden in Textformaten durch eine zusätzliche Leerzeile markiert. Das weicht von der üblichen Vorgehensweise bei der Arbeit mit Word ab. Dort markiert der einfache Zeilenumbruch einen Absatz.

> **Praxis**
>
> Trennen Sie beim Transkribieren mit MS Word Absätze **immer** mit einer zusätzlichen Leerzeile. In dieser Leerzeile dürfen **keine** anderen Symbole stehen (auch keine Leerschläge). Diese Zeile erzeugen Sie durch zwei Zeilenumbrüche mit der Eingabetaste. Sie halten sich so alle Optionen für die nachfolgende Analyse offen.

Sind die Textdaten entsprechend vorbereitet, dann können wir unsere Texte mit der Funktion `unnest_paragraphs()` in Absätze gliedern.

```
Rohdaten |>
    unnest_paragraphs(
        saetze,
        rohtext,
        to_lower = FALSE,
        paragraph_break = "\r\n\r\n"
   )
```

> **Warnung**
>
> Der zusätzliche Parameter `paragraph_break = "\r\n\r\n"` ist hier notwendig, weil die Daten aus Wort heraus als `Nur Text (.txt)` gespeichert wurden.

9.8.4. n-Gramme extrahieren

n-Gramme sind ein wichtiges Werkzeug für einen besseren inhaltlichen Überblick. Worte sind spezielle n-Gramme mit $n = 1$. Bei dieser Länge haben bestimmte häufig vorkommende Worte (die Stoppworte) die unerwünschte Eigenschaft, das Ergebnis inhaltlich zu verzerren. Durch das Entfernen dieser Worte wird aber immer auch ein Teil der inhaltlichen Bedeutung entfernt. Dieser besondere Effekt von Stoppworten tritt bei n-Grammen mit $n > 1$ nicht auf.

n-Gramme extrahieren wir mit der Funktion `unnest_ngrams()`. Dabei wird der Text in Wortsequenzen mit der Länge `n` gegliedert.

```
Rohdaten |>
    unnest_ngrams(ngram, rohtext, to_lower = FALSE) |>
    head()
```

ngram<chr>
Neues Lehrbuch für
Lehrbuch für modernes
für modernes Marketing
modernes Marketing Marketing
Marketing Marketing wandelt
Marketing wandelt sich

Dieser Code ist identisch mit dem folgenden Code:

```
Rohdaten |>
    unnest_ngrams(ngram, rohtext, to_lower = FALSE, n = 3) |>
    head()
```

> **Praxis**
>
> Typische n-Gram-Längen sind `3`, `5` oder `7`.
> Wobei die n-Gram-Länge von `3` am üblichsten ist.

Am Beispiel ist die Arbeitsweise von n-Gram-Tokenisierung erkennbar: Beginnend vom ersten Wort wird eine Sequenz von `n` Worten extrahiert und so lange ein Wort weiter gegangen bis keine Wortsequenz der Länge `n` mehr möglich ist. Dabei ist zu beachten, dass Satz- und Zeilengrenzen nicht automatisch berücksichtigt werden. Um nur inhaltlich zusammenhängende n-Gramme zu erhalten, müssen zwei Tokenisierungen nacheinander vorgenommen werden. Das folgende Beispiel zeigt eine 3-Gram-Zerlegung auf Satzebene.

```
Rohtext |>
    unnest_sentences(saetze, rohtext, to_lower = FALSE)  |>
    unnest_ngrams(ngram, saetze, to_lower = FALSE)  |>
    head()
```

ngram<chr>
Neues Lehrbuch für
Lehrbuch für modernes
für modernes Marketing
Marketing wandelt sich
wandelt sich im
sich im rasanten

9.9. Rezepte

9.9.1. Word als Datenquelle

Kodierte Texte sind keine Tabellen, sondern liegen als Word-Dokumente auf unserem Rechner. In diesen Word-Dokumenten sind unsere Daten Paare von markierten Textstellen und Kommentaren . Diese Paare wollen wir extrahieren.

Dieses Beispiel hat vier Beiträge vom offiziellen Blog des Studienschwerpunkts Marketing der ZHAW kopiert und kodiert. Dabei wurden Bilder entfernt. Mit diesem Beispiel untersuchen wir, ob diese Beiträge eine genderneutrale Sprache verwenden. Dazu wurden Substantive jeweils kategorisiert und mit der jeweiligen Geschlechtlichkeit (`feminin`, `maskulin`, `neutral`) kodiert.

9.9.1.1. Schritt 1: Datei einlesen und bereinigen

Wenn wir unsere Texte mit Word kodiert haben, können wir sie mit Hilfe der `docxtractr` Bibliothek einlesen.

```
library(tidyverse)
library(docxtractr)
```

Nun können wir kodierte Dokumente in unsere R-Umgebung importieren. Dazu verwenden wir die besondere Funktion `read_docx()`. Diese Funktion liest das ganze Word-Dokument ein. Mit Hilfe der Funktion `docx_extract_all_cmnts()` sammeln wir unsere markierten Textstellen ein.

```
read_docx("kodiert/marketing_1.docx") |>
    docx_extract_all_cmnts(include_text = TRUE) -> documentCodes
```

Drei Vektoren sind für uns von besonderer Bedeutung:

- Der Vektor `comment_text` enthält nun unsere Kodierungen.
- Der Vektor `id` zeigt uns die Reihenfolge der Kommentare im Dokument.
- Der Vektor `word_src` enthält den beim Kodieren markierten Text.

Oft müssen wir unsere Codes und Texte noch bereinigen. In diesem Fall, sind die Codes nicht durchgehend einheitlich geschrieben und in jedem Kommentar stehen zwei Codes. Wir wollen deshalb die Codes trennen und vereinheitlichen. In unserem Fall sind enthalten die Kommentartexte *immer* zwei Kodierungen. Weil immer die gleiche Kode-Anzahl in den Kommentaren vorliegt, können wir die Funktion `separate()` verwenden.

> **Tipp**
>
> Die Funktion `separate()` trennt einen Zeichenketten Vektor in mehrere Zeichenkettenvektoren auf.

> **Wichtig**
>
> Die `separate()`-Funktion darf nur verwenden werden, wenn alle Kommentare die gleiche Anzahl von Codes enthalten!

```
documentCodes |>
    select(id, comment_text, word_src) |>
    separate(comment_text, into =c("kategorie", "gender"), sep =",") |>
    mutate(
        gender = gender |> str_trim() |> str_to_lower(),
        kategorie = kategorie |> str_trim() |> str_to_lower()
    ) -> kodierteDaten
```

Nach diesem Schritt sind unsere Codes vereinheitlicht und für jede Textstelle können nun die Codes bearbeitet werden.

Falls wir unterschiedlich viele Kodierungen in den Kommentaren vorliegen, müssen wir die Funktion `str_split()` und anschliessend `unnest()` verwenden.

```
data |>
    select(id, comment_text, word_src) |>
    mutate(
        code = comment_text |> str_split(",")
    ) |>
    unnest(code) -> allgemeinereExtraktion
```

9.9.1.2. Schritt 2: Word Hyperlinks entfernen

In diesem Beispiel enthalten die markierten Texte Hyperlinks zu externen Seiten. Diese Links stören uns bei der Analyse und deshalb entfernen wir sie aus den Textstellen.

```
kodierteDaten |>
    mutate(
        word_src = word_src |> str_remove("HY\\s?\\S+ \"[^\"]+\"[^\"]+\"[^\"]+\" "),
    ) -> kodierteDaten2
```

9.9.1.3. Schritt 3: Kategorien organisieren

```
kodierteDaten2 |>
    count(kategorie)
```

kategorie	count
aktivität	7
aufgabe	1
cas	10
eigenschaft	4
fähigkeit	1
funktion	5
gruppe	12
information	1
kontext	1
konzept	1
organisation	1
person	6
produkt	3
technik	1
zeit	7

Beachten Sie, dass der vorherige Teilschritt in der Regel **nicht** berichtet wird, weil das Ergebnis nur dazu dient, die Codes den richtigen Variablen zuzuordnen.

Die Codes im Vektor `Kategorie` gehören zu verschiedenen Variablen. Diese Zuordnung muss explizit erfasst werden.

```
Akteure = c("person", "organisation", "gruppe")

Studiengang = c("cas", "bsc", "mas", "msc")
```

```
Funktion = c("aktivität", "eigenschaft", "fähigkeit",
             "möglichkeit", "funktion", "produkt", "anwendung")

Kontext = c("kontext", "konzept", "technik", "zeit", "information")
```

Wir können so die Kodierungen als Merkmalsausprägungen nominalskalierter Variablen verwenden, beschreiben und auswerten.

9.9.1.4. Weiterführende Textanalyse

Wir können nun weiter analysieren und besonders häufige Worte für unsere Codes auswerten.

Wir extrahieren die einzelnen Worte aus der jeweiligen Markierung und entfernen Artikel und andere oft benutzte Worte aus unseren Daten.

Anschliessend zählen wir die verbleibenden Worte nach Kategorien und Gender.

Damit das Ergebnis einfacher zu lesen ist, wird das Ergebnis mit `arrange()` sortiert. Weil in unserem Fall sehr viele Worte nur einmal vorkommen, tragen diese nicht viel zum Gesamtinhalt bei. Daher werden diese Worte mit dem abschliessenden Filter für die Darstellung entfernt.

```
library(tidytext)

stopwords_DE = tibble(
    word = stopwords::stopwords("de", source = "stopwords-iso")
)

kodierteDaten2 |>
    unnest_tokens(word, word_src) |>
    anti_join(stopwords_DE) |>
    count(word, kategorie, gender) |>
    arrange(desc(n)) |>
    filter(n > 1)
```

word	kategorie	gender	n
cas	cas	maskulin	9
marketing	funktion	neutral	3
mitstudierenden	gruppe	neutral	3
austausch	aktivität	maskulin	2
digital	cas	maskulin	2
marketing	cas	maskulin	2
netzwerk	gruppe	neutral	2

word	kategorie	gender	n
npo	cas	maskulin	2
referenten	gruppe	maskulin	2

Solche Auswertungen geben zusätzliche Einblicke in die Inhalte und helfen bei der Interpretation der Daten.

9.9.2. Word als Datenquelle

Kodierte Texte sind keine Tabellen, sondern liegen als Word-Dokumente auf unserem Rechner. In diesen Word-Dokumenten sind unsere Daten Paare von markierten Textstellen und Kommentaren . Diese Paare wollen wir extrahieren.

Dieses Beispiel hat vier Beiträge vom offiziellen Blog des Studienschwerpunkts Marketing der ZHAW kopiert und in MS Word kodiert. Dabei wurden Bilder entfernt. Mit diesem Beispiel untersuchen wir, ob diese Beiträge eine genderneutrale Sprache verwenden. Dazu wurden Substantive jeweils kategorisiert und mit der jeweiligen Geschlechtlichkeit (`feminin`, `maskulin`, `neutral`) kodiert.

9.9.2.1. Schritt 1: Datei einlesen und bereinigen

Wenn wir unsere Texte mit Word kodiert haben, können wir sie mit Hilfe der `docxtractr` Bibliothek einlesen.

```
library(tidyverse)
library(docxtractr)
```

Nun können wir kodierte Dokumente in unsere R-Umgebung importieren. Dazu verwenden wir die besondere Funktion `read_docx()`. Diese Funktion liest das ganze Word-Dokument ein. Mit Hilfe der Funktion `docx_extract_all_cmnts()` sammeln wir unsere markierten Textstellen ein.

```
read_docx("kodiert/marketing_1.docx") |>
    docx_extract_all_cmnts(include_text = TRUE) -> documentCodes
```

Drei Vektoren sind für uns von besonderer Bedeutung:

- Der Vektor `comment_text` enthält nun unsere Kodierungen.
- Der Vektor `id` zeigt uns die Reihenfolge der Kommentare im Dokument.
- Der Vektor `word_src` enthält den beim Kodieren markierten Text.

Oft müssen wir unsere Codes und Texte noch bereinigen. In diesem Fall, sind die Codes nicht durchgehend einheitlich geschrieben und in jedem Kommentar stehen zwei Codes. Wir wollen deshalb die Codes trennen und vereinheitlichen. In unserem Fall sind enthalten die Kommentartexte *immer* zwei Kodierungen. Weil immer die gleiche Kode-Anzahl in den Kommentaren vorliegt, können wir die Funktion `separate()` verwenden.

> **Tipp**
>
> Die Funktion `separate()` trennt einen Zeichenketten Vektor in mehrere Zeichenkettenvektoren auf.

> **Warnung**
>
> Die `separate()`-Funktion darf nur verwenden werden, wenn alle Kommentare die gleiche Anzahl von Codes enthalten!

```
documentCodes |>
    select(id, comment_text, word_src) |>
    separate(comment_text, into =c("kategorie", "gender"), sep =",") |>
    mutate(
        gender = gender |> str_trim() |> str_to_lower(),
        kategorie = kategorie |> str_trim() |> str_to_lower()
    ) -> kodierteDaten
```

Nach diesem Schritt sind unsere Codes vereinheitlicht und für jede Textstelle können nun die Codes bearbeitet werden.

Falls wir unterschiedlich viele Kodierungen in den Kommentaren vorliegen, müssen wir die Funktion `str_split()` und anschliessend `unnest()` verwenden.

```
data |>
    select(id, comment_text, word_src) |>
    mutate(
        code = comment_text |> str_split(",")
    ) |>
    unnest(code) -> allgemeinereExtraktion
```

9.9.2.2. Schritt 2: Word Hyperlinks entfernen

In diesem Beispiel enthalten die markierten Texte Hyperlinks zu externen Seiten. Diese Links stören uns bei der Analyse und deshalb entfernen wir sie aus den Textstellen.

```
kodierteDaten |>
    mutate(
```

```
        word_src = word_src |> str_remove("HY\\s?\\S+ \"[^\"]+\"[^\"]+\"[^\"]+\" "),
    ) -> kodierteDaten2
```

9.9.2.3. Schritt 3: Kategorien organisieren

```
kodierteDaten2 |>
    count(kategorie)
```

kategorie	count
aktivität	7
aufgabe	1
cas	10
eigenschaft	4
fähigkeit	1
funktion	5
gruppe	12
information	1
kontext	1
konzept	1
organisation	1
person	6
produkt	3
technik	1
zeit	7

Beachten Sie, dass der vorherige Teilschritt in der Regel **nicht** berichtet wird, weil das Ergebnis nur dazu dient, die Codes den richtigen Variablen zuzuordnen. Die Codes im Vektor `Kategorie` gehören zu verschiedenen Variablen. Diese Zuordnung muss explizit erfasst werden.

```
Akteure = c("person", "organisation", "gruppe")

Studiengang = c("cas", "bsc", "mas", "msc")

Funktion = c("aktivität", "eigenschaft", "fähigkeit",
             "möglichkeit", "funktion", "produkt", "anwendung")

Kontext = c("kontext", "konzept", "technik", "zeit", "information")
```

Wir können so die Kodierungen als Merkmalsausprägungen nominalskalierter Variablen verwenden, beschreiben und auswerten.

9.9.2.4. Weiterführende Textanalyse

Wir können nun weiter analysieren und besonders häufige Worte für unsere Codes auswerten.

Wir extrahieren die einzelnen Worte aus der jeweiligen Markierung und entfernen Artikel und andere oft benutzte Worte aus unseren Daten.

Anschliessend zählen wir die verbleibenden Worte nach Kategorien und Gender.

Damit das Ergebnis einfacher zu lesen ist, wird das Ergebnis mit `arrange()` sortiert. Weil in unserem Fall sehr viele Worte nur einmal vorkommen, tragen diese nicht viel zum Gesamtinhalt bei. Daher werden diese Worte mit dem abschliessenden Filter für die Darstellung entfernt.

```
library(tidytext)

stopwords_DE = tibble(
    word = stopwords::stopwords("de", source = "stopwords-iso")
)

kodierteDaten2 |>
    unnest_tokens(word, word_src) |>
    anti_join(stopwords_DE) |>
    count(word, kategorie, gender) |>
    arrange(desc(n)) |>
    filter(n > 1)
```

word	kategorie	gender	n
cas	cas	maskulin	9
marketing	funktion	neutral	3
mitstudierenden	gruppe	neutral	3
austausch	aktivität	maskulin	2
digital	cas	maskulin	2
marketing	cas	maskulin	2
netzwerk	gruppe	neutral	2
npo	cas	maskulin	2
referenten	gruppe	maskulin	2

Solche Auswertungen geben zusätzliche Einblicke in die Inhalte und helfen bei der Interpretation der Daten.

9.9.3. Kodierte Daten aus mehreren Word-Dateien einlesen.

Kodierte Texte sind keine Tabellen, sondern liegen in mehrere Dateien auf unserem Rechner. Diese Dateien sollen in einem Schritt eingelesen werden und in ein Stichprobenobjekt umgewandelt werden.

9.9.3.1. Lösung

```
library(tidyverse)
library(docxtractr)

datenordner = "kodiert"

tibble(
    datei = list.files(
                    path = datenordner,
                    pattern = "^[^~]+.docx$"
                )
) |>
    group_by(datei) |>
    mutate(
        pfad = str_c(datenordner, "/", datei),
        codes = read_docx(pfad) |>
                    docx_extract_all_cmnts(include_text = TRUE) |> list()
    ) |>
    ungroup() |>
    unnest(codes) -> alleCodes
```

9.9.3.2. Erklärung

Die Code-Beispiele basieren auf Dateien aus dem Beispieldaten

Wenn wir unsere Texte mit Word kodiert haben, können wir sie mit Hilfe der `docxtractr` Bibliothek einlesen.

```
library(tidyverse)
library(docxtractr)
```

Dazu erstellen wir uns ein Stichprobenobjekt zur Unterstützung, in das wir die Namen der Dateien einlesen.

```
datenordner = "kodiert"

tibble(
    datei = list.files(
                    path = datenordner,
                    pattern = "^[^~]+.docx$"  # nur reguläre Word Dokumente auswählen
                )
)  -> dateinamen
```

Die Funktion `list.files()` gibt einen Vektor mit allen Datennamen im angegebenen Verzeichnis `path` zurück. Mit dem Parameter `pattern` können Dateien nach ihrem Namen noch gezielter ausgewählt werden. Der hier gezeigte *reguläre Ausdruck* wird als Teil eines *logischen Ausdrucks* verwendet, um reguläre Word-Dokumente zu erhalten. Hier müssen wir aufpassen, denn die Dateiendung reicht nicht aus. Word erzeugt beim Bearbeiten einer Datei Hilfsdokumente, die ebenfalls auf `docx` enden, aber im vorderen Teil des Dateinamens eine Tilde (~) haben. Diese Dateien können wir nicht verwenden und sie dürfen deshalb nicht in unserer Dateiliste vorkommen.

Anschliessend können wir die kodierten Dokumente einzeln einlesen. Dabei müssen wir beachten, dass die Funktion `read_docx()` nur eine Datei gleichzeitig einlesen kann. Wir müssen deshalb über die Dateinamen mit `group_by()` gruppieren. Dadurch erhalten wir Teilstichproben mit genau einen Dateinamen.

```
dateinamen |>
    group_by(datei) |>
    mutate(
        pfad = str_c(datenordner, "/", datei),
        codes = read_docx(pfad) |>
                    docx_extract_all_cmnts(include_text = TRUE) |> list()
    ) |>
    ungroup() |>
    unnest(codes) -> alleCodes
```

> **Warnung**
>
> Beachten Sie, dass Sie mit dem Parameter `include_text = TRUE` nicht nur die Kodierung einlesen, sondern auch den Text, der beim Kodieren markiert wurde.

Mit dieser Operation lesen wir jede einzelne Datei ein. In der Variablen `alleCodes` liegen nun alle vorgenommenen Kodierungen mit den relevanten Zusatzinformation. Weil die Dateinamen Teil der Stichprobe ist, kann jeder Code und jeder markierte Text der Ursprungsdatei zugeordnet werden.

9.9.3.3. Lösung für normale Textdateien

Das gleiche Prinzip funktioniert auch für beliebige Textdateien.

```
datenordnet = "texte"
dateiendung = "txt"

tibble(
    datei = list.files(
                    path = datenordner,
                    pattern = str_c("\\.", dateiendung, "$")
                )
) |>
    group_by(datei) |>
    mutate(
        pfad = str_c(datenordner, "/", datei),
        codes = read_file(pfad)
    ) |>
    ungroup() -> eingeleseneTexte
```

10. Faktoren

R kennt diskrete Daten in zwei Varianten.

1. Diskrete Daten als Abfolge von Werten in einem Vektor.
2. Diskrete Daten als Faktor.

Base-R-Funktionen, wie `read.csv()` oder `data.frame()`, erstellen automatisch Faktoren, wenn diskrete Daten erkannt werden. Diese Daten werden immer dann erkannt, wenn es sich nicht um Zahlenwerte handelt. Dieser Automatismus ist nicht immer erwünscht, weil nicht alle Daten automatisch Faktoren sind, wenn die Werte keine Zahlen sind, und weil ordinalskalierte Daten oft mit Zahlen erhoben werden. Deshalb lesen die modernen Funktionen der `tidyverse`-Bibliothek Daten grundsätzlich als Vektoren atomarer Datentypen ein. So können wir entscheiden, ob ein Vektor als Faktor behandelt werden soll oder nicht.

Wir erkennen Faktoren in Stichprobenobjekten von R am Symbol `<fct>` für den Datentyp und können mit der Funktion `is.factor()` prüfen, ob ein Vektor ein Faktor ist. In R ist ein Faktor ein *komplexer Datentyp*, der neben den Werten auch den geordneten Wertebereich speichert. Dieser Wertebereich wird in R als `levels` bezeichnet. Der Wertebereich eines Faktors bildet also die "*Ausprägungen*" bzw. "Ebenen" (engl. levels) der diskreten Daten ab.

10.1. Verwendung von Faktoren in R

Neben der Bedeutung für die Empirie sind Faktoren in R auch für die Darstellung von Daten von Bedeutung. Dabei wird die Ordnung der Faktorstufen für die Anordnung von Ergebnissen verwendet, die mit Hilfe eines Faktors berechnet wurden. Diese Ordnung wird von `ggplot()` und für die Sortierung der Ergebnisse gruppierter Daten verwendet. Die Verwendung von Faktoren für solche Aufgaben hat den Vorteil, dass die Reihenfolge der Datensätze nicht verändert werden muss.

Im folgenden werden die Funktionen der `tidyverse`-Bibliothek `forcats` (Wickham, 2023a) vorgestellt.

Die folgenden Beispiele verwenden die Stichprobe `digitales_umfeld.csv`

```
digitales_umfeld = read_csv("digitales_umfeld1.csv")
```

Wenn wir zum Beispiel die Anzahl der Mobilgerätetypen bestimmen möchten, dann können wir naiv vorgehen:

```
digitales_umfeld |>
    group_by(mobilgeraet) |>
    count()
```

mobilgeraet <chr>	n <int>
Android Smartphone	64
iPhone	69
Mobiltelefon	2

Wenn wir die häufigste Nennung eines Mobilgeräts als erstes in dieser Tabelle stehen haben möchten, dann bietet sich die Verwendung eines Faktors mit organisierten Faktorstufen an.

```
digitales_umfeld |>
    mutate(
        mobilgeraet = mobilgeraet |> as_factor() |> fct_infreq()
    ) |>
    group_by(mobilgeraet) |>
    count()
```

mobilgeraet <chr>	n <int>
iPhone	69
Android Smartphone	64
Mobiltelefon	2

10.2. Erstellen von Faktoren

Faktoren sind ein zentraler Bestandteil von R. Ohne die `tidyverse` Bibliothek kann ein Faktor mittels der Funktion `factor()` erstellt werden.

Gegeben sei zum Beispiel der Vektor mit den Namen der Studiengänge des ZHAW Departements LSFM:

```
Studiengaenge = c(
    "Chemie",
    "Umweltingenieurwesen",
    "Facility Management",
    "Biotechnologie",
    "Lebensmitteltechnology",
    "Applied Digital Life Sciences",
    "Biomedical Labordiagnostik"
)
```

Dieser Vektor enthält nur Zeichenketten und ist daher ein Zeichenkettenvektor. Der Funktionsaufruf `is.factor(Studiengaenge)` gibt entsprechend `FALSE` als Ergebnis zurück.

Dieser Vektor kann einfach in einen Faktor umgewandelt werden.

```
stgFaktor = Studiengaenge |> factor()

stgFaktor
```

- Chemie
- Umweltingenieurwesen
- Facility Management
- Biotechnologie
- Lebensmitteltechnology
- Applied Digital Life Sciences
- Biomedical Labordiagnostik

Das Ergebnis unterscheidet sich nicht wesentlich vom ursprünglichen Vektor. Wir können mit dem Aufruf `is.factor(stgFaktor)` überprüfen, ob es sich nun um einen Faktor handelt. Wir erhalten nun `TRUE` als Ergebnis.

Wir können nun mit der Funktion `levels()` die Faktorstufen abfragen. Diese Funktion gibt uns einen Vektor mit allen Faktorstufen zurück.

```
stgFaktor |>
    levels()
```

- Applied Digital Life Sciences
- Biomedical Labordiagnostik
- Biotechnologie
- Chemie
- Facility Management
- Lebensmitteltechnology
- Umweltingenieurwesen

Dieses Ergebnis ist etwas überraschend, weil die Reihenfolge der Faktorstufen nicht mehr mit der Reihenfolge der Werte in unserem Vektor übereinstimmt. Wir sehen am Ergebnis, dass die Funktion `factor()` die Annahme macht, dass unsere Faktorstufen alphabetisch sortiert sind. Leider ist das oft nicht der Fall und gerade bei unsortierten Faktoren müssen die Werte Präsentationen oft neu arrangiert werden. Deshalb hat sich die Konvention eingebürgert, für die initiale Reihenfolge von Faktorstufen das erste Auftreten des jeweiligen Werts zu wählen. Dazu müssen wir der `factor()`-Funktion auch die Faktorstufen mitgeben.

```
stgFaktor = Studiengaenge |>
    factor(
         # erzeugt alle Faktorstufen in der Reihenfolge des ersten Auftretens,
         # selbst wenn Werte doppelt auftreten.
```

```
        Studiengaenge |> unique()
    )
```

Nun können wir mit der `levels()`-Funktion die Reihenfolge der Faktorstufen überprüfen.

Weil diese Vorgehensweise eine Konvention moderner R-Programmierung ist, gibt es eine Funktion, die uns diesen Schritt kompakter schreiben lässt. Dazu verwenden wir die Funktion `as_factor()`.

```
stgFaktor = Studiengaenge |>
    as_factor()
```

Wenn wir einzelne Vektoren in einem Stichprobenobjekt in Faktoren umwandeln wollen, dann führen wir eine Umwandlung mit `mutate()` durch.

Wir wollen nur den Vektor `mobilgeraet`, `geschlecht` und `digitalisiert` in Faktoren umwandeln.

```
digitales_umfeld |>
    mutate(
        mobilgeraet = mobilgeraet |> as_factor(),
        geschlecht = geschlecht |> as_factor(),
        digitalisiert = digitalisiert |> as_factor()
    ) -> duFaktorisiert

duFaktorisiert
```

geschlecht <fct>	alter <dbl>	tage <dbl>	monate <dbl>	geburtsjahr <dbl>	digitalisiert <fct>	mobilgeraet <fct>
Männlich	23	8474	278	1998	3	iPhone
Männlich	27	9970	327	1994	6	Android Smart-phone
Männlich	27	10131	332	1994	6	iPhone
Weiblich	25	9253	304	1996	5	Android Smart-phone
Männlich	25	9363	307	1996	6	iPhone
Andere	23	8750	287	1997	2	iPhone
…	…	…	…	…	…	…

Wir sehen nun, dass die faktorisierten Vektoren nun den Datentyp Factor (`fct`) haben.

10.3. `forcats` - Faktoren leicht gemacht

Die Arbeit mit Faktoren ist in Base-R nicht immer ganz einfach. Wollschläger (2017, Kap. 2.6) zeigt detailliert, wie Faktoren mit R erstellt und manipuliert werden. Das ist zum Teil recht komplex und aufwändig. Zum Glück versteckt die `forcats`-Bibliothek die Komplexität von R-Faktoren vor uns, sodass wir präzise ausdrücken können, wie wir die interne Struktur unserer diskreten Daten organisieren möchten.

> **Hinweis**
>
> Die Funktionen der `forcats`-Bibliothek sind verfügbar, sobald Sie die `tidyverse`-Bibliothek eingebunden haben.

Die `forcats`-Bibliothek bietet neun Funktionen zum Umorganisieren von Faktoren.

10.4. Organisieren von Faktorstufen

Die zentrale Funktion in R von Faktorstufen ist ihre Bedeutung für die Ordnung der Werte des jeweiligen Faktors. Indem wir die Faktorstufen organisieren, können wir die Werte des Vektors strukturieren, ohne ihre Reihenfolge zu ändern.

Für die Organisation von Faktorstufen gibt es vier häufig vorkommende Aufgaben:

1. Organisation entlang einer vordefinierten Reihenfolge (*Skala*),
2. Organisation entlang der internen Organisation eines Datentyps,
3. Organisation entlang der Häufigkeit eines Werts,
4. Organisation entlang des Auftretens in der Stichprobe.

Die erste Aufgabe tritt immer ein, wenn wir mit etablierten Methoden arbeiten. In diesem Fall ist die Reihenfolge der Faktorstufen bereits bekannt. In solchen Fällen sprechen wir von der Zuordnung einer Skala. Für solche Zuordnungen verwenden wir die Funktion `fct_relevel()`.

Zum Beispiel wollen wir die Faktorstufen in der Stichprobe digitales Umfeld nach einer externen Vorgabe festlegen, sodass die folgende "Ladies-First" Reihenfolge gilt:

```
geschlecht_faktorstufen = c( "Weiblich", "Andere", "Keine Angabe", "Männlich" )
```

Nun können wir den Faktor `geschlecht` entsprechend umformen.

```
duFaktorisiert |>
    mutate(
        geschlecht = geschlecht |> fct_relevel( geschlecht_faktorstufen )
    ) -> digitales_umfeld_externe_skala
```

> ⚠ Warnung
>
> Wenn Sie mit `fct_relevel()` eine vorgegebene Skala als Vektor übergeben, die nicht vollständig in der Stichprobe abgedeckt wurde, dann erhalten Sie eine kryptische Warnung, die auf `Unknown levels in `f`:` endet. **Diese Warnung können Sie ignorieren**. R fügt die fehlenden Faktorstufen nicht ein, behält aber deren innere Organisation bei.

Die zweite Aufgabe ist die Organisation der Faktorstufen nach der internen Organisation des jeweiligen Datentyps. Liegen die Werte als Zahlen vor, dann werden die Faktorstufen entsprechend des jeweiligen Nennwerts sortiert. Liegen die Werte als Zeichenketten vor, dann werden die Faktorstufen alphabetisch sortiert. Diese Organisation entspricht der Vorgehensweise der meisten Base-R Funktionen. Falls unsere Daten als Zahlen (oder Wahrheitswerte) vorliegen, können wir für diese Vorgehensweise die Funktion `fct_inseq()` verwenden.

```
duFaktorisiert |>
    mutate(
        digitalisiert = digitalisiert |>  fct_inseq()
    ) -> duFaktorisiert_sequenziell
```

Im Fall von Zeichenketten als Faktorstufen, müssen wir die bestehenden Faktorstufen selbst umsortieren und dann als neue Vorgabe mit `fct_relevel()` übergeben. Im folgenden Beispiel ist das alphabetische Sortieren der Faktorstufen eine eingebettete Funktionskette.

```
duFaktorisiert |>
    mutate(
        mobilgeraet = mobilgeraet |> fct_relevel( mobilgeraet |> levels() |> sort() )
    ) -> duFaktorisiert_alphabetisch
```

Die dritte Aufgabe ist die Organisation nach der Häufigkeit eines Werts in einer Stichprobe. Diese Vorgehensweise ist besonders für nominalskalierte Daten interessant, um über die Häufigkeiten die Reihenfolge der Faktorstufen festzulegen. Hierbei hilft die Funktion `fct_infreq()`. Die Faktorstufen werden so den Häufigkeiten entsprechend in der Stichprobe ab- bzw. aufsteigend organisiert.

```
duFaktorisiert |> mutate(
        mobilgeraet = mobilgeraet |>  fct_infreq()
    ) -> duFaktorisiert_frequenz
```

Bei der vierten Aufgabe sollen die Faktorstufen entlang der Reihenfolge des ersten Auftretens in der Stichprobe erfolgen. Dieser Fall tritt meistens dann ein, wenn die Faktorstufen umorganisiert wurden und wieder in die ursprüngliche Reihenfolge gebracht werden müssen. Die Funktion `fct_inorder()` übernimmt diese Aufgabe.

```
duFaktorisiert |> mutate(
        mobilgeraet = mobilgeraet |>  fct_inorder()
    ) -> duFaktorisiert_reihenfolge
```

10.4.1. Faktorstufen an den Werten eines anderen Vektors ausrichten

Mit der Funktion `fct_reorder()` können wir die Faktorstufen eines Vektors über die Werte eines anderen Vektors organisieren. Solche Ausrichtungen sind oft nach Aggregationen sinnvoll, wenn für jede Faktorstufe eines nominalskalierten Vektors genau ein Wert in einem anderen Vektor festliegt. Dabei **muss** aber immer **genau ein Wert** einer Faktorstufe entsprechen. Dabei wird für die Faktorstufen eine absteigende bzw. (optional) aufsteigende Reihenfolge des Referenzvektors angenommen.

```
msleep |>
    mutate(
        name = name |> as_factor() |> fct_reorder(sleep_total)
    )
```

Ähnlich wie beim Gruppieren, ändert sich die sichtbare Struktur des Stichprobenobjekts nicht. Es wird lediglich die interne Reihenfolge der Faktorstufen angepasst. Solche Schritte sind für aussagekräftige Visualisierungen sehr hilfreich.

10.5. Faktorstufen und Visualisierung

Faktoren erleichtern das Veranschaulichen von Daten. Der grosse Vorteil bei der Verwendung von Faktoren ist, dass sich der Code für die Visualisierung nicht ändert. Mit Hilfe der Faktorstufen geben wir `ggplot` Hinweise über die Struktur der Werte in einem Vektor. Dadurch werden unsere Visualisierungen aussagekräftiger.

10.5.1. Überzählige Achsenbeschriftungen entfernen

Faktoren werden regelmässig für die Datenvisualisierung verwendet, um diskrete Daten richtig darzustellen. Das ist besonders dann notwendig, wenn die Werte im entsprechenden Vektor mit Zahlen dargestellt werden und die Reihenfolge dieser Zahlen vom Üblichen abweicht.

Nehmen wir zum Beispiel die Stichprobe `mtcars` und den Vektor `cyl` (Zylinder) . Dieser Vektor nimmt nur die folgenden diskreten Werte an.

```
mtcars |>
    summarise( cyl = unique(cyl) )
```

cyl <dbl>
6
4
8

In beiden Fällen haben wir also diskrete Daten. Wenn wir die Daten darstellen, dann ergibt sich das folgende Bild.

```
mtcars |>
   ggplot(aes(x = cyl )) +
      geom_bar()
```

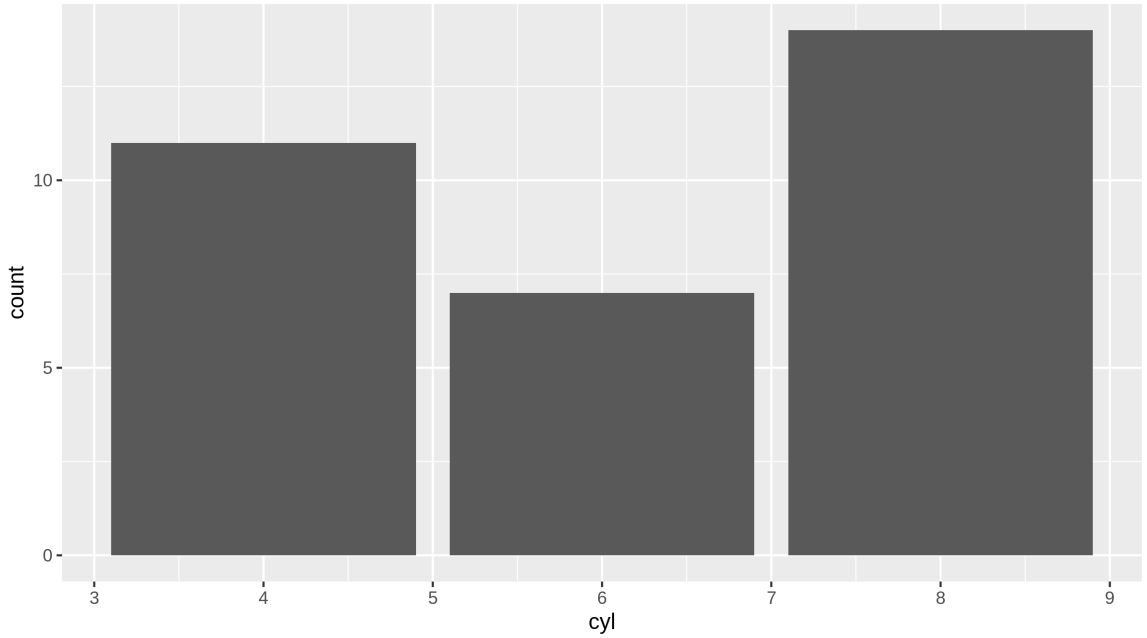


Abbildung 10.1.: mtcars cyl ohne Faktorisierung

Hier erkennen wir, dass die X-Achse Werte anzeigt, die gar nicht vorkommen können. Dieses Problem können wir leicht beheben, indem wir den Vektor faktorisieren.

```
mtcars |>
   mutate( cyl = cyl |> as_factor() )  |>
   ggplot(aes(x = cyl )) +
      geom_bar()
```

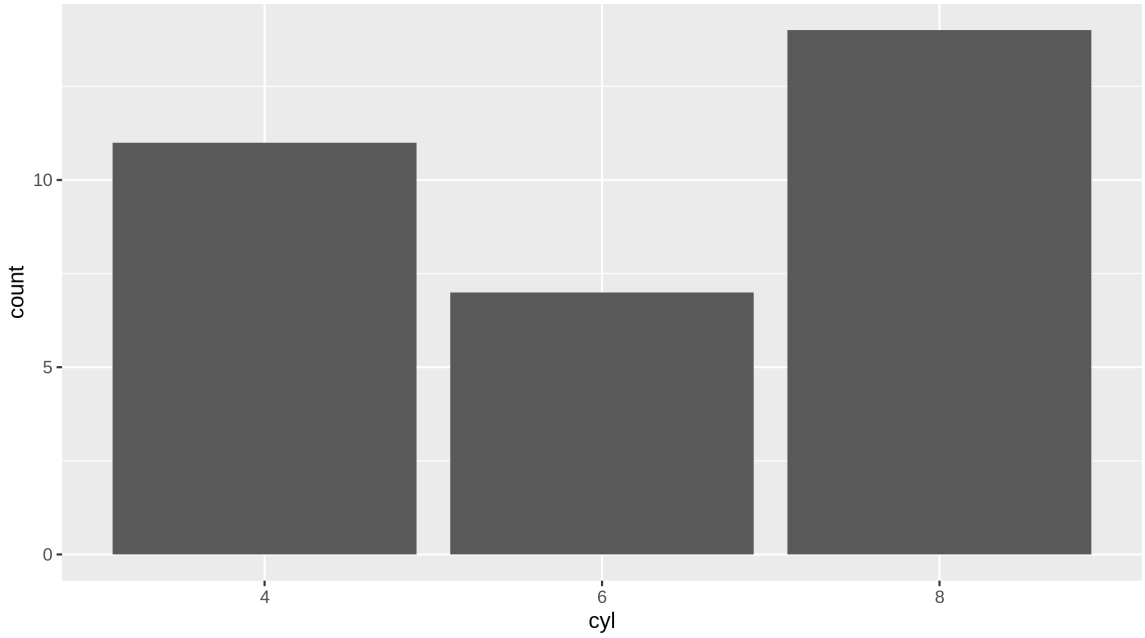


Abbildung 10.2.: mtcars cyl mit Faktorisierung

10.5.2. Sortierte Balkendiagramme

Das Arrangieren von Daten erreichen wir ebenfalls mit Faktoren. Hierzu betrachten wir die Vektoren `name` und `sleep_total` aus der Stichprobe `msleep`. Stellen wir die beiden Vektoren in einem Balkendiagramm gegenüber, dann können wir kaum die verschiedenen Spezies vergleichen.

```
msleep |>
    ggplot(aes(name, sleep_total)) +
        geom_col() +
        coord_flip()
```

Wesentlich anschaulicher wird dieses Balkendiagramm, wenn wir den `name`-Vektor vor der Visualisierung in einen Faktor umwandeln und die Faktorstufen entlang der Gesamtschlafdauer organisieren.

```
msleep |>
    mutate(
        name = name |> as_factor() |> fct_reorder(sleep_total)
    ) |>
    ggplot(aes(name, sleep_total)) +
        geom_col() +
        coord_flip()
```

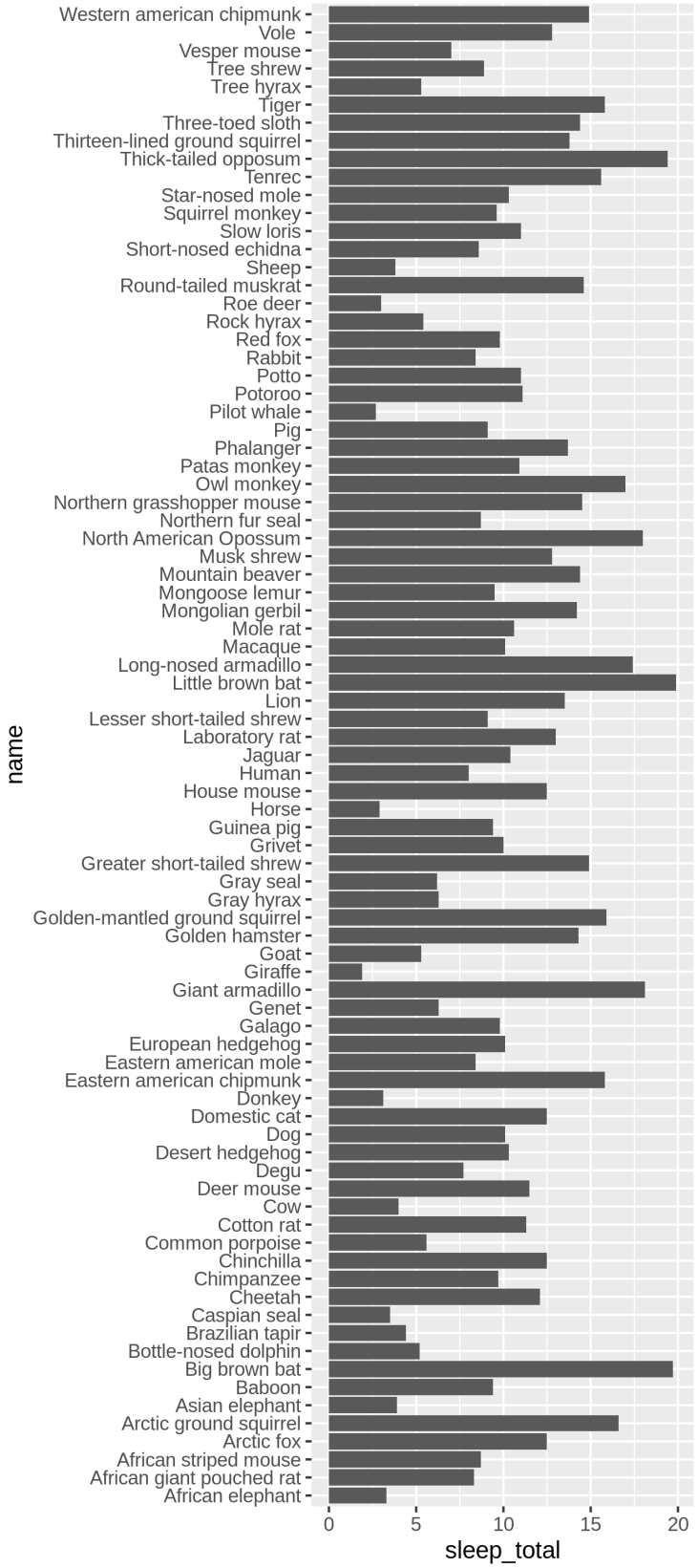


Abbildung 10.3.: msleep data ohne Faktorisierung

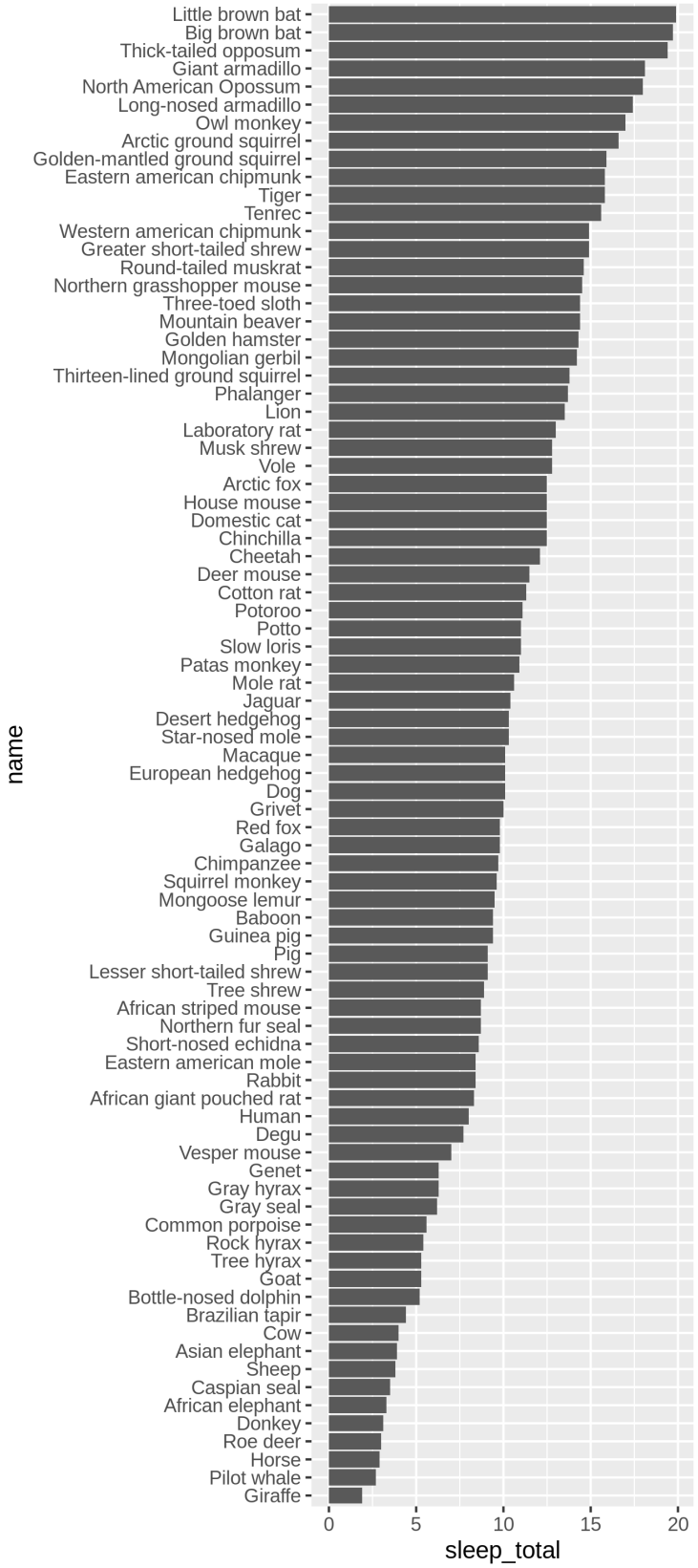


Abbildung 10.4.: msleep data mit Faktorisierung

11. Boole'sche Operationen

In R stehen die logischen Operationen als *binäre* Operatoren zur Verfügung, bzw. als Funktionen mit genau zwei Parametern. Diese logischen Operatoren sind vektorisiert. Es ist deshalb unnötig, logische Ausdrücke durch die Boole'sche Arithmetik zu ersetzen. Lediglich die Reihenfolge der Ausführung dieser Operatoren folgt der gleichen Regel wie die Arithmetik.

> **Merke**
>
> - Das logische Und entspricht der Multiplikation.
> - Das logische Oder entspricht der Addition.

Daraus folgt, dass immer zuerst das logische Und und erst danach das logische Oder ausgewertet wird. Dieser Regel folgt auch R.

Die Tabelle 11.1 stellt die logischen Operationen und die verschiedenen Schreibweisen gegenüber.

Tabelle 11.1.: Die wichtigsten logischen Operatoren und ihre Entsprechung in R

Operation	neutrales Element	Mathematisch	R	arithmetische Operation
Nicht	-	$\neg$	`!`	$1 - a$
Und	WAHR	$\wedge$	`&`	$a \cdot b$
Oder	FALSCH	$\vee$	`|`	$a + b$
Exklusiv-Oder/Antivalenz	-	$\oplus$	`xor()`	(a - b)^2

> **Achtung**
>
> Es gibt neben den beiden Operatoren `&` und `|` auch die gedoppelte Varianten `&&` und `||`. Diese Varianten arbeiten auf den Binärwerten von Ganzzahlen und werden normalerweise **nicht** im Zusammenhang mit logischen Ausdrücken verwendet.

Das **Logisches Nicht** wird in R durch den Nicht-Operator (`!`) ausgedrückt. Dieser Operator wird auf jeden Wert eines Vektors einzeln angewandt.

Beispiel 11.1 (Logisches Nicht)**.**

```r
logischer_vektor = c(TRUE, FALSE, FALSE, TRUE, TRUE)

! logischer_vektor
```

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

R wandelt numerische Werte automatisch in Wahrheitswerte um, wenn sie mit logischen Operationen verwendet werden. Dabei gilt:

- `FALSE` entspricht `0`
- `TRUE` entspricht *ungleich* `0`

Beispiel 11.2.

```r
! c(1, 2, 0, 4, 0)
```

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

Wenn Sie in R zwei Vektoren mit dem Und- (`&`), dem Oder-Operator (`|`) oder der Antivalenz (`xor()`) verknüpfen, dann werden die Werte **immer** *paarweise* miteinander verglichen. Ein einzelner Vektor kann nicht an die Funktion des jeweiligen Operators übergeben werden.

Beispiel paarweise Verknüpfung

```r
vektor_a = c(TRUE, FALSE, FALSE, TRUE, TRUE)
vektor_b = c(TRUE,  TRUE, FALSE, FALSE, TRUE)

vektor_a & vektor_b
```

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

11.1. Logische Aggregationen mit `reduce()`

> **Merke**
>
> Um logische Vektoren in **R** zu aggregieren, muss der Vektor **reduziert** (engl. *reduce*) werden. Das *Reduzieren* ist eine besondere *Aggregation* über eine Reihe von Werten, bei der jeder Wert gemeinsam mit dem Ergebnis der Vorgängerwerte an eine Funktion übergeben wird.

Beispiel 11.3 (Aggregation logischer Vektoren)**.**

```r
beispielWerte = c(TRUE, TRUE, FALSE, TRUE)

beispielWerte |> reduce(`&`)
```

```
[1] FALSE
```

```r
beispielWerte |> reduce(`|`)
```

```
[1] TRUE
```

```r
beispielWerte |> reduce(`xor`)
```

```
[1] TRUE
```

> **!** Wichtig
>
> Beim Reduzieren muss beachtet werden, dass eine Funktion und nicht den Operator übergeben wird. Deshalb muss der jeweilige logische Operator in Backticks (`) gesetzt und so als Funktionsbezeichner markiert werden.

11.2. Vergleiche

Neben den logischen Operationen sind Vergleiche ein wichtiges Konzept, das wir in logischen Ausdrücken regelmässig anwenden.

Es gibt genau sechs (6) Vergleichsoperatoren:

- Gleich (`==`)
- Ungleich (`!=`)
- Grösser als (`>`)
- Grösser gleich (`>=`)
- Kleiner als (`<`)
- Kleiner gleich (`<=`)

> **⚠** Warnung
>
> Vergleiche erfordern, dass beide Werte vom gleichen Datentyp sind.

Die Vergleiche funktionieren für alle fundamentalen Datentypen.

Bei Zeichenketten wertet R die alphabetische Reihenfolge der Symbole vom Beginn einer Zeichenkette aus, um grösser oder kleiner Vergleiche durchzuführen.

11.2.1. Die Existenz eines Werts in einem Vektor überprüfen

Häufig müssen Sie überprüfen, ob ein Wert in einer Liste vorkommt. Grundsätzlich können Sie das mit komplizierten logischen Verknüpfungen in der Art von Beispiel 11.4 schreiben.

Beispiel 11.4 (Existstenzprüfung ohne `%in%`).

```
meinWert = 3
wertVektor = c(8, 2, 3)

meinWert == wertVektor[1] | meinWert == wertVektor[2] | meinWert == wertVektor[3]
```

```
[1] TRUE
```

Einfacher ist aber ein sogenannter *Existenztest*. Dabei wird überprüft, ob ein Wert in einem Vektor vorkommt. Ein solcher Test lässt sich wie in Beispiel 11.5 schreiben:

Beispiel 11.5 (Existstenzprüfung mit `%in%`).

```
meinWert = 3
wertVektor = c(8, 2, 3)

meinWert %in% wertVektor
```

```
[1] TRUE
```

Entsprechend der Definition des Existenzvergleichs $\in$ funktioniert R's `%in%`-Operator auch für Vektoren als linker Operand.

11.3. Fälle unterscheiden

11.3.1. Bedingte Operationen

R kennt die beiden Schlüsselworte `if` und `else`, um die Ausführung Operationsblöcken an Bedingungen zu knüpfen. Das Schlüsselwort `if` erwartet einen logischen Ausdruck, der **genau** einen Wahrheitswert zurückgibt. Logische Ausdrücke mit Vektoren sind damit nicht möglich.

Soll sowohl die Bedingung als auch die Alternative behandelt werden, dann muss das Schlüsselwort `else` in der gleichen Zeile stehen, wie das Ende des Blocks für die Bedingung.

Beispiel 11.6 (Ungültige Vektorbedingung mit `if`).

```r
werte = c(-1, 2, 0, 1)

if (werte > 1) {
    werte = werte - 1
} else {
    werte = 0
}
```

```
Error in if (werte > 1) {: Bedingung hat Länge > 1
```

> **Praxis**
>
> Bedingte Operationen sind in R nur selten notwendig. Die einzige relevandte Anwendung ist Datentypkontrolle für Parameter bevor die eigentliche Operation durchgeführt wird.

Beispiel 11.7 (Datentypprüfung mit `if`)**.**

```r
if (!is.list(werte)) {
    stop("Variable enthält keine Liste")
}
```

```
Error in eval(expr, envir, enclos): Variable enthält keine Liste
```

11.3.2. Vektorisierte Unterscheidungen

Häufiger als Bedingungen kommen in R vektorisierte Unterscheidungen vor. Dafür stehen zwei Funktionen zur Verfügung:

- `ifelse()`
- `case_when()`

Die Funktion `ifelse()` hat drei Parameter und immer einen Vektor als Ergebnis. Die Parameter sind:

1. Einen vektorisierten logischen Ausdruck.
2. Eine Operation für den Fall, dass der logische Ausdruck Wahr (`TRUE`) ergibt.
3. Eine Operation für den Fall, dass der logische Ausdruck Falsch (`FALSE`) ergibt.

Die Ergebnisse der beiden Operationen stehen im Ergebnisvektor an den Positionen, an denen der logische Ausdruck Wahr oder Falsch ergab.

Beispiel 11.8 (Vektorisierte Unterscheidung mit `ifelse()`)**.**

```
ifelse(werte > 1, werte * 2, 0)
```

```
[1] 0 4 0 0
```

Die Funktion `case_when()` erlaubt es, mehrere miteinander verbundene vektorisierte Unterscheidungen in einer Operation durchzuführen. Dazu werden logische Ausdrücke mit Ergebnisoperationen bzw. -Werten verknüpft. Eine Ergebnisoperation wird dann ausgeführt, wenn der zugehörige logische Ausdruck Wahr (`TRUE`) ergibt. Die logischen Ausdrücke werden in der angegebenen Reihenfolge geprüft, wobei die Operation abbricht, sobald ein logischer Ausdruck Wahr ergibt.

Beispiel 11.9 (`case_when()` über einen Vektor)**.**

```
case_when(
    werte > 0 ~ "positiv",
    werte == 0 ~ "null",
    werte < 0 ~ "negativ"
)
```

```
[1] "negativ" "positiv" "null"    "positiv"
```

Für den Fall, dass für einen Wert kein logischer Ausdruck Wahr ergibt, kann ein **Rückfallergebnis** angegeben werden. Dieses Rückfallergebnis muss mit `.default =` eingeleitet werden.

Beispiel 11.10 (`case_when()` mit Rückfallergebnis)**.**

```
case_when(
    werte > 0 ~ "positiv",
    werte < 0 ~ "negativ",
    .default = "null"
)
```

```
[1] "negativ" "positiv" "null"    "positiv"
```

11.4. Filtern

Das Filtern von Werten in Vektoren und Stichproben ist ein zentrales Element von R. Dafür stehen viele Funktionen bereit. Es ist auch möglich über die Index-Operatoren zu filtern.

In der Praxis wird meistens die Funktion `filter()` zum Auswählen von Datensätzen verwendet. Diese Funktion ermöglicht es, einen Datenrahmen mittels eines logischen Ausdrucks einzuschränken. Die Funktion `filter()` hat zwei Parameter:

1. Den Datenrahmen und
2. Den logischen Ausdruck für die Auswahl der Datensätze.

Das Ergebnis ist ein Datenrahmen, der nur Datensätze enthält, für die der logische Ausdruck Wahr (`TRUE`) ergibt.

Beispiel 11.11 (Filtern)**.**

	A	B	C
1	*Name*	*Sprache*	*Einwohner:innen*
2	Basel	deutsch	173863
3	Genf	französisch	203856
4	Lugano	italienisch	62315
5	Zug	deutsch	30934
6	Zürich	deutsch	421878

Für diese Stichprobe möchten wir wissen, wie viele Einwohner in Städten mit mehr als 100000 Einwohnenden leben?

Diese Frage beantworten wir mit der folgenden Logik:

1. Alle Städte mit mehr als 100000 Einwohner:innen filtern.

2. Die Einwohner:innen der gefilterten Städte zusammenzählen.

Der logische Ausdruck zum Filtern ist `` `Einwohner:innen` > 100000 ``, weil dieser Ausdruck nur für die Datensätze Wahr wird, wenn im Vektor `Einwohner:innen` der Wert grösser als `100000` ist. Nach dem Filtern im ersten Schritt liegt nur noch die folgende Stichprobe vor:

	A	B	C
1	*Name*	*Sprache*	*Einwohner:innen*
2	Basel	deutsch	173863
3	Genf	französisch	203856
6	Zürich	deutsch	421878

Für diese Teilstichprobe muss im zweiten Schritt nur noch die Summe über den Vektor `Einwohner:innen` gebildet werden.

Daraus ergibt sich die folgende Funktionskette:

```
read_delim("daten/einwohnende.psv",                                    ①
           delim = "|",
           trim_ws = T,
           show_col_types = F) |>
    filter(`Einwohner:innen` > 100000) |>                              ②
    summarise(
        Gesamteinwohnende = sum(`Einwohner:innen`)
    )
```

① `read_delim()` muss verwendet werden, weil ein besonderes Trennzeichen benutzt wird.
② Filter operation.

```
# A tibble: 1 x 1
  Gesamteinwohnende
              <dbl>
1            799597
```

11.4.1. `NA`-Werte filtern

Die Funktion `drop_na()` ist eine spezielle Filterfunktion, deren Ergebnis nur Datensätze enthält, in denen bei keinem Vektor den Wert `NA` vorkommt. Die Funktion hat keinen Effekt, wenn ein Datenrahmen nur gültige Werte enthält (Beispiel 11.12).

Beispiel 11.12 (`drop_na()` ohne Effekt)**.**

11.5. Selektieren

> Praxis
>
> Die `tidyverse` Bibliothek umfasst die `tidyselect`-Funktionen. Dabei handelt es sich um eine Reihe von Hilfsfunktionen, die die Vektorenauswahl nachvollziehbarer macht. Auf der `tidyselect`-Homepage finden sich ausführliche Code-Beispiele.

In R können Vektoren mit der Funktion `select()` selektiert werden. Dieser Funktion werden Regeln übergeben, nach denen die Vektoren auswählt werden sollen. Die einfachste Regel ist die direkte Eingabe der Vektorennamen. Ein typischer Anwendungsfall ist die Datenbereinigung, damit die Funktion `drop_na()` nicht zu viele Datensätze löscht. Diese Situation kommt vor, wenn ein Datenrahmen viele fehlende Werte enthält, die ungleichmässig in den Vektoren vorkommen. Die Analyse muss deshalb auf die gewünschten Vektoren beschränkt werden.

Für die folgenden Beispiele verwenden wir die Daten der Befragung zum digitalen Umfeld, die mit der `read_csv()`-Funktion eingelesen wird.

```
stichprobe = read_csv("daten/befragung_digitales_umfeld/deviceuse.csv")
```

```
Rows: 76 Columns: 4
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (4): q00_demo_gen, q00_demo_studyload, q01_mob_typ, q12_fav_apps

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.
```

11.5.1. Vektoren direkt selektieren

Wir wollen die Vektoren `q00_demo_gen` (Gender), `q00_demo_studyload` (Studienmodell) und `q01_mob_typ` (Mobile OS des Smartphones) auswählen.

Beispiel 11.13 (Direktes selektieren)**.**

```
stichprobe |>
    select(q00_demo_gen, q00_demo_studyload, q01_mob_typ) |>
    head()
```

```
# A tibble: 6 x 3
  q00_demo_gen q00_demo_studyload q01_mob_typ
  <chr>        <chr>              <chr>
1 Weiblich     Vollzeit           iPhone
2 Weiblich     Vollzeit           iPhone
3 Weiblich     Teilzeit           iPhone
4 Weiblich     Teilzeit           iPhone
5 Männlich     <NA>               Android Smartphone
6 Weiblich     Vollzeit           iPhone
```

Durch diesen Aufruf von `select()` wird der Datenrahmen auf die drei ausgewählten Vektoren reduziert.

11.5.2. Alle ausser die benannten Vektoren selektieren

Vektoren direkt zu benennen ist eine einfache direkte Methode. Wenn man sehr viele Vektoren auswählen möchte, dann ist es manchmal einfacher, nur die Vektoren anzugeben, die *nicht* in der Ergebnisstichprobe enthalten sein sollen. Mit `select()` erreichen wir das, indem wir ein `-` den ungewollten Vektoren voranstellen.

Das folgende Beispiel selektiert alle Vektoren ausser `q00_demo_gen` aus der Stichprobe.

Beispiel 11.14 (Selektieren durch Ausschliessen)**.**

```
stichprobe |>
    select(-q00_demo_gen) |>
    head()
```

Wenn mehrere Vektoren ausgeschlossen werden sollen, dann müssen diese zu einem Vektor zusammengefasst werden.

Beispiel 11.15 (Selektieren durch mehrfaches Ausschliessen)**.**

```
stichprobe |>
    select(- c(q00_demo_gen, q00_demo_studyload)) |>
    head()
```

Diese Vektorenauswahl wählt alle Vektoren ausser das Geschlecht und das Studienmodell.

11.5.3. Vektoren mit ähnlichen Namen auswählen

Drei leistungsfähige Hilfsfunktionen für `select()` sind:

- `starts_with()`,
- `ends_with()` sowie
- `contains()`

Diesen Funktionen akzeptieren einen Teilnamen, über den mehrere Vektoren ausgewählt werden, in denen der angegebene Teil im Vektornamen vorkommt.

Diese Funktionen lassen sich mittels der `iris`-Stichprobe veranschaulichen.

Beispiel 11.16 (Selektieren mit `starts_with()`)**.**

```
iris |>
    select(starts_with("Sepal")) |>  # wählt die Vektoren Sepal.Width und Sepal.Length aus
    head()
```

```
  Sepal.Length Sepal.Width
1          5.1         3.5
2          4.9         3.0
3          4.7         3.2
4          4.6         3.1
5          5.0         3.6
6          5.4         3.9
```

Beispiel 11.17 (Selektieren mit `ends_with()`).

```
iris |>
    select(ends_with("Length")) |> # wählt die Vektoren Petal.Length und Sepal.Length aus
    head()
```

```
  Sepal.Length Petal.Length
1          5.1          1.4
2          4.9          1.4
3          4.7          1.3
4          4.6          1.5
5          5.0          1.4
6          5.4          1.7
```

11.5.4. Alle Vektoren zwischen zwei benannten Vektoren auswählen

Eine weitere Möglichkeit schneller viele Vektoren auszuwählen ist der `:`-Operator. Damit können wir alle Vektoren zwischen zwei Vektoren inklusive der benannten Vektoren auswählen.

Der folgende Aufruf veranschaulicht dies:

Beispiel 11.18 (Vektorenbereich selektieren).

```
stichprobe |>
    select(q00_demo_gen:q01_mob_typ) |>
    head()
```

```
# A tibble: 6 x 3
  q00_demo_gen q00_demo_studyload q01_mob_typ
  <chr>        <chr>              <chr>
1 Weiblich     Vollzeit           iPhone
2 Weiblich     Vollzeit           iPhone
3 Weiblich     Teilzeit           iPhone
4 Weiblich     Teilzeit           iPhone
5 Männlich     <NA>               Android Smartphone
6 Weiblich     Vollzeit           iPhone
```

Diese Vektorenauswahl wählt die Vektoren `q00_demo_gen`, `q00_demo_studyload` und `q01_mob_typ` für das Ergebnis aus.

> **Warnung**
>
> Die Reihenfolge von Vektoren kann durch andere Transformationen geändert werden. Deshalb sollte das Selektieren mit Vektorbereichen vermieden werden.

11.6. Sortieren

R erlaubt kein Sortieren über logische Ausdrücke. Es ist nur möglich, Werte nach vorgegebenen grösser-kleiner Beziehungen zu sortieren. Deshalb muss für komplexe Sortierungen ein numerischer Hilfsvektor erzeugt werden, der anschliessend sortiert werden kann.

Die Werte eines Vektors werden mit `sort()` sortiert.

Beispiel 11.19 (Vektorsortierung für zufällige Ganzzahlen)**.**

```
set.seed(10)
runif(10, min = 1, max = 10) |> trunc() |> sort()
```

```
 [1] 1 3 3 3 3 4 4 5 6 7
```

Die `sort()`-Funktion kann nur einzelne Vektoren sortieren. Das ist unpraktisch, wenn Daten in einem Datenrahmen vorliegen. In diesem Fall lassen sich die Datensätze mithilfe der Funktion `arrange()` sortieren. Der Funktion werden die Vektoren übergeben, über die eine neue Reihenfolge festgelegt werden soll. Standardmässig sortiert `arrange()` aufsteigend. Mit der Hilfsfunktion `desc()` (engl. descending = absteigen) werden Datensätze entsprechend der Werte im Sortiervektor absteigend sortiert.

Beispiel 11.20 (Absteigende Datenrahmensortierung mit `arrange()`)**.**

```
mtcars |>
    as_tibble(rownames = "model") |>
    select(model, hp, disp, mpg, am) |>
    arrange(desc(hp)) |>
    head()
```

```
# A tibble: 6 x 5
  model                hp  disp   mpg    am
  <chr>             <dbl> <dbl> <dbl> <dbl>
1 Maserati Bora       335   301  15       1
2 Ford Pantera L      264   351  15.8     1
3 Duster 360          245   360  14.3     0
4 Camaro Z28          245   350  13.3     0
5 Chrysler Imperial   230   440  14.7     0
6 Lincoln Continental 215   460  10.4     0
```

12. Vektoroperationen

> ⚠ Work in Progress

Vektoren sind zusammengesetzte Datenstrukturen, die Werte vom gleichen Datentyp enthalten. In R bilden Vektoren die Basisstruktur für *alle Daten*. R unterscheidet zwischen Listen und Vektoren, wobei für Vektoren immer sichergestellt wird, dass alle Werte vom gleichen Datentyp sind.

> ⚠ Wichtig
>
> Vektoren haben in R keine Orientierung, d.h. R unterscheidet nicht zwischen Zeilen- und Spaltenvektoren.

> ℹ Merke
>
> Alle R-Vektoren werden grundsätzlich als Spaltenvektoren behandelt.

12.1. Konkatenation

Vektoren werden in R mit der Funktion `c()` erzeugt (Beispiel 12.1).

Beispiel 12.1 (Einen Vektor aus Skalaren erstellen)**.**

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

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

Die Funktion `c()` dient nicht nur zum erzeugen von Vektoren, sondern auch zur Konkatenation von Vektoren (Beispiel 12.2). Anders als mit der `list`-Funktion, können mit der `c()`-Funktion keine komplexen Datenstrukturen erzeugt werden.

Beispiel 12.2 (Konkatenation von Vektoren)**.**

```
c(vektor, c(6, 7))
```

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

> Praxis
>
> Die Funktion `c()` sollte nur zur Konkatenation von alleinstehenden Vektoren eingesetzt werden und nie für Vektoren, die zu einem Datenrahmen gehören!

12.2. Vektorlänge

Die Länge eines R-Vektors wird mit der Funktion `length()` bestimmt. Diese Funktion liefert immer eine ganze Zahl mit der Anzahl der Vektorelemente (Beispiel 12.3).

Beispiel 12.3 (Vektorlänge bestimmen)**.**

```
c(1, 3, 2, 4) |> length()
```

```
[1] 4
```

R kennt den leeren Vektor. Dieser Vektor hat die Länge `0`. Er wird durch den argumentlosen Aufruf der Funktion `c()` erzeugt (Beispiel 12.4).

Beispiel 12.4 (Vektorlänge des leeren Vektors)**.**

```
c() |> length()
```

```
[1] 0
```

12.3. Wertereferenzierung

Jeder Wert in einem Vektor hat eine eindeutige Position. Der erste Wert hat die Position `1` und alle weiteren Positionen sind fortlaufend nummeriert. Der Wert der Position wird auch als *Vektorindex* bezeichnet.

Der Index eines Vektors wird in eckigen Klammern (bzw. Blockklammern: `[` und `]`) gerahmt angegeben. Die Blockklammern sind R's *Indexoperator*. Ein Index wird als ***Referenz*** für einen Wert verwendet. Ein Wert in einem Vektor wird über den Index referenziert.

Es werden drei Varianten der Wertereferenzierung unterschieden. Alle Varianten sind gleichwertig, dürfen aber nicht gemischt werden.

- Die Indexreferenz
- Die negative Indexreferenz
- Die Referenz mit Wahrheitsvektoren

12.3.1. Indexreferenz

Beispiel 12.5 (Index eines Vektors)**.**

```
vektor[2]
```

```
[1] 3
```

Weil alle Werte gleichzeitig Vektoren sind, können Indizes ebenfalls als Vektoren angegeben werden, um mehrere Vektorwerte gleichzeitig abzufragen. Der Ergebnisvektor enthält die Werte in der Reihenfolge

> **Achtung**
>
> Werden ungültige Indizes angegeben, füllt R diese nicht existierenden Werte mit `NA` auf.

Beispiel 12.6 (Vektorwerte über einen Indexvektor abfragen)**.**

```
vektor[c(2, 7, 4)]
```

```
[1]  3 NA  5
```

Ein Indexvektor kann länger als der Wertevektor sein, ausserdem können Indexwerte wiederholt werden. Dadurch lassen sich Vektoren systematisch erzeugen.

Beispiel 12.7 (Vektorwerte über einen Indexvektor wiederholt abfragen)**.**

```
vektor[c(2, 4, 4, 2)]
```

```
[1] 3 5 5 3
```

12.3.2. Negative Indexreferenz

Alternativ können *negative* Indizes verwendet werden. Das Vorzeichen bedeutet in diesem Fall *"ausser"*. Es werden also alle Indizes, ausser dem angegebenen, zurückgegeben (Team et al., 2023).

> **Achtung**
>
> Negative und positive Indizes dürfen nicht gemischt werden!

> **Jargon**
>
> Diese Verwendung von negativen Referenzen ist eine Eigenheit von R und wird in den meisten anderen Programmiersprachen nicht bzw. nicht so unterstützt.

Beispiel 12.8 (Negative Indizes)**.**

```
vektor[-2]
```

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

12.3.3. Referenz durch Wahrheitsvektoren

Die letzte Variante des Wertezugriffs ist der Zugriff mit einem Wahrheitsvektor. Ein Wahrheitsvektor zum Wertezugriff **muss** die gleiche Länge haben, wie der Wertevektor. Das Ergebnis dieser Operation ist ein Vektor, der nur Werte enthält, die an den Positionen stehen, an welchen im Wahrheitsvektor `TRUE` steht.

Beispiel 12.9 (Negative Indizes)**.**

```
vektor[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
```

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

Diese Art des Wertereferenzierung wird häufig mit einer Transformation (Abschnitt 12.6) als Teil eines logischen Ausdrucks kombiniert. Beispiel 12.10 referenziert nur die gereaden Werte in `vektor`.

> **Merke**
>
> Das Referenzieren mit einem logischen Ausdruck ist eine Variante des Filterns.

> **Praxis**
>
> Das Filtern durch Referenzieren sollte nur für alleinstehende Vektoren eingesetzt werden. Zum Filtern von Vektoren in Datenrahmen sollte **immer** die `dplyr`-Funktion `filter()` verwendet werden.

Beispiel 12.10 (Gerade Werte aus einem Vektor auswählen)**.**

```
vektor[vektor %% 2 == 0]
```

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

12.4. Sequenzen

Sequenzen werden mit der Funktion `seq()` erstellt. Die Funktion hat fünf Parameter. Die drei wichtigsten Parameter sind:

- `from`: Der Initialwert der Sequenz.
- `length`: Die Länge der Sequenz.
- `by`: Die Schrittweite der Sequenz.

Neben diesen Parameter gibt es noch die beiden Parameter:

- `to`: Der Endwert der Sequenz
- `along.with`: Übernimmt die Länge der Sequenz der Länge des angegebenen Vektors.

Beispiel 12.11 (Sequenzen erstellen)**.**

```
seq(5) # seq(length = 5)
```

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

```
seq(2, 5) # seq(from = 2, to = 5)
```

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

```
seq(from = 2, length = 5, by = 2)
```

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

Neben der `seq()`-Funktion kennt R den Sequenzoperator `:`. Die Operanden des Squenzoperators sind der Initial- und Endwert, wobei eine Schrittweite von `1` angenommen wird (Beispiel 12.12).

Beispiel 12.12 (Sequenzoperator verwenden)**.**

```
3:8  # identisch mit seq(from = 3, to = 8, by = 1)
```

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

12.5. Wiederholungen

Vektoren mit gleichen Werten an allen Positionen können als Sequenzen mit der Schrittweite von `0` verstanden werden. Genauso lassen sich solche Vektoren als Wiederholungen eines Werts beschreiben. Diesen Weg geht R mit der `rep()`-Funktion (rep für engl. *repeat*).

Die `rep()`-Funktion hat zwei Parameter:

- `x`: Der Wiederholungswert
- `length`: Die Länge des Zielvektors

Mit dieser Funktion lassen sich Vektoren wie der Nullvektor oder der Einsvektor leicht erzeugen (Beispiel 12.13).

Beispiel 12.13 (Null- und Einsvektor der Länge fünf erzeugen)**.**

```
rep(0, 5)
```

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

```
rep(1, 5)
```

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

Weil die Datengrundstruktur in R Vektoren sind, können Vektoren mit der `rep()`-Funktion vervielfacht werden. So lassen sich Vektoren mit Wertemustern erzeugen (Beispiel 12.14).

Beispiel 12.14 (Mustervektor erzeugen)**.**

```
rep(c(1,0), 3)
```

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

12.6. Transformationen

Vektortransformationen bedeuten in R das wiederholte Ausführen einer Funktion für jeden Wert eines Vektors. Alle arithmetischen, logischen und Vergleichsoperatoren sind automatisch Vektortransformationen (Beispiel 12.15).

Beispiel 12.15 (Einfache Vektortransformationen)**.**

```
# Arithmetische Operationen
vektor + 4
```

```
[1]  6  7  5  9  8 10
```

```
vektor %% 2
```

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

```
# Logische und Vergleichsoperationen
vektor <= 3
```

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

```
2 < vektor & vektor < 5
```

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

Eine Besonderheit von R ist die Verallgemeinerung von Skalaroperationen auf Vektoren mit *unterschiedlicher* Länge. Dabei wird der kürzere Vektor auf die Länge des längeren Vektors durch Wiederholung erweitert, so dass eine Vektortransforamtion möglich wird (Beispiel 12.16). Bei einer Vektortransformation werden die Werte an den gleichen Positionen aus zwei Vektoren paarweise miteinander verknüpft.

Beispiel 12.16 (Erweiterte Vektortransformationen)**.**

```
vektor
```

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

```
vektor + 4 # vektor + rep(4, length(vektor))
```

```
[1]  6  7  5  9  8 10
```

```
vektor + c(1, 2) # vektor + rep(c(1, 2), 3)
```

```
[1] 3 5 2 7 5 8
```

Im Beispiel 12.16 werden Vektoren mit Längen, die Vielfache des kürzesten Vektors sind. Dadurch ist sichergestellt, dass keine überzähligen Werte bleiben. Die Wiederholung des kürzeren Vektors wird auch dann ausgeführt, wenn die Vektorenlängen keine Vielfachen voneinander sind. Dabei wird die Operation nur für alle die Werte im ursprünglich längsten Vektor ausgeführt. Dadurch bleiben nicht verwendete Werte im wiederholten Vektor. Diese überbleibenden Werte erzeugen eine Warnmeldung (Beispiel 12.17).

Beispiel 12.17 (Unvollständige Vektortransformationen)**.**

```
vektor
```

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

```
vektor + c(1, 2, 3, 4, 5) # vektor + c(1, 2, 3, 4, 5, 1)
```

```
Warning in vektor + c(1, 2, 3, 4, 5): Länge des längeren Objektes
     ist kein Vielfaches der Länge des kürzeren Objektes
```

```
[1] 3 5 4 9 9 7
```

Die Vektoroperationen von R sind sehr flexibel und leistungsfähig. Allerdings lassen sich auch damit nicht alle beliebigen Vektortransformationen durchführen.

> **Merke**
>
> Beliebige Transformationen lassen sich mit `map()` aus der Bibliothek `purrr()` umsetzen.

Die `map()`-Funktion hat immer eine Liste als Ergebnis einer Vektortransforamtion. Der eigentliche Vektor ist in dieser Liste geschachtelt und muss mit `unlist()` extrahiert werden (Beispiel 8.22).

12.7. Aggregationen

Aggregationen fassen mehrere Werte eines Vektors zusammen.Das Ergebnis ist ein Vekter, der höchstens so lang ist, wie der aggregierte Vektor. Alle Aggregationen werden in R durch Funktionen realisiert. Eine solche Funktion heisst *Aggregator*.

Häufig verwendete Aggregatoren sind:

- `sum()`, zum Addieren aller Werte eines Vektors.
- `mean()`, zur Mittelwertbildung.
- `max()`, zum Finden des grössten Werts eines numerischen Vektors
- `min()`, zum Finden des kleinsten Werts eines numerischen Vektors

Das Filtern ist eine spezielle Aggregation, die zu einem neuen und oft kürzeren Vektor führt. Das Filtern von Werten eines einzelnen Vektors erfolgt über die Wertereferenzierung mit einem logischen Ausdruck (Beispiel 12.10).

Für häufig verwendete Aggregationen finden sich in R eigene Funktionen. Sollte eine spezielle Aggregation ausnahmsweise nicht existieren, dann kann eine spezieller Aggregator implementiert werden.

> **Merke**
>
> Beliebige Vektoraggregationen lassen sich mit `reduce()` aus der Bibliothek `purrr` umsetzen.

> **Wichtig**
>
> Implementieren Sie keinen Aggregator, falls eine entsprechende Funktion bereits existiert. Die meisten vordefinierten Aggregatoren sind effizienter umgesetzt, als es mit einer naiven Umsetzung in R möglich wäre.

Beim Reduzieren wird eine Reduktionsfunktion nacheinander auf die Werte eines Vektors ausgeführt, wobei das Zwischenergebnis im nächsten Schritt als Argument verwendet wird.

Die `reduce()`-Funktion erwartet einen Vektor `.x` und eine Reduktionsfunktion `.f` als Parameter. Die Reduktionsfunktion ist eine zwei-parametrige Funktion (bzw. Operator), wobei der erste Parameter als **Akkumulator** bezeichnet wird und das Zwischenergebnis des vorangegangenen Reduktionsschritts enthält.

Beispiel 12.18 (Summe als Reduktion)**.**

```
vektor |> reduce(function(acc, wert) { acc + wert })
```

```
[1] 21
```

```
# oder kürzer
vektor |> reduce(`+`)
```

```
[1] 21
```

Falls die Reduktionsfunktion auch für den ersten Wert in einem Vektor ausfgeführt werden soll, muss zusätzlich ein Initialwert angegeben werden. Typische Initialwerte für eine Reduktionsfunktion sind die neutralen Elemente der wichtigsten Operationen:

- `0` (neutrales Element der Addition)
- `1` (neutrales Element der Multiplikation)
- `TRUE` (neutrales Element des logischen Und)
- `FALSE` (neutrales Element des logischen Oder)
- `""` (leere Zeichenkette, neutrales Element der Textverkettung)
- `c()` (leerer Vektor, neutrales Element der Konkatenation)

Eine spezielle Form der Reduktion steht mit der `purrr`-Funktion `accumulate()`. Während `reduce()` nur das Endergebnis der Reduktion liefert, gibt `accumulate()` auch die Zwischenergebnisse. Das Ergebnis der `accumulate()` Funktion ist immer ein Vektor mit der gleichen Länge wie der ursprüngliche Vektor.

Beispiel 12.19 (Laufende Summe als Reduktion mit `accumulate()`)**.**

```
vektor |> accumulate(`+`)
```

```
[1]  2  5  6 11 15 21
```

12.8. Zählen

Mit der Funktion `length()` lassen sich die Werte in einem Vektor zählen.

> **Hinweis**
>
> In der Literatur wird das Zählen von Werten regelmässig hinter komplexen Algorithmen verborgen. Gelegentlich müssen Werte als Teil einer komplexen Analyse gezählt werden. Um unnötigen Code zu vermeiden oder um übermässig komplexe formulierte Arbeitsschritte zu erkennen, ist es notwendig die wichtigsten Varianten des Zählens zu kennen.

12.8.1. Zählen durch Summieren

Beim Zählen durch Summieren werden zählbare Einheiten durch eine `1` markiert und von nicht-zählbaren Einheiten getrennt, die mit einer `0` markiert wurden. Dadurch entsteht ein Vektor, der nur aus den Werten `0` und `1` besteht. Durch das Bilden der Summe ergibt sich die Anzahl der zählbaren Elemente.

Der erste Schritt lässt sich durch einen logischen Ausdruck umsetzen. Der zweite Schritt wird mit R's Summe-Funktion `sum()`.

```
(vektor > 3) |> sum()
```

```
[1] 3
```

12.8.2. Zählen durch Filtern

Beim Zählen durch Filtern werden die zählbaren Einheiten in einem separaten Vektor isoliert. Anschliessend muss nur die Länge dieses Vektors ermittelt werden.

```
vektor[vektor > 3] |> length()
```

```
[1] 3
```

12.8.3. Zählen durch Nummerieren

Beim Zählen durch Nummerieren wird ein zweiter Vektor mit den Nummern der zählbaren Einheiten verwendet. Das Maximum der Nummerierung entspricht der Anzahl der nummerierten Elemente.

```
vektor2 = 1 : length(vektor)

max(vektor2)
```

```
[1] 6
```

> Praxis
>
> In der Regel wird dieser Vektor nicht für das Zählen neu erzeugt. Stattdessen wird dieser Vektor oft für eine andere Operation erzeugt und zum Zählen wiederverwendet.

13. Matrix-Operationen

Matrizen sind in R 2-dimensionale *numerische* Vektoren, die aus Zeilen und Spalten bestehen. Die Anzahl der Zeilen und Spalten gibt die *Dimensionalität* der Matrix an. Allgemein wird von einer $m \times n$-Matrix gesprochen, wobei m für die Anzahl der Zeilen und n für die Anzahl der Spalten steht.

> **Merke**
>
> Bei Matrizen werden immer zuerst die Zeilen und dann die Spalten angegeben.

13.1. Matrizen erstellen

Eine Matrix wird in R mit der Funktion `matrix()` erstellt. Diese Funktion erwartet einen Vektor als erstes Argument. Zusätzlich muss mit `ncol` oder `nrow` angegeben werden, wie viele Spalten bzw. Zeilen die Matrix haben soll. Der optionale Paramter `byrow` zeigt an, ob die Werte zeilen- oder spaltenweise in die Matrix übernommen werden sollen. Die Zeilenschreibweise erleichtert die Eingabe einer Matrix (Beispiel 13.1).

Beispiel 13.1 (Matrix in Zeilenschreibweise erzeugen)**.**

```
( # Diese Klammer gibt das Ergebnis der Operation aus.
    matrixA = matrix(
    c(
        1, 2, 3,
        3, 2, 1
    ), ncol = 3, nrow = 2, byrow = TRUE)
)
```

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

Ist die Länge des Eingabevektors grösser als die gewünschten Zeilen- oder Spaltenanzahl erlauben, dann werden alle überzähligen Werte verworfen (Beispiel 13.2) und eine Warnung erzeugt.

Praxis

Es muss nur die Anzahl der Spalten oder der Zeilen beim Erzeugen einer Matrix angegeben werden. Der jeweils andere Wert wird aus der Länge und dem angegebenen Wert ermittelt.

Beispiel 13.2 (Matrix mit überlangem Eingabevektor erzeugen)**.**

```
matrix(
    c(
        1, 2, 3,
        3, 2, 1,
        4, 5, 6
    ), ncol = 3, nrow = 2, byrow = TRUE)
```

```
Warning in matrix(c(1, 2, 3, 3, 2, 1, 4, 5, 6), ncol = 3, nrow = 2, byrow =
TRUE): Datenlänge [9] ist kein Teiler oder Vielfaches der Anzahl der Zeilen [2]
```

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

Ist die Länge des Eingabevektors kürzer als für eine Matrix mit den gewünschten Dimensionen notwendig wäre, dann wird der Vektor wie bei einer Vektortransformation solange wiederholt, bis alle Positionen in der Matrix besetzt sind (Beispiel 13.3). Ist der Vektor kein Vielfaches der Zeilen- oder der Spaltenzahl, dann werden alle überzähligen Werte mit einer entsprechenden Warnung verworfen.

Beispiel 13.3 (Matrix durch Auffüllen des Eingabevektors erzeugen)**.**

```
matrix(
    c(
        1, 2, 3
    ), ncol = 4, nrow = 3, byrow = TRUE)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    1
[2,]    2    3    1    2
[3,]    3    1    2    3
```

13.1.1. Identitätsmatrix erzeugen

Die Identitätsmatrix ist eine quadratische Diagonalmatrix, wobei an allen Positionen der Hauptdiagonalen der Wert `1` steht. Diese Matrix wird in R mit der Funktion `diag()` erzeugt (Beispiel 13.4).

Beispiel 13.4 (5x5 Identitätsmatrix)**.**

```
diag(5)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

13.2. Matrixdimensionen

Die *Länge* einer Matrix entspricht der Anzahl der Positionen in der Matrix und kann wie bei Vektoren mit der Funktion `length()` ermittelt werden. Diese Information ist jedoch nicht sehr nützlich.

Die Dimensionalität einer Matrix wird mit der Funktion `dim()` ausgegeben. Das Ergebnis dieser Funktion gibt einen Vektor der Länge `2` zurück. Das erste Element dieses Vektors enthält die Anzahl der Zeilen und das zweite Element enthält die Anzahl der Spalten der Matrix.

Beispiel 13.5 (Matrixdimensionen mit `dim()` abfragen)**.**

```
matrixA |> dim()
```

```
[1] 2 3
```

Die beiden Werte können separat mit den beiden Funktionen `nrow()` und `ncol()` abgefragt werden.

Beispiel 13.6 (Matrixdimensionen mit `nrow()` und `ncol()` abfragen)**.**

```
matrixA |> nrow()
```

```
[1] 2
```

```
matrixA |> ncol()
```

```
[1] 3
```

13.3. Matrixwerte referenzieren

Das Referenzieren von Werten einer Matrix erfolgt analog zum Referenzieren von Vektorwerten. Der Index eines Werts einer Matrix ist über die Zeilen und die Spalten definiert.

Beispiel 13.7 (Matrixwerte referenzieren)**.**

```
matrixA[2,1]
```

```
[1] 3
```

Über diese Notation ist es möglich eine ganze Zeile oder eine ganze Spalte einer Matrix zu referenzieren.

Beispiel 13.8 (Matrixspalten und -zeilen referenzieren)**.**

```
matrixA[2,]  # Zeile 2
```

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

```
matrixA[, 2] # Spalte 2
```

```
[1] 2 2
```

⚠ Achtung

Eine Matrix bleibt im Hintergrund eine Vektorstruktur. Wird der Separator zwischen den Zeilen- und Spaltenindex weggelassen und nur *ein* Index angegeben, dann behandelt R die Matrix als Vektor in der spaltenweisen Form. Die Reihenfolge der Werte kann mit der Funktion `as.vector()` angezeigt werden. Das gleiche gilt für die Verwendung der Konkatenation ohne das dimensionstrennende Komma.

```
matrixA |> as.vector()
```

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

```
matrixA[2,3]
```

```
[1] 1
```

```
matrixA[c(2,3)]
```

```
[1] 3 2
```

```
matrixA[,c(2,3)]
```

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

13.3.1. Zeilen- und Spaltenüberschriften

Die Zeilen- und Spalten einer Matrix können in R mit Überschriften ***benannt*** werden. Das Benennen erfolgt über einen sog. Namensvektor. Diese Vektor muss die gleiche Länge haben, wie die Anzahl der zu benennenden Zeilen bzw. Spalten der Matrix, wobei die Zeilen und Spalten nicht gleichzeitig benannt werden können.

> **Merke**
>
> Die Zeilen- und Spaltennamen gehören nicht zu den Werten einer Matrix und werden in Operationen in der Regel nicht berücksichtigt.

Beispiel 13.9 (Zeilennamen zuweisen)**.**

```
rownames(matrixA) <- c("oben", "unten")
colnames(matrixA) <- c("links", "mitte", "rechts")

matrixA
```

```
      links mitte rechts
oben      1     2      3
unten     3     2      1
```

> **Praxis**
>
> Grundsätzlich kann auch die Funktionsverkettung beim Benennen der Zeilen oder Spalten einer Matrix verwendet werden.

```
matrixA |> rownames() <- c("oben", "unten")
```

Diese Verwendung sollte allerdings vermieden werden, weil die Zuweisung und der Datenstrom in diesen Fällen gegenläufig wären.

Nachdem eine Zeile oder eine Spalte benannt wurde, werden die Namen in allen nachfolgenden Zugriffen immer in das Ergebnis aufgenommen. Ausserdem können die Namen zur Referenzierung der Werte verwendet werden (Beispiel 13.10).

Beispiel 13.10 (Zeilennamen zur Rerferenzierung verwenden)**.**

```
matrixA["oben",]
```

```
 links  mitte rechts
     1      2      3
```

```
matrixA["unten", "mitte"]
```

```
[1] 2
```

Es ist möglich einzelne Zeilen- oder Spaltennamen zuzuweisen oder zu ändern (Beispiel 13.11).

Beispiel 13.11 (Zeilennamen ändern)**.**

```
colnames(matrixA)[2] <- "zentrum"

matrixA
```

```
      links zentrum rechts
oben      1       2      3
unten     3       2      1
```

Die Namen einer Matrix lassen sich entfernen, indem den Zeilen- oder Spaltennamen der Wert `NULL` zugewiesen wird.

Beispiel 13.12 (Zeilennamen entfernen)**.**

```
rownames(matrixA) <- NULL
colnames(matrixA) <- NULL

matrixA
```

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

13.4. Matrizen transponieren

Beim Transponieren einer Matrix werden die Indizes für alle Werte vertauscht. Diese Operation übernimmt die R-Funktion `t()`.

Beispiel 13.13 (Eine Matrix transponieren)**.**

```
matrixA |> t()
```

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

> **Merke**
>
> Beim **Transponieren einer Matrix** werden die Spalten in Zeilen bzw. die Zeilen in Spalten umgewandelt.

> **Tipp**
>
> Beim Transponieren werden vorhandene Zeilen- und Spaltennamen mittransponiert.

13.5. Vektorform

Matrizen kennen zwei Arten der Vektorform:

- Die spaltenweise Vektorform, bei der die Spaltenvektoren einer Matrix zu einem Vektor konkateniert werden.
- Die zeilenweise Vektorform, bei der die Zeilenvektoren einer Matrix zu einem Vektor konkateniert werden.

R speichert eine Matrix intern in ihrer spaltenweisen Vektorform. Die spaltenweise Vektorform kann also direkt über die Funktion `as.vector()` ermittelt werden. Das gleiche Ergebnis wird erzeugt, wenn die Matrix der Vektorkonkatenation `c()` als Argument übergeben wird (Beispiel 13.14).

> **Praxis**
>
> Weil bei der Verwendung der Funktion `c()` nicht leicht ersichtlich ist, dass die Vektorform ermittelt wird, sollte für diese Operation immer die Funktion `as.vector()` verwendet werden.

Beispiel 13.14 (Spaltenweise Vektorform einer Matrix)**.**

```
matrixA |> as.vector()
```

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

```
matrixA |> c()
```

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

Beim Umwandeln einer Matrix in ihre **zeilenweise Vektorform** wird in R ausgenutzt, dass diese Vektorform identisch mit der spaltenweisen Vektorform ihrer transponierten Matrix ist. Diese Gleichheit ergibt sich aus dem Transponieren, bei der Zeilen in Spalten umgewandelt werden (Beispiel 13.15).

Beispiel 13.15 (Zeilenweise Vektorform einer Matrix)**.**

```
matrixA |> t() |> as.vector()
```

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

```
matrixA |> t() |> c()
```

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

13.6. Skalar- und Vektortransformationen

Bei Skalartransformationen wird ein einzelner Wert mit jedem Wert in der Matrix verknüpft.

Beispiel 13.16 (Skalaraddition mit einer Matrix)**.**

```
2 * matrixA
```

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

Dieses Prinzip lässt sich auf Vektoren und Matrizen verallgemeinern. Bei Vektoren muss der Vektor die gleiche Länge wie die Zeilen der Matrix haben. Der Vektor wird dann *spaltenweise* mit der Matrix verknüpft (Beispiel 13.17).

Beispiel 13.17 (Vektoraddition mit einer Matrix)**.**

```
matrixA + 1:2
```

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

> ⚠ Achtung!
>
> Hat der Vektor keine passende Länge, dann wird die Länge des Vektors durch Wiederholung an die Länge der Matrix angepasst. Die Transformation erfolgt in diesem Fall wie eine Vektortransformation, wobei die Zeilen- und Spaltenstruktur der Matrix erhalten bleibt (Beispiel 13.18).

Beispiel 13.18 (Vektoraddition mit einer Matrix und einem inkompatiblen Vektor)**.**

```
matrixA + 1:3
```

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

Für die Transformation einer Matrix mit einer anderen Matrix mit R *müssen* beide Matrizen die gleiche Dimensionalität haben. Ist diese Voraussetzung gegeben, dann werden die Werte mit dem gleichen Index paarweise miteinander verknüpft (Beispiel 13.19).

> ⚠ Achtung!
>
> Werden zwei Matrizen unterschiedlicher Dimensionalität verknüpft, dann erweitert R die kleinere der beiden Matrizen **nicht**!

Beispiel 13.19 (Verknüpfung von zwei Matrizen)**.**

```
matrixB = matrix(c(1, 2, 2, 1, 1, 2), nrow = 2)

matrixA + matrixB
```

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

13.6.1. Matrizen vergleichen

Um die Gleichheit zwischen zwei Matrizen zu vergleichen, müssen zwei Bedingungen erfüllt sein:

1. Die Länge der beiden Matrizen muss gleich sein.
2. Die Dimensionalität der beiden Matrizen muss gleich sein.
3. An allen Positionen müssen die Werte gleich sein.

Diese Prüfung übernimmt in R die Funktion `all.equal()`. Damit diese Funktion in logischen Ausdrücken verwendet werden kann, muss der Wert an der ersten Position des Ergebnisvektors ausgewertet werden, weil diese Funktion alle Unterschiede zwischen zwei Matrizen zurückgibt (Beispiel 13.20).

Beispiel 13.20 (Zwei Matrizen vergleichen)**.**

```
all.equal(matrixA, matrixB)
```

```
[1] "Attributes: < Length mismatch: comparison on first 1 components >"
[2] "Mean relative difference: 0.5555556"
```

```
# Für logische Ausdrücke
all.equal(matrixA, matrixB)[1] == TRUE
```

```
[1] FALSE
```

13.7. Kreuzprodukt

Das Kreuzprodukt (oder Matrixmultiplikation) ist eine Erweiterung der bekannten Multiplikation für Matrizen. Beim Kreuzprodukt muss die linke Matrix genausoviele Spalten haben, wie Zeilenanzahl der rechten Matrix. Das Kreuzprodukt ist dann wie folgt definiert:

$$A \times B = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{i=1}^{n} a_{1i} \cdot b_{i1} & \sum_{i=1}^{n} a_{1i} \cdot b_{i2} & \cdots & \sum_{i=1}^{n} a_{1i} \cdot b_{ip} \\ \sum_{i=1}^{n} a_{2i} \cdot b_{i1} & \sum_{i=1}^{n} a_{2i} \cdot b_{i2} & \cdots & \sum_{i=1}^{n} a_{2i} \cdot b_{ip} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} a_{mi} \cdot b_{i1} & \sum_{i=1}^{n} a_{mi} \cdot b_{i2} & \cdots & \sum_{i=1}^{n} a_{mi} \cdot b_{ip} \end{bmatrix}$$

In R wird das Kreuzprodukt mit dem `%*%`-Operator durchgeführt.

```
matrixA
```

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

```
(matrixC = matrix(
    c(3, 2, 2, 3,
      1, 2, 3, 1,
      1, 2, 1, 1),
    ncol = 4,
    byrow = TRUE)
)
```

```
     [,1] [,2] [,3] [,4]
[1,]    3    2    2    3
[2,]    1    2    3    1
[3,]    1    2    1    1
```

```
matrixA %*% matrixC
```

```
     [,1] [,2] [,3] [,4]
[1,]    8   12   11    8
[2,]   12   12   13   12
```

Praxis

Werden mehrere Matrizen nacheinander multipliziert, so können bekannte Matrizen vorab miteinander multipliziert werden, wenn sie in der Berechnung direkt aufeinanderfolgen.

13.7.1. Zeilen- und Spaltensummen

Für die Zeilensumme ist ein Spaltenvektor notwendig. Dieser Vektor ist ein Einsvektor mit der Länge von der Anzahl der **Matrixspalten**. Wegen der Regeln für das Kreuzprodukt muss der Einsvektor der rechte Operand des Kreuzprodukts sein (Beispiel 13.21).

Beispiel 13.21 (Zeilensumme)**.**

```
matrixA %*% rep(1, ncol(matrixA))
```

```
     [,1]
[1,]    6
[2,]    6
```

Für die Spaltensumme ist ein Zeilenvektor notwendig. Dieser Vektor ist ein Einsvektor mit der Länge von der Anzahl der **Matrixzeilen**. Weil R-Vektoren keine Orientierung haben, sondern immer als Spaltenvektoren behandelt werden, muss der Vektor noch transponiert werden. Bei der Zeilensumme ist ausserdem zu beachten, dass der Einsvektor der linke Operand des Kreuzprodukts sein muss (Beispiel 13.22).

Beispiel 13.22 (Zeilensumme)**.**

```
t(rep(1, nrow(matrixA))) %*% matrixA
```

```
     [,1] [,2] [,3]
[1,]    4    4    4
```

13.8. Äusseres Vektorprodukt

Das äussere Vektorprodukt erlaubt es Matrizen aus den Werte von zwei Vektoren mit einem *beliebigen Operators* zu erzeugen. Im Gegensatz zum Kreuzprodukt ist die Orientierung der beiden Vektoren durch das äussere Produkt vorgegeben: Der zweite Vektor wird immer als Zeilenvektor angenommen.

Der Operator des äusseren Produkts ist `%o%`, wobei als Verknüpfung die Multiplikation verwendet wird. Mit der Funktion `outer()` können beliebige Operatoren (d.h. Funktionen mit zwei Operatoren) angegeben werden.

Beispiel 13.23 (Äusseres Produkt mit der Addition als Operator)**.**

```
outer(1:3, 0:5, `+`)
```

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

In der Anwendung sind logische Ausdrücke als Operatoren für das äussere Produkt von besonderer Bedeutung. In R müssen alle logischen Ausdrücke, die nicht nur eine Vergleichsoperation umfassen, als Funktion der `outer()`-Funktion übergeben werden (s. Beispiel 13.25).

Praxis

Mit dem äusseren Produkt über logische Ausdrücle lassen sich beliebige Matrizenstrukturen aus Sequenzen erzeugen.

13.8.1. Dreieckmatrizen erzeugen

Dreiecksmatrizen werden in der Regel über das äussere Produkt zweier Sequenzen erzeugt. Es wird zwischen oberen und unteren Dreiecksmatrizen unterschieden. Bei einer oberen Dreiecksmatrix sind alle Positionen mit Zeilenindizes, welche kleiner als der jeweilige Spaltenindex sind, mit dem Wert `1` und alle anderen Position mit `0` belegt. Bei einer unteren Dreieckmatrix sind die Positionen mit dem Wert `1` belegt, an denen der Zeilenindex grösser als der Spaltenindex ist. Alternativ erzeugen die beiden Funktionen `upper.tri()` und `lower.tri()` die gewünschte Dreiecksmatrix aus einer existierenden Matrix.

Beispiel 13.24 (Erzeugen einer unteren Dreiecksmatrix)**.**

```
1 * outer(1:4, 1:4, `>=`)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    1    1    0    0
[3,]    1    1    1    0
[4,]    1    1    1    1
```

```
# etwas expressiver
1 * lower.tri(diag(4), diag = TRUE)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    1    1    0    0
[3,]    1    1    1    0
[4,]    1    1    1    1
```

13.8.2. Vorgänger- und Nachfolgersummen

Vorgänger- und Nachfolgersummen bilden die Summe aus aufeinanderfolgenden Werten eines Vektors. Diese Summe kann durch die Kombination einer Matrixmultiplikation mit dem äusseren Produkt errreicht werden. Für diese Summen wird eine Matrix benötigt, in welcher an allen zu summierenden Positionen der Wert `1` (das neutrale Element der Multiplikation) steht.

Beispiel 13.25 bestimmt die Summe vom Wert und den beiden Vorgängern einer bestimmten Position in einem Vektor. Die Subtraktion mit 2 zeigt an, wie viele Werte vor der aktuellen Position berücksichtigt werden sollen.

Beispiel 13.25 (Vorgängersumme für einen Vektor der Länge 10)**.**

```
vektor = 1:10
vektor %*% outer(
    1:10,
    1:10,
    function (a, b) (b >= a & a >= b - 2)
)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    3    6    9   12   15   18   21   24    27
```

13.9. Co-Occurence Matrizen

Eine Co-Occurence Matrix oder **Kontingenztabelle** aus zwei Vektoren lässt sich am leichtesten mit der `table()`-Funktion erstellen. Diese Funktion zählt das gemeinsame Auftreten von Werten in zwei *gleichlangen* Vektoren von beliebigen Datentyp.

Das Ergebnis ist eine *benannte Matrix*, wobei die Zeilen den Werten im ersten Vektor und die Spalten den Werten im zweiten Vektor entsprechen.

Beispiel 13.26 (Kontingenztabelle für zwei Vektoren)**.**

```
v1 = c(1, 2, 4, 3, 2, 1, 1, 2, 4)
v2 = c(9, 8, 7, 7, 7, 8, 9, 9, 8)

(ctable = table(v1, v2))
```

```
   v2
v1  7 8 9
  1 0 1 2
  2 1 1 1
  3 1 0 0
  4 1 1 0
```

```
ctable[,2]
```

```
1 2 3 4
1 1 0 1
```

```r
ctable[3,]
```

```
7 8 9
1 0 0
```

13.10. Determinanten

Die Determinante einer **quadratischen Matrix** A wird mit der `det()`-Funktion ermittelt.

Beispiel 13.27 (Determinante einer Matrix bestimmen)**.**

```r
# Eine quadratische Matrix aus matrixA erzeugen
(matrixQ = matrixA %*% t(matrixA))
```

```
     [,1] [,2]
[1,]   14   10
[2,]   10   14
```

```r
det(matrixQ)
```

```
[1] 96
```

13.11. Eigenwerte

Die Eigenwerte λ und Eigenvektoren einer **quadratischen Matrix** A werden mit der Funktion `eigen()` bestimmt. Diese Funktion gibt eine benannte Liste zurück. Unter `values` finden sich die Eigenwerte der Matrix und unter `vectors` sind die Eigenvektoren als *Spaltenvektoren* gespeichert.

Beispiel 13.28 (Eigenwerte und Eigenvektoren einer Matrix bestimmen)**.**

```r
basisQ = eigen(matrixQ)

basisQ[["values"]] # oder basisQ$values
```

```
[1] 24  4
```

```r
basisQ[["vectors"]] # oder basisQ$vectors
```

```
          [,1]       [,2]
[1,] 0.7071068 -0.7071068
[2,] 0.7071068  0.7071068
```

13.12. Inversematrix

Die Inverse A^{-1} einer **quadratischen Matrix** A wird in R mit der Funktion `solve()` bestimmt (Beispiel 13.29).

Beispiel 13.29 (Inverse Matrix berechnen)**.**

```r
solve(matrixQ)
```

```
           [,1]       [,2]
[1,]  0.1458333 -0.1041667
[2,] -0.1041667  0.1458333
```

13.13. `Matrix`-Bibliothek

Das Kreuzprodukt, das äussere Produkt und das Transponieren werden durch R automatisch bereitgestellt. Für komplexere Aufgaben oder sehr grossere Matrizen dient die Bibliothek `Matrix` (Bates et al., 2023). Diese Bibliothek gehört zur R-Standardinstallation und ist auf jedem System vorhanden. Die Bibliothek stellt alle relevanten Funktionen für Matrix-Operationen, die für mathematische Matrix-Operationen benötigt werden.

Diese Bibliothek sollte immer geladen werden, wenn sehr grosse Matrizen oder Matrizen mit vielen `0`-Werten verwendet werden. Besonders für den zweiten Fall ist die sog. **Sparse Matrix** (deutsch "dünnbesetzte Matrix") von besonderer Bedeutung. Eine dünnbesetzte Matrix ist eine Matrix mit vielen `0`-Werten, wobei *viel* "mehr als ein Drittel der Positionen" bedeutet. In solchen Matrizen lässt R die `0`-Werte weg, so dass diese Werte keinen Speicher benötigen und keine Berechnungen für die entsprechenden Operationen (insbesondere beim Kreuzprodukt) durchgeführt werden. Abgesehen von dieser speziellen Behandlung der `0`-Werte unterscheiden sich dünnbesetzte Matrizen in R nicht von anderen Matrizen.

Beispiel 13.30 (Dünnbesetzte Matrix erzeugen)**.**

```r
library(Matrix)
```

```
Attache Paket: 'Matrix'

Die folgenden Objekte sind maskiert von 'package:tidyr':

    expand, pack, unpack
```

```
Matrix( matrix(c(0, 1, 0, 0, 0, 1, 2, 0, 0), ncol = 3), sparse = TRUE)
```

```
3 x 3 sparse Matrix of class "dgCMatrix"

[1,] . . 2
[2,] 1 . .
[3,] . 1 .
```

Die `Matrix`-Bibloiothek stellt ausserdem viele Hilfreiche Funktionen für die Arbeit mit Matrizen bereit. Dazu gehören beispielsweise die Funktionen `colSums()` und `rowSums()`, mit denen die Spalten- bzw. die Zeilensummen berechnet werden. Beispiel 13.22 und Beispiel 13.21 lassen sich so vereinfachen (Beispiel 13.31).

Beispiel 13.31 (Zeilen- und Spaltensumme mit der `Matrix`-Bibliothek)**.**

```
colSums(matrixA)
```

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

```
rowSums(matrixA)
```

```
[1] 6 6
```

Die Bibliothek stellt ausserdem mit der `band()`-Funktion eine komfortable Methode für diagonale Band-Matrizen bereit. Eine Bandmatrix heisst eine quadratische Matrix, die ein diagonales Werteband enthält. Mit dieser Funktion lassen sich Band-Matrize für gleitende Summen (Beispiel 13.25) leichter erzeugen (Beispiel 13.32)

Beispiel 13.32 (Vorgängersumme mit der `band()`-)**.**

```
band(Matrix(1, 10, 10, sparse = TRUE), 0, 2)
```

```
10 x 10 sparse Matrix of class "dtCMatrix"

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

```
vektor %*% band(Matrix(1, 10, 10, sparse = TRUE), 0, 2)
```

```
1 x 10 Matrix of class "dgeMatrix"
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    3    6    9   12   15   18   21   24    27
```

14. Indizieren und Gruppieren

⚠ Work in Progress

14.1. Indizieren

Es werden drei Arten von Indizes unterschieden:

1. Der **Primärindex**, mit dem ein einzelner Datensatz eindeutig *identifiziert* werden kann.
2. **Fremdschlüssel** sind Sekundärindizes für *Querverweise* auf eine zweite Datenstruktur (eine sog. *Indextabelle* oder engl. *Lookup-Table*).
3. **Gruppenindizes** sind Sekundärindizes zur Identifikation von Datensätzen mit *gemeinsamen Eigenschaften*.

Weil ein Index Werte über einen Datensatz enthält, gehört ein Index zum jeweiligen Datensatz und wird über einen *Indexvektor* in einer Stichprobe abgebildet.

14.1.1. Hashing eines Primärindex

Die einfachste Technik zur eindeutigen Indizierung ist das ***Durchnummerieren*** der Datensätze einer Stichprobe. Bei dieser Technik wird jedem Datensatz eine Nummer zugewiesen. In R verwenden wir dazu die Funktion `row_number()`. Diese Funktion ist einer *Sequenz* vorzuziehen, weil diese Funktion auch bei leeren Stichproben fehlerfrei arbeitet.

⚠ Warnung

Intuitiv würde man beim Durchnummerieren an seine Sequenz von `1:n()` denken. Diese Sequenz führt aber für leere Datenrahmen zu Fehlermeldungen. Die Funktion `row_number()` kann mit leeren Datenrahmen umgehen und erzeugt dieses Problem nicht.

Beispiel 14.1 (Primärindex erzeugen)**.**

```
mtcars |>
    as_tibble(rownames = "modell") ->
        mtcars_df

mtcars_df |>
    mutate(
        nr = row_number()
    ) ->
        mtcars_df_nbr

# Nummerierung zeigen
mtcars_df_nbr |>
    head()
```

```
# A tibble: 6 x 13
  modell   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb    nr
  <chr>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <int>
1 Mazda~  21       6   160   110  3.9   2.62  16.5     0     1     4     4     1
2 Mazda~  21       6   160   110  3.9   2.88  17.0     0     1     4     4     2
3 Datsu~  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1     3
4 Horne~  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1     4
5 Horne~  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2     5
6 Valia~  18.1     6   225   105  2.76  3.46  20.2     1     0     3     1     6
```

14.1.2. Hashing zum Gruppieren

Beim Hashing zum Gruppieren müssen wir Werte erzeugen, die eine Zuordnung zu einer Gruppe oder einen Wert in einer anderen Stichprobe ermöglichen. Die Hashing-Funktion orientiert sich dabei an den konkreten Analyseanforderungen.

Vier gängige Techniken können dabei unterschieden werden:

- Kodieren (alle Datentypen)
- Reihenfolgen bilden durch Ganzzahldivision (nur Zahlen)
- Reihenfolgen bilden durch Modulo-Operation (nur Zahlen)
- Reihenfolgen durch Anfangsbuchstaben (nur Zeichenketten)

14.1.2.1. Beispiel Hashing zum Gruppieren.

Das folgende Beispiel bildet einen Index, um die Motorisierung der Fahrzeugtypen in der Stichprobe `mtcars` zu bestimmen. Dabei sollen die Modelle in schwach-, mittel-, stark- und sehr starkmotorisierte Typen unterschieden werden. Die Motorisierung richtet sich dabei zum einen nach der Leistung (`hp`). Zum anderen richtet sich die Motorisierung nach dem Fahrzeuggewicht (`wt`), weil für ein schweres Fahrzeug mehr Leistung zum Bewegen benötigt

wird als für ein leichtes. Um beide Werte zu berücksichtigen, wird das Verhältnis der beiden Werte bestimmt. Ein Verhältnis ist eine *Division*. In diesem Fall wird das Gewicht als Nenner verwendet und die Leistung als Zähler. So ergeben sich immer Werte grösser als 1, weil die Leistung immer viel grösser als das Gewicht ist.

In diesem Beispiel besteht die Hashing-Funktion aus zwei Teilen:

1. Das Verhältnis zwischen Leistung und Gewicht wird bestimmt und im Vektor `verhaeltnis` abgelegt.
2. Die Leistungsklassen werden durch *Kodieren* den oben festgelegten Klassen zugewiesen und im Vektor `klasse` gespeichert.

```
mtcars_df |>
    mutate(
        verhaeltnis = hp/wt,
        klasse = case_when(
            verhaeltnis > 60 ~ "sehr stark",
            verhaeltnis > 50 ~ "stark",
            verhaeltnis > 40 ~ "mittel",
            TRUE ~ "schwach")
    ) |>
    # nur die relevanten Vektoren zeigen
    select(modell, hp, wt, verhaeltnis, klasse)
```

```
# A tibble: 32 x 5
   modell              hp    wt verhaeltnis klasse
   <chr>            <dbl> <dbl>       <dbl> <chr>
 1 Mazda RX4          110  2.62        42.0 mittel
 2 Mazda RX4 Wag      110  2.88        38.3 schwach
 3 Datsun 710          93  2.32        40.1 mittel
 4 Hornet 4 Drive     110  3.22        34.2 schwach
 5 Hornet Sportabout  175  3.44        50.9 stark
 6 Valiant            105  3.46        30.3 schwach
 7 Duster 360         245  3.57        68.6 sehr stark
 8 Merc 240D           62  3.19        19.4 schwach
 9 Merc 230            95  3.15        30.2 schwach
10 Merc 280           123  3.44        35.8 schwach
# i 22 more rows
```

14.1.3. Hashing für Querverweise

Beim Hashing für Querverweise gibt es zwei Stichproben. Die erste Stichprobe ist die Hauptstichprobe mit den eigentlichen Werten. Die zweite Stichprobe ist die Referenzstichprobe, die zusätzliche Information enthält. Ein Indexvektor für Querverweise in der ersten Stichprobe bezieht sich immer auf einen Primärindex aus der zweiten Stichprobe.

Die Hashing-Funktion muss deshalb einen Verweis zur zweiten Stichprobe herstellen. Diese Verbindung kann mit der gleichen Strategie erzeugt werden, wie beim Gruppieren. Dabei muss jedoch darauf geachtet werden, dass alle Zuordnungen des Primärvektors korrekt abgebildet sind.

14.2. Randomisieren

Wenn wir mit Teilstichproben arbeiten und diese mit anderen teilen, müssen wir vermeiden, dass zwei Stichproben leicht zusammengesetzt werden können und so Rückschlüsse über die Probanden möglich werden.

> **!** Achtung
>
> Sobald personenbezogene Daten statistisch ausgewertet und zur Publikation vorbereitet werden, **müssen** die Daten randomisiert werden!

Dieses Rezept beschreibt eine Technik zur Anonymisierung von Daten durch Mischen. Entscheidend bei dieser Technik ist, dass wir die Werte für unsere Analyse zusammenhalten möchten, sodass unsere Ergebnisse nachvollziehbar bleiben. Gleichzeitig soll es unmöglich werden, diese Daten mit anderen Teilen unserer Studien in Verbindung zu bringen.

Die Technik der Anonymisierung durch Mischen besteht aus vier Schritten:

1. Auswahl der Vektoren, die wir in einer Publikation teilen möchten,
2. Erzeugung eines eindeutigen Vektors,
3. zufälliges Mischen,
4. Entfernen der eindeutigen Vektoren und exportieren der Daten.

14.2.1. Schritt 1: Auswahl der Vektoren

Die zu veröffentlichenden Vektoren werde mit der `select()`-Funktion selektiert.

14.2.2. Schritt 2: Erzeugung eines eindeutigen Vektors

Alle Werte müssen zusammengehalten werden,damit nachgereihte Analysen nachvollziehbar bleiben. Dazu werden die Datensätze nummeriert.

14.2.3. Schritt 3: Mischen

Dieser Schritt greift auf die Funktion `sample()` zurück. Wir erzeugen aus den ursprünglichen Nummerierungen eine neue Nummerierung durch `daten |> mutate( id_neu = sample(id) )`. Nach dieser neuen Nummerierung sind unsere Datensätze aber immer noch in der gleichen Reihenfolge und noch nicht gemischt. Wir müssen also die Reihenfolge so

anpassen, dass die neue Nummerierung gilt. Das erreichen wir mit dem Funktionsaufruf `daten |> arrange( id_neu )`.

14.2.4. Schritt 4: Entfernen des eindeutigen Vektors und exportieren der Daten

Abschliessend müssen wir **unbedingt** die beiden Hilfsvektoren, die wir zum Mischen verwendet haben, aus unserer Stichprobe wieder entfernen. Das erreichen wir mit einer Vektorauswahl: `daten |> select(-c(id, id_neu))`.

14.2.5. Vollständige Lösung

Wir greifen hier auf eine Stichprobe zurück, die GeschlechtsInformation, Alter und digitale Nutzungsgewohnheiten umfasst. Wir erstellen zwei getrennte Teilstichproben, von denen eine nur die Nutzungsgewohnheiten und das Geschlecht und eine nur die Nutzungsgewohnheiten und das Alter beinhaltet.

```
daten = read_csv(
    "daten/befragung_digitales_umfeld/befragung.csv",
    show_col_types = F)

# mischen Funktion aus einer Funktionskette erstellen, damit
#    wir nicht so viel tippen müssen.
mischen = . %>%
    mutate(
        id = row_number(),
        id_neu = sample(id) # wählt zufällig eine id aus.
    ) %>%
    arrange(id_neu) %>%
    select(-c(id, id_neu))

daten |>
    select(q00_2, starts_with("q15")) |>
    mischen() |>
    write_csv("teilstichprobe_geschlecht_technik.csv")

daten |>
    select(kurs, starts_with("q18")) |>
    mischen() |>
    write_csv("teilstichprobe_alter_sozial.csv")
```

> **Hinweis**
>
> Die Definition der Funktion `mischen()` verwendet die spezielle Funktionsverkettung von `tidyverse`, weil damit Funktionen erstellt werden können. In diesem Fall dürfen die Operatoren `|>` und `%>%` nicht im gleichen Arbeitsschritt gemischt werden.

Die Daten in den beiden Dateien lassen sich nicht mehr zusammenführen. Damit erkennen wir auch die Grenzen dieser Technik: Wenn zwei gemischte Stichproben ausreichend viele gemeinsame oder sehr detaillierte Vektoren haben, die in beiden Teilstichproben vorkommen, dann können diese Stichproben trotz mischen wieder zusammengeführt werden.

Mit den gemischten Daten ist es nun nicht mehr möglich, die Werte mit einem anderen Teil der Stichprobe zu kombinieren und so tiefere Rückschlüsse über die Teilnehmenden (womöglich unwissend) zuzulassen. Nur noch durch den Zugriff auf die ursprünglichen Daten können diese Zusammenhänge hergestellt werden. Daher sind die ursprünglichen Daten oft besonders schützenswert und sollten ohne Randomisierung nicht weitergegeben werden.

14.3. Gruppieren

In R lassen sich grundsätzlich alle Opeationen auch als gruppierte Operationen über einen Datenrahmen durchführen. Dazu wird die `group_by()`-Funktion verwendet, um gruppierte Operationen anzuzeigen. Dieser Funktion können ein oder mehrere Sekundärindizes als Parameter übergeben werden, die zum Gruppieren verwendet werden.

Beispiel 14.2 (Gruppierte Summe und Mittelwert)**.**

```
daten |>
    group_by(index) |>
    summarise(
        summe = sum(werte)
        mittelwert = mean(werte)
    ) |>
    ungroup()
```

> **Achtung**
>
> Wird eine Funktion `group_by()` ohne eine nachfolgende Operation ausgeführt, dann hat die Funktionen **keinen** erkennbaren Effekt auf das Ergebnis.

> **Praxis**
>
> Nach eine gruppierten Operation sollte die Gruppierung **immer** mit `ungroup()` aufgehoben werden, damit nachfolgende nichtgruppierte Operationen nicht versehentlich als gruppierte Operationen durchgeführt werden.

14.3.1. Gruppiertes Zählen

R verfügt mit der Funktion `table()` das gemeinsame Auftreten von Werten in zwei diskretskalierten Vektoren zu zählen. Diese Funktion erfordert jedoch zwei voneinander unabhängige Vektoren. Die Funktion `count()` bietet eine elegante Alternative für Vektoren in Datenrahmen. In diesem Fall werden die Vektoren als Sekundärindizes behandelt.

Beispiel 14.3 (Gruppiertes Zählen)**.**

```
daten |>
    count(index)
```

Das gruppierte Zählen hat gegenüber der Funktion `table()` den Vorteil, dass mehr als zwei Vektoren berücksichtigt werden können.

14.3.2. Gruppiertes Nummerieren

Nummerieren ist eine hilfreiche Funktion, um die Position von Werten zu bestimmen. Das Ergebnis ist ein eindeutige Nummer für jeden Datensatz. Beim Gruppierten Nummerieren werden die Datensätze innerhalb ihrer Gruppe nummeriert.

Beispiel 14.4 (Gruppiertes Nummerieren)**.**

```
daten |>
    group_by(index) |>
    mutate(
        nr = row_number()
    ) |>
    ungroup()
```

15. Daten kombinieren und kodieren

⚠ Work in Progress

Das Kombinieren und kodieren von Daten haben in R eine enge Beziehung. Die notwendigen Funktionen werden durch die `tidyverse`-Bibliothek `dplyr` (Wickham, François, et al., 2023) bereitgestellt.

15.1. Kombinieren

15.1.1. Konkatenation

Die Bibliothek `dplyr` stellt zwei Funktionen zur Konkatenation von Datenrahmen bereit:

1. Die Funktion `bind_rows()` und
2. Die Funktion `bind_cols()`

Die Funktion `bind_rows()` kombiniert zwei Datenrahmen, indem alle Vektoren der beiden Datenrahmen konkateniert werden. Dabei muss beachtet werden, dass alle zu konkatenierenden Vektoren in beiden Datenrahmen *gleich benannt* sind.

> **Merke**
>
> Die Funktion `bind_rows()` sollte nur dann eingesetzt werden, wenn eine normale Vereinigung mithilfe von Indizes nicht möglich ist.

Oft soll nach dem Konkatenieren der Datensätze die ursprünglichen Datenrahmen nachvollziehbar bleiben. Die Funktion `bind_row()` bietet hierzu den optionalen Parameter `.id` an. Dieser Parameter erwartet einen *Vektornamen*. Ist dieser Parameter gesetzt, dann erzeugt die Funktion eine Sequenz für die Datenrahmen und ergänzt alle Datensätze um den entsprechenden Wert.

Sollen andere Werte zur Identifikation der Datenrahmen verwendet werden, so müssen die Datenrahmen als benannte Liste übergeben werden ***und*** der Parameter `.id` muss gesetzt sein.

Die Funktion `bind_cols()` kombiniert zwei Datenrahmen, indem alle Datensätze der beiden Datenrahmen konkateniert werden. Dabei muss beachtet werden, dass die erzeugten Datensätze nach dem Konkatenieren auch zusammengehörende Werte umgfassen. Dazu müssen die Datensätze in beiden Datenrahmen tatsächlich in der gleichen Reihenfolge vorliegen.

Sollten die konkatenierten Datenrahmen nicht genau passen, weil entweder unterschiedlich benannte Vektoren oder unterschiedliche viele Datensätze in den beiden Datenrahmen existieren, dann erzeugt für die jeweils fehlenden Werte `NA`-Werte. Das ist oft unerwünscht. Deshalb sollte *vor* der Verwendung von `bind_rows()` oder `bind_cols()` geprüft werden, ob die Datenrahmen für die konkatenation geeignet sind.

15.1.2. Vereinigung

R kennt neben der Spaltenkonkatenation mit `bind_rows()` noch drei weitere Formen der Vereinigung:

1. Die vollständige Vereinigung (`full_join()`)
2. Die linksseitige Vereinigung (`left_join()`)
3. Die rechtsseitige Vereinigung (`right_join()`)

Diese Funktionen kommen immer dann zum Einsatz, wenn die Datensätze in den Datenrahmen nicht in der gleichen Reihenfolge vorliegen und die Verknüpfung über einen Index erfolgen soll. Dieser Index muss in **beiden** Datenrahmen existieren.

> **!** Achtung
>
> Wird bei der vollständigen Vereinigung mit `full_join()` kein Index für die Kombination angegeben, dann erzeugt R für das Ergebnis alle Permutationen der Datensätze in beiden Datenrahmen. Diese Operation ist sehr speicherhungrig und kann selbst bei Datenrahmen mit vergleichsweise wenig Datensätzen zu unerwarteten Programmabstürzen führen, wenn das Ergebnis nicht mehr in den Arbeitsspeicher des Rechners passt.

> **i** Merke
>
> Die Funktionen `left_join()` und `right_join()` erlauben die ***partielle Vereinigung*** der beiden Datenrahmen.

15.1.3. Schnittmenge

Die Schnittmenge bzw. *inner join* zweier Datenrahmen wird in R mit der Funktion `inner_join()` gebildet. Das Ergebnis der Schnittmenge enthält Datensätze, die für den verknüpfenden Index eine Entsprechung in *beiden* Datensätzen haben. Alle anderen Datensätze werden entfernt.

15.1.4. Differenz

Gelegentlich sollen vorgegebene Werte entfernt werden. Dazu kommt eine spezielle *Kodierungstabelle* zum Einsatz. Diese Tabelle legt fest, welche Werte aus den Daten entfernt oder ausgeschlossen werden sollen. Entsprechend heisst diese Tabelle eine **Ausschlusstabelle**.

In R lassen sich Ausschlusstabellen mit der Funktion `anti_join()` verwenden.

15.2. Kodierungstabellen

Eine Kodierungstabelle ist ein Datenrahmen, mit der Werte in andere Werte überführt werden. In der Regel werden Kodierungstabellen für die Neukodierung von diskreten Daten verwendet. Eine Kodierungstabelle enthält dazu den ursprünglichen zu kodierenden Wert und die jeweiligige neue Kodierung.

> **i** Merke
>
> Die zu kodierenden Werte dürfen in einer Kodierungstabelle genau **einmal** vorkommen. Deshalb sind die zu kodierenden Werte in der Kodierungstabelle ein **Primärindex**.

15.3. Kodierung durch Kombination

Der einfachste Weg, um Werte in einem Datenrahmen in R mithilfe einer Kodierungstabelle neu zu kodieren, ist die Kombination der Daten mit einer Kodierungstabelle. Dabei wird der zu kodierende Vektor als Sekundärindex verwendet, der mit dem Primärindex der Kodierungstabelle über die Schnittmenge mithilfe der Funktion `inner_join()` kombiniert wird. Durch diese Kombination werden gleichzeitig alle ungültigen Werte (inkl. `NA`) entfernt.

15.4. Mit Faktoren kodieren

Ordinalskalierte Daten werden sehr oft als Zahlen kodiert. In R ist für diese Aufgabe *keine* Kodierungstabelle notwendig. Stattdessen werden die Werte als Faktor (Kapitel 10) behandelt, für den die Reihenfolge des Wertebereichs mit den geeigneten Funktionen der Bibliothek `forcats` (Wickham, 2023a) festgelegt wird. Für die Kodierung in Zahlen wird ausgenutzt, dass R die Faktorstufen intern bereits Zahlenwerten zuweist. Diese Zuweisung verwendet die Funktion `as.numeric()`, um aus einem Faktor numerische Werte abzuleiten.

16. Daten formen

> ⚠ Work in Progress

Die Funktionen zum Formen von Datenrahmen werden durch die `tidyverse`-Bibliothek `tidyr` bereitgestellt.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.2     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.2     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to be
```

16.1. Transponieren

Beim Transponieren von Datenrahmen werden die Spalten und Zeilen von Merkmalen neu ausgerichtet.

Das Transponieren von Datenrahmen ist keine vollständige Operation, sondern unterliegt je nach Ausgangsorientierung unterschiedlichen Anforderungen.

Die Funktion `pivot_longer()` überführt Vektoren von der Matrixform (Breitform) in die Vektorform (Langform). Alle der transponierten Vektoren müssen dafür vom **gleichen Datentyp** sein. Beim Transponieren in die Vektorform werden immer zwei Vektoren erzeugt:

- Der Wertevektor, der die Werte der ursprünglichen Vektoren aufnimmt und
- Der Namenvektor für die ursprünglichen Vektorennamen.

Falls keine anderen Angaben gemacht werden, heisst der Wertevektor `value` und der Namenvektor `name`.

> **Merke**
>
> Der Namenvektor ist gleichzeitig ein **Sekundärindex**.

Fall nicht alle Vektoren transponiert werden, dann werden die nicht transponierten Teile der Datensätze auf alle Datensätze der transponierten Vektoren erweitert.

> **Merke**
>
> Ein Primärindex wird nach dem Transponieren zum Sekundärindex.

Beispiel 16.1 (Transponieren in die Vektorform)**.**

```
# Ursprünglich Daten
(schweizerStaedte =
    read_csv("geschlechter_schweizer_staedte.csv") |>
    select(-c(S, N)))
```

```
Rows: 10 Columns: 8
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): Ort
dbl (7): Gesamt, S_M, S_F, N_M, N_F, S, N

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.
```

```
# A tibble: 10 x 6
   Ort         Gesamt    S_M    S_F   N_M   N_F
   <chr>        <dbl>  <dbl>  <dbl> <dbl> <dbl>
 1 Winterthur  118681  57832  59074   817   958
 2 Zürich      435319 214943 212778  3858  3740
 3 Biel/Bienne  55600  27243  27827   232   298
 4 Bern        137529  65215  69291  1580  1443
 5 Luzern       85840  40576  43264   837  1163
 6 Basel       178270  84896  88656  2586  2132
 7 St. Gallen   78109  38303  38628   529   649
 8 Lugano       63796  30116  32348   548   784
 9 Lausanne    145524  68516  72902  2119  1987
10 Genève      207630  97990 105850  1496  2294
```

```
# Transponieren in die Vektorform
(schweizerStaedte |>
    pivot_longer(- c(Ort, Gesamt)) ->
        schweizerStaedteLang)
```

```
# A tibble: 40 x 4
   Ort         Gesamt name   value
   <chr>        <dbl> <chr>  <dbl>
 1 Winterthur  118681 S_M    57832
 2 Winterthur  118681 S_F    59074
 3 Winterthur  118681 N_M      817
 4 Winterthur  118681 N_F      958
 5 Zürich      435319 S_M   214943
 6 Zürich      435319 S_F   212778
 7 Zürich      435319 N_M     3858
 8 Zürich      435319 N_F     3740
 9 Biel/Bienne  55600 S_M    27243
10 Biel/Bienne  55600 S_F    27827
# i 30 more rows
```

Die Funktion `pivot_wider()` transponiert einen Datenrahmen von der Vektorform (Langform) in die Matrixform (Breitform). Damit diese Operation durchführbar ist, muss neben dem Wertevektor ein Sekundärindex für die Vektorennamen **und** ein Sekundärindex für die *zeilenweise Zuordnung* der Werte vorliegen. Dieser Index wird von R verwendet, um die Vektorennamen zu erzeugen: Die Operation erzeugt für jeden diskreten Wert im Sekundärindex einen Vektor für die Matrixform. Der zweite Sekundärindex wird nach dem Transponieren zum Primärindex.

Beispiel 16.2 (Transponieren in die Matrixform)**.**

```
# Daten transponieren
schweizerStaedteLang |>
    select(-Gesamt) |>
    pivot_wider(names_from = Ort)
```

```
# A tibble: 4 x 11
  name  Winterthur Zürich `Biel/Bienne`  Bern Luzern Basel `St. Gallen` Lugano
  <chr>      <dbl>  <dbl>         <dbl> <dbl>  <dbl> <dbl>        <dbl>  <dbl>
1 S_M        57832 214943         27243 65215  40576 84896        38303  30116
2 S_F        59074 212778         27827 69291  43264 88656        38628  32348
3 N_M          817   3858           232  1580    837  2586          529    548
4 N_F          958   3740           298  1443   1163  2132          649    784
# i 2 more variables: Lausanne <dbl>, Genève <dbl>
```

Existieren weitere Vektoren im Datenrahmen, dann werden diese beim Transponieren zusammengefasst, so dass die Werte nach dem Transponieren nur noch einmal im Ergebnis erscheinen.

> **Achtung**
>
> Existiert kein zweiter Sekundärindex beim Transponieren in die Breitform, dann fasst R alle Datensätze zusammen, bei denen alle Werte in den zusätzlichen Vektoren gleich sind. Dieses Verhalten **kann** dazuführen, dass mehr Werte transponiert werden müssen als Zieldatensätze erzeugt werden können. In solchen Fällen werden die überzähligen Werte als Listenvektor abgelegt (Beispiel 16.3). Dieses Verhalten ist normalerweise *unerwünscht*.

Beispiel 16.3 (Transponieren mehrdeutiger Werte in die Matrixform)**.**

```
# Ursprüngliche Daten einlesen
(abDaten = read_csv2("data_ab_semi.csv"))
```

```
i Using "','" as decimal and "'.'" as grouping mark. Use `read_delim()` for more control.

Rows: 136 Columns: 4
-- Column specification --------------------------------------------------------
Delimiter: ";"
chr (1): Angebot
dbl (3): Punkte, Interesse, Bedeutung

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.

# A tibble: 136 x 4
   Punkte Angebot Interesse Bedeutung
    <dbl> <chr>       <dbl>     <dbl>
 1   207. B               5         5
 2   101. A               2         3
 3   312. A               5         5
 4   242. B               3         2
 5   256. A               2         2
 6   188. B               6         2
 7   343. B               3         6
 8   202. A               5         5
 9   175. A               3         3
10   213. B               3         2
# i 126 more rows
```

```
# Daten transponieren
abDaten |>
    pivot_wider(names_from = Angebot, values_from = Punkte)
```

```
Warning: Values from `Punkte` are not uniquely identified; output will contain
list-cols.
* Use `values_fn = list` to suppress this warning.
* Use `values_fn = {summary_fun}` to summarise duplicates.
* Use the following dplyr code to identify duplicates.
  {data} %>%
  dplyr::group_by(Interesse, Bedeutung, Angebot) %>%
  dplyr::summarise(n = dplyr::n(), .groups = "drop") %>%
  dplyr::filter(n > 1L)


# A tibble: 31 x 4
   Interesse Bedeutung B         A
       <dbl>     <dbl> <list>    <list>
 1         5         5 <dbl [4]> <dbl [4]>
 2         2         3 <dbl [4]> <dbl [4]>
 3         3         2 <dbl [6]> <dbl [2]>
 4         2         2 <dbl [1]> <dbl [6]>
 5         6         2 <dbl [2]> <NULL>
 6         3         6 <dbl [4]> <dbl [3]>
 7         3         3 <dbl [8]> <dbl [2]>
 8         2         4 <dbl [2]> <dbl [2]>
 9         4         2 <dbl [2]> <dbl [2]>
10         5         6 <dbl [1]> <dbl [1]>
# i 21 more rows
```

16.1.1. Rezept: Operationen zusammenfassen

16.1.1.1. Problem

Es sollen mehrere Vektoren eines Datenrahmens auf die gleiche Art transformiert oder aggregiert werden.

16.1.1.2. Lösung

Die Vektoren werden zuerst in die Vektorform (Langform) *transponiert*. Anschliessend wird eine (gruppierte) Transformation oder gruppierte Aggregation durchgeführt.

Beispiel 16.4 (Zusammengefasstes Aggregieren)**.**

```
schweizerStaedte |>
    pivot_longer(-c(Ort, Gesamt)) |>
    group_by(Ort, Gesamt) |>
    summarise(
```

```
        sum(value)
    )
```

```
`summarise()` has grouped output by 'Ort'. You can override using the `.groups`
argument.
```

```
# A tibble: 10 x 3
# Groups:   Ort [10]
   Ort          Gesamt `sum(value)`
   <chr>         <dbl>        <dbl>
 1 Basel        178270       178270
 2 Bern         137529       137529
 3 Biel/Bienne   55600        55600
 4 Genève       207630       207630
 5 Lausanne     145524       145524
 6 Lugano        63796        63796
 7 Luzern        85840        85840
 8 St. Gallen    78109        78109
 9 Winterthur   118681       118681
10 Zürich       435319       435319
```

16.1.1.3. Erklärung

Durch das transponieren entstehen ein Namenvektor und ein Wertevektor, wobei der Namenvektor auch als Sekundärindex behandelt werden kann. Sich wiederholende Operationen lassen sich so in einer gruppierten Operation zusammenfassen. Durch das Transponieren lässt sich die Code-Komplexität reduzieren, wodurch die Lösung insgesamt weniger fehleranfällig wird.

Diese Logik lässt sich auf alle Datenoperationen anwenden. Im Beispiel 16.6 werden die Anteile am Gesamtvolumen von vier Merkmalen berechnet. Ohne das Transponieren würden repetitive Codeteile entstehen (Beispiel 16.5). Die sich wiederholenden Codeteile entfallen durch das Transponieren und können in einer Operation behandelt werden.

Beispiel 16.5 (Bestimmung des Anteils mehrerer Merkmale ohne Transponieren)**.**

```
schweizerStaedte |>
    mutate(
        Anteil_S_M = S_M / Gesamt,
        Anteil_S_F = S_F / Gesamt,
        Anteil_N_M = N_M / Gesamt,
        Anteil_N_F = N_F / Gesamt
    ) |>
    select(c(Ort, starts_with("Anteil")))
```

```
# A tibble: 10 x 5
   Ort         Anteil_S_M Anteil_S_F Anteil_N_M Anteil_N_F
   <chr>            <dbl>      <dbl>      <dbl>      <dbl>
 1 Winterthur       0.487      0.498    0.00688    0.00807
 2 Zürich           0.494      0.489    0.00886    0.00859
 3 Biel/Bienne      0.490      0.500    0.00417    0.00536
 4 Bern             0.474      0.504    0.0115     0.0105
 5 Luzern           0.473      0.504    0.00975    0.0135
 6 Basel            0.476      0.497    0.0145     0.0120
 7 St. Gallen       0.490      0.495    0.00677    0.00831
 8 Lugano           0.472      0.507    0.00859    0.0123
 9 Lausanne         0.471      0.501    0.0146     0.0137
10 Genève           0.472      0.510    0.00721    0.0110
```

Beispiel 16.6 (Bestimmung des Anteils mehrerer Merkmale)**.**

```
schweizerStaedte |>
    pivot_longer(- c(Ort, Gesamt)) |>
    mutate(
        value = value / Gesamt
    ) |>
    # Anpassen des Ergebnisses zur Präsentation
    select(- Gesamt) |>
    pivot_wider(names_prefix = "Anteil_")
```

```
# A tibble: 10 x 5
   Ort         Anteil_S_M Anteil_S_F Anteil_N_M Anteil_N_F
   <chr>            <dbl>      <dbl>      <dbl>      <dbl>
 1 Winterthur       0.487      0.498    0.00688    0.00807
 2 Zürich           0.494      0.489    0.00886    0.00859
 3 Biel/Bienne      0.490      0.500    0.00417    0.00536
 4 Bern             0.474      0.504    0.0115     0.0105
 5 Luzern           0.473      0.504    0.00975    0.0135
 6 Basel            0.476      0.497    0.0145     0.0120
 7 St. Gallen       0.490      0.495    0.00677    0.00831
 8 Lugano           0.472      0.507    0.00859    0.0123
 9 Lausanne         0.471      0.501    0.0146     0.0137
10 Genève           0.472      0.510    0.00721    0.0110
```

16.1.2. Rezept: Gruppiertes Zählen schöner präsentieren

16.1.2.1. Problem

Beim gruppierten Zählen werden die Sekundärindizes und die Ergebnisse nebeneinander dargestellt, so dass sich die Ergebnisse nur schwer vergleichen lassen.

16.1.2.2. Lösung

Nach dem gruppierten Zählen wird das Ergebnis in die Matrixform transponiert.

Beispiel 16.7 (Lesbarere Präsentation des gruppierten Zählen)**.**

```
daten = read_csv2("data_ab_semi.csv")
```

```
i Using "',' as decimal and "'.'" as grouping mark. Use `read_delim()` for more control.
```

```
Rows: 136 Columns: 4
-- Column specification --------------------------------------------------------
Delimiter: ";"
chr (1): Angebot
dbl (3): Punkte, Interesse, Bedeutung

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.
```

```
daten |>
    count(Angebot, Interesse) |>
    pivot_wider(
        names_from = Angebot,
        values_from = n,
        values_fill = 0
    )
```

```
# A tibble: 7 x 3
  Interesse     A     B
      <dbl> <int> <int>
1         1     3     2
2         2    13    11
3         3    13    25
4         4    17    22
5         5     9    10
6         6     4     6
7         7     0     1
```

16.1.2.3. Erklärung

Werden beim Zählen über einen Datenrahmen mehrere Sekundärindizes verwendet, ist das Ergebnis nur schwer interpretierbar. Damit das Ergebnis für Menschen leichter zu lesen ist, muss die spaltenweise Darstellung in ein eleganteres Format überführt werden. Mit

der Funktion `pivot_wider()` lassen sich die Daten in eine Kreuztabelle überführen. Das Ergebnis ähnelt nach dem Transponieren dem Ergebnis der `table()`-Funktion. Auf diese Weise lassen sich Häufungen in den Werten leichter erkennen.

In der Lösung werden drei besondere Parameter eingesetzt:

1. Der Parameter `names_from` zeigt an, aus welchem Index die neuen Vektornamen erzeugt werden sollen.
2. Der Parameter `values_from` zeigt an, in welchem Vektor die zu transponierenden Werte stehen.
3. der Parameter `values_fill` kommt zur Anwendung, wenn für eine Position kein Wert bereitsteht. Für die Zählung bedeutet das, dass nichts gezählt wurde. Deshalb wurde der Wert im Beispiel auf den Wert `0` gesetzt.

16.2. Hierarchisieren

Beim Hierarchisieren werden ausgewählte Vektoren eines Datenrahmens entlang eines Sekundärinzes zu separaten Datenrahmen organisiert und in den ursprünglichen Datenrahmen eingebettet. Über solche eingebettete Datenrahmen lassen sich hierarchische Datenstrukturen erzeugen, die beispielsweise in einem JSON- oder YAML-Dokument (s. Kapitel 7) gespeichert werden können. In R übernimmt diese Aufgabe die Funktion `nest()`.

Weil R jedoch keine Vektoren mit Datenrahmen erlaubt, müssen diese Datenrahmen zusätzlich in eine Liste geschachtelt werden. Diese Liste dient nur zur Ablage in einem Vektor und besteht immer nur aus einem Element, nämlich dem eingebetteten Datenrahmen.

Mit der Funktion `unnest()` lassen sich eingebettete Datenrahmen wieder ausbetten. Diese Operation funktioniert jedoch nur dann zuverlässig, wenn alle eingebetteten Datenrahmen einheitliche Vektoren haben. Ist diese Bedingung nicht gegeben, dann erzeugt R zusätzliche Vektoren mit `NA` für alle Datensätze, die die Vektoren ursprünglich nicht enthielten. Die Funktion `unnest()` führt mehrere Spaltenkonkatenationen aus und ähnelt damit im Ergebnis der Funktion `bind_rows()`.

Die beiden Funktionen `nest()` und `unnest()` sind speziell zur Erzeugung bzw. zum Auflösen von eingebetteten Datenrahmen gedacht. Gelegentlich enthält ein Datenrahmen einen Vektor mit einfachen Listen oder benannten Listen. Solche Vektoren sind vom Typ `list` und sollten mit den Funktionn `unnest_longer()` für einfache Listen sowie `unnest_wider()` für einheitlich benannte Listen (Wickham, Vaughan, et al., 2023).

16.3. Transponieren mit Zeichenketten

Ein spezieller Fall ist gegeben, wenn die Daten in Zeichenketten kodiert sind. Streng genommen handelt es sich hierbei nicht um eine Transponierenoperation, sondern um eine normale Transformation. Die entsprechenden Funktionen für diese Operation wurden aber nach dem gleichen Prinzip wie beim Hierarchisieren bzw. beim Transponieren angepasst.

Wie beim Transponieren kann ein Vektor in die Lang- oder die Breitform getrennt werden. Für die meisten Anwendungen eignen sich zwei Funktionen:

- `separate_longer_delim()`
- `separate_wider_delim()`

Gelegentlich sind die Daten in einer Zeichenkette nicht über ein eindeutiges Trennzeichen, sondern über ein Trennmuster kodiert. In diesem Fall kann das Trennmuster als *regulärer Ausdruck* der Funktion `separate_longer_regex()` bzw. `separate_wider_regex()` übergeben werden.

Für festkodierte Werte stehen die beiden Funktionen `separate_longer_position()` und `separate_wider_position()` zur Verfügung. Diese Funktionen erwarten einen Vektor mit den Feldbreiten, um die Werte trennen zu können.

Die Umkehrfunktion für alle `separate_`-Funktionen ist die Funktion `unite()`, mit der sich die Werte aus einer Lang- oder Breitform zu einem Zeichenkettevektor verketten lassen.

Teil III.

Deskriptive Datenanalyse

17. Daten beschreiben

ie Funktionen zur Ermittlung der Kennzahlen von Datenrahmen werden durch die Bibliothek `rstatix` bereitgestellt. `rstatix` ist eine Bibliothek, die die am häufigsten verwendeten statistischen Operationen und Tests in Abstimmung mit den `tidyverse`-Bibliotheken bereitstellt.

> **!** Achtung
>
> `rstatix` ist eine eigenständige Bibliothek und muss separat mit `install.packages()` oder `pak::pkg_install()` installiert werden.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to be
```

```
library(rstatix)
```

```
Attache Paket: 'rstatix'

Das folgende Objekt ist maskiert 'package:stats':

    filter
```

17.1. Universelle Kennwerte

In R ergibt sich dieser Wert direkt aus dem Datenrahmen: Der **Stichprobenumfang** entspricht der **Anzahl der Datensätze** in unserem Stichprobenobjekt. Diese Anzahl be-

stimmen wir mit Hilfe der `count()`-Funktion oder innerrhalb einer Transforrmation mit `mutate()` mit Hilfe der `n()`-Funktion.

Beispiel 17.1 (Beispieldaten für die universellen Kennwerte)**.**

```
stichprobe = read_csv2('data_ab_missing.csv')
```

```
i Using "','" as decimal and "'.'" as grouping mark. Use `read_delim()` for more control.

Rows: 156 Columns: 4
-- Column specification --------------------------------------------------------
Delimiter: ";"
chr (1): Angebot
dbl (3): Punkte, Interesse, Bedeutung

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.
```

Der *Stichprobenumfang* ist einer der drei allgemeinen Kennwerte, die jede Stichprobe beschreiben. Wir bestimmen zuerst die numerischen Kennwerte unserer Beispielstichprobe.

> ⚠ Wichtig
>
> Die beiden universellen Kennwerte werden beim Import ebenfalls **angezeigt**. Diese Werte stehen für das automatisierte Reporting nicht zur Verfügung. Deshalb müssen diese Kennwerte noch einmal bestimmt werden.

```
# Stichprobenumfang
stichprobe |>
    count() |>
    pull() -> stichprobenumfang

stichprobenumfang
```

```
[1] 156
```

```
stichprobe |>
    names() |>
    length()
```

```
[1] 4
```

Dabei erkennen wir, dass der Stichprobenumfang 156 beträgt. Wir erkennen zusätzlich, dass wir 4 Vektoren in unserem Stichprobenobjekt vorliegen haben. Die Funktion `dim()` funktioniert auch für Datenrahmen, so dass die beiden Kennwerte auch mit dieser Funktion bestimmt werden können. Das Ergebnis `dim()` Funktion gibt als ersten Wert immer den Stichprobenumfang und als zweiten Wert den Umfang der Umfang der Vektoren zurück.

```
stichprobe |>
    dim()
```

```
[1] 156   4
```

17.2. Variablenumfang und fehlende Werte

Neben dem *Stichprobenumfang* werden zusätzlich die *Variablenumfänge* ermittelt. Der **Variablenumfang** bezeichnet die **Anzahl der *gemessenen* Merkmalsausprägungen**. Damit ist die Gesamtzahl der gemessenen Werte für ein Merkmal in einem Vektor gemeint. Für diesen Wert müssen die *nicht vorhandene* Werte aus der Variable entfernt werden. Diese Werte sind in R mit dem Wert `NA` gekennzeichnet.

Die `NA`-Werte werden in R mithilfe der `drop_na()`-Funktion aus einem Datenrahmen entfernt. Die Funktion `drop_na()` entfernt alle *Datensätze*, in denen ein Vektor keinen Wert enthält. Das ist für den Variblenumfang nicht erwünscht.

Statdessen werden die fehlenden Werte bei einer *Aggregaion* mit `na.omit()` ausgeblendet.

> Praxis
>
> Die Variablenumfänge werden normalerweise gemeinsam mit den Lagemassen des Merkmals präsentiert.

```
stichprobe |>
    summarise(
        n_Punkte = Punkte |> na.omit() |> length(),
        n_Angebot = Angebot |> na.omit() |> length(),
        n_Interesse = Interesse |> na.omit() |> length(),
        n_Bedeutung = Bedeutung |> na.omit() |> length()
    )
```

```
# A tibble: 1 x 4
  n_Punkte n_Angebot n_Interesse n_Bedeutung
     <int>     <int>       <int>       <int>
1      153       152         151         150
```

17.3. Lagemasse

R stellt als Programmiersprache für die Data Sciences zentrale Funktionen zur Beschreibung von Daten direkt bereit. Es ist deshalb *in der Regel* nicht notwendig, die Formeln für die Lagemasse zu implementieren. Tabelle 17.1 liestet die Funktionen für die einzelnen Lagemasse.

Werden die Lagemasse für für mehrere Merkmale gleichzeitig bestimmt, dann handelt es sich um eine **gruppierte Operation**. Der Index dieser Operation wird über die Vektornamen gebildet. Dabei dürfen nur Vektoren von *gleichen Datentyp* zusammengefasst werden.

> **Merke**
>
> Alle Lagemasse sind Aggregationen.

Tabelle 17.1.: R-Funktionen für die Lagemasse

Mass	Funktion
Mittelwert	`mean()`
Median	`median()`
Standardabweichung	`sd()`
Varianz	`var()`
Interquartilsabstand	`IQR()`
Mittlere Absolute Abweichung	`mad()`
Quartile	`quantile()`
Standardfehler	`sd(x) / sqrt(length(x))`

Der Standardfehler hat in Base-R keine eigene Funktion, sondern muss über die Standardabweichung hergeleitet werden.

Die statistischen Kennwerte lassen sich also leicht mit der Aggregation in Beispiel 17.2 bestimmen.

Beispiel 17.2 (Alle Kennwerte bestimmen)**.**

```
stichprobe |>
    summarise(
        n = Punkte |> na.omit() |> length(),
        mn = Punkte |> mean(na.rm = TRUE),
        sd = Punkte |> sd(na.rm = TRUE),
        se = sd/sqrt(n),
        min = Punkte |> min(na.rm = TRUE),
        max = Punkte |> max(na.rm = TRUE),
        q1 = Punkte |> quantile(.25, na.rm = TRUE),
        md = Punkte |> median(na.rm = TRUE),
```

```
        q3 = Punkte |> quantile(.75, na.rm = TRUE),
        mad = Punkte |> mad(na.rm = TRUE),
        iqr = Punkte |> IQR(na.rm = TRUE)
    )
```

```
# A tibble: 1 x 11
      n    mn    sd    se   min   max    q1    md    q3   mad   iqr
  <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1   153  199.  69.7  5.64  16.5  395.  158.  196.  250.  68.8  92.2
```

Noch einfacher lassen sich die Kennwerte mit der Funktion `get_summary_stats()` aus der Bibliothek `rstatix` ermitteln (Beispiel 17.3). Ein wichtiger Vorteil dieser Funktion ist, dass sie ungültige Werte korrekt entfernt.

Beispiel 17.3 (Alle Kennwerte mit rstatix bestimmen)**.**

```
stichprobe |>
    get_summary_stats(Punkte)
```

```
# A tibble: 1 x 13
  variable     n   min   max median    q1    q3   iqr   mad  mean    sd    se
  <fct>    <dbl> <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Punkte     153  16.5  395.   196.  158.  250.  92.2  68.8  199.  69.7  5.64
# i 1 more variable: ci <dbl>
```

Für ordinalskalierte Daten sind nicht alle Kennwerte zulässig. Deshalb dürfen für diese Daten nur die zulässigen Werte berichtet werden. Der Funktion `get_summary_stats()` müssen deshalb die zulässigen Kennwerte angegeben werden (Beispiel 17.4).

Beispiel 17.4 (Alle Kennwerte für ordinalskalierte Daten bestimmen)**.**

```
kennwerte_ordinal = c(
    "n", "min", "max", "median", "mad", "iqr", "q1", "q3"
)

stichprobe |>
    get_summary_stats(
        -Punkte, # Alle Vektoren ausser Punkte
        show = kennwerte_ordinal
    )
```

```
# A tibble: 2 x 9
  variable      n   min   max median   mad   iqr    q1    q3
  <fct>     <dbl> <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>
1 Interesse   151     1     7      4  1.48   1.5     3   4.5
2 Bedeutung   150     1     6      3  1.48   2       2   4
```

> **Wichtig**
>
> Ordinalskalierte Daten müssen als numerische Werte *kodiert* vorliegen, damit die Kennwerte bestimmt werden können. Damit die Kennwerte interpretiert werden können, müssen alle verwendeten Kodierungstabelle ebenfalls berichtet werden.

17.3.1. Kennwerte über das Datenschema bestimmen

Das Datenschema ist ein zentraler Teil der technischen Dokumentation eines Projekts. Liegt das Datenschema als Tabelle vor und die Skalierungen sind in einer eigenen Spalte dokumentiert, dann kann das Datenschema zur Beschreibung der Daten eingesetzt werden.

Beispiel 17.5 zeigt ein reduziertes Datenschema für die geladenen Daten. Normalerweise würde dieses Schema aus einer Datei geladen werden.

Beispiel 17.5 (Reduziertes Datenschema für die Daten)**.**

```
# datenSchema = read_csv("datenschema.csv")
datenSchema = tribble(
    ~Name, ~Skalierung,
    "Punkte", "metrisch",
    "Angebot", "nominal",
    "Interesse", "ordinal",
    "Bedeutung", "ordinal"
)
```

Nachdem das Schema geladen wurde, können die Namen der Vektoren des gleichen Skalenniveaus gefiltert werden und an die Selektorfunktion `all_of()` übergeben werden (s. Beispiel 17.6). Auf diese Weise wird der Funktion `get_summary_stats()` mitgeteilt, für welche Vektoren die Kennwerte bestimmt werden sollen.

Beispiel 17.6 (Verwendung des Datenschames zur Beschreibung der Daten)**.**

```
# Metrisch-skalierte Kennwerte
stichprobe |>
    get_summary_stats(
        datenSchema |>
            filter(Skalierung == "metrisch") |>
```

```
            pull(Name) |>
            all_of()
    )
```

```
# A tibble: 1 x 13
  variable     n   min   max median    q1    q3   iqr   mad  mean    sd    se
  <fct>    <dbl> <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Punkte     153  16.5  395.   196.  158.  250.  92.2  68.8  199.  69.7  5.64
# i 1 more variable: ci <dbl>
```

```
# Ordinalskalierte Kennwerte
stichprobe |>
    get_summary_stats(
        datenSchema |>
            filter(Skalierung == "ordinal") |>
            pull(Name) |>
            all_of()
        ,
        show = kennwerte_ordinal
    )
```

```
# A tibble: 2 x 9
  variable      n   min   max median   mad   iqr    q1    q3
  <fct>     <dbl> <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>
1 Interesse   151     1     7      4  1.48   1.5     3   4.5
2 Bedeutung   150     1     6      3  1.48   2       2   4
```

17.4. Kontinenztabellen erstellen

In Kapitel 13 wurden Kontingenztabellen eingeführt und die Funktion `table()` für zwei unabhängige Vektoren vorgestellt.

Für Datenrahmen ist die Funktion `freq_table()` etwas flexibler und unterliegt nicht der Beschränkung auf zwei Vektoren. Ausserdem bestimmt die Funktion `freq_table()` automatisch die relativen Häufigkeiten.

Beispiel 17.7 (Kontingenztabelle für ein Merkmal)**.**

```
stichprobe |>
    freq_table(Angebot)
```

```
# A tibble: 2 x 3
  Angebot     n  prop
  <chr>   <int> <dbl>
1 A          69  45.4
2 B          83  54.6
```

Beispiel 17.8 (Kontingenztabelle für zwei Merkmale)**.**

```
stichprobe |>
    freq_table(Angebot, Bedeutung)
```

```
# A tibble: 12 x 4
   Angebot Bedeutung     n  prop
   <chr>       <dbl> <int> <dbl>
 1 A               1     5   7.5
 2 A               2    15  22.4
 3 A               3    12  17.9
 4 A               4    15  22.4
 5 A               5    14  20.9
 6 A               6     6   9
 7 B               1    11  13.9
 8 B               2    12  15.2
 9 B               3    18  22.8
10 B               4    22  27.8
11 B               5     8  10.1
12 B               6     8  10.1
```

Vorteilhaft ist auch, dass das Ergebnis der Funktion `freq_table()` ein Datenrahmen ist. Dadurch können weitere Datenrahmen-Operationen verkettet werden. Im Beispiel 17.9 werden die relativen Werte in einer Kreuztabelle so gegenübergestellt, dass die unterschiedliche Bewertung der Bedeutung für die beiden Angebote leichter verglichen werden können.

Beispiel 17.9 (Kontingenztabelle für zwei Merkmale als Kreuztabelle)**.**

```
stichprobe |>
    freq_table(Angebot, Bedeutung) |>
    select(-n) |>
    pivot_wider(names_from = Angebot, values_from = prop)
```

```
# A tibble: 6 x 3
  Bedeutung     A     B
      <dbl> <dbl> <dbl>
1         1   7.5  13.9
```

```
2         2  22.4  15.2
3         3  17.9  22.8
4         4  22.4  27.8
5         5  20.9  10.1
6         6   9    10.1
```

18. Daten visualisieren

In R erstellen wir Plots mit Hilfe der `ggplot2`-Funktionen. Anders als in Excel werden mit diesen Funktionen Plots schrittweise aufgebaut und können auf diese Weise einheitlich reproduziert werden.

Als Erstes laden wir wie immer die `tidyverse`-Bibliothek, damit wir die ggplot-Funktionen und die Funktionsverkettung verwenden können.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.3     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.3     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to be
```

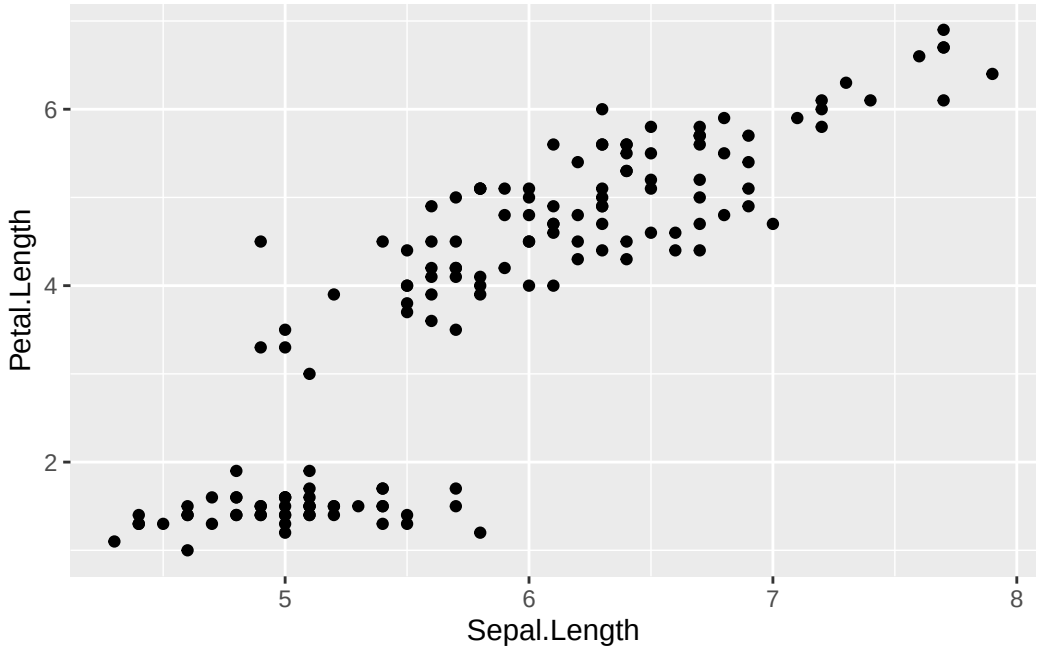
Wir erzeugen eine Datenvisualisierung immer mit den folgenden Funktionsaufrufen:

1. Wir initiieren einen Plot mit der Angabe der darzustellenden Vektoren.
2. Anschliessend wählen wir die Darstellung der Datenpunkte mit einer *Geometriefunktion* aus.

> **Achtung**
>
> `ggplot2` verwendet zur Funktionsverkettung den `+`-Operator und nicht wie der Rest der modernen R-Funktionen den Verkettungsoperator `|>`. Wir können also leicht erkennen, dass ein Code-Fragment einen Plot erzeugt, wenn Funktionen mit `+` verkettet sind.

Mit den folgenden Funktionsaufrufen erzeugen wir ein einfaches Punktdiagramm. Als Beispielstichprobe verwenden wir hier die `iris`-Daten, die mit R mitgeliefert werden.

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



18.1. Technischer Aufbau von Visualisierungen

In der ersten Zeile legen wir fest, welche Daten visualisiert werden sollen. Die Grundlage für jede Datenvisualisierung ist immer ein Transformationsergebnis.

In der zweiten Zeile signalisieren wir R mit der Funktion `ggplot()`, dass wir einen Plot erzeugen wollen. Wir übergeben als Parameter das Ergebnis der `aes()`-Funktion.

> **Hinweis**
>
> Die Funktion `aes()` legt die ***ästhetischen*** Voraussetzungen für einen Plot fest. Damit legen wir fest, welche Daten für unsere Datenpunkte verwendet werden sollen.

In diesem Beispiel legen wir den Vektor `Sepal.Length` für die Koordinaten auf der x-Achse und `Petal.Length` für die Koordinaten auf der y-Achse fest. Die Datenpunkte werden also durch die beiden gemeinsam auftretenden Werte in diesen Vektoren festgelegt.

Mit der dritten Zeile legen wir die ***Geometrie*** der Datenpunkte fest. Alle `ggplot2`-Funktionsnamen zur Darstellung von Datenpunkten beginnen mit `geom_` (für Geometrie). In diesem Beispiel wollen wir unsere Datenpunkte mit Punkten (engl. Points) darstellen. Deshalb verwenden wir die Funktion `geom_point()`.

Diese drei Schritte zeigen die grundsätzliche Logik zum Erstellen von Plots mit R.

18.2. Mathematische Funktionen visualisieren

In der Mathematik werden regelmässig Funktionen besprochen. Diese Funktionen können wir mit R leicht visualisieren.

Dabei nutzen wir aus, dass wir in R neue Funktionen mit dem Schlüsselwort `function` definieren können. Im folgenden Beispiel verwenden wir die beiden Funktionen.

$$f1(x) \rightarrow x^2 - 3x$$

und

$$f2(x) \rightarrow 4x + 2$$

Im nächsten Schritt erstellen wir unsere beiden mathematischen Funktionen. Dabei beachten wir, dass wir den Namen der jeweiligen Funktion als eine Variable zuweisen müssen. Die rechte Seite der Zuweisung zeigt R an, dass wir eine neue Funktion mit dem Parameter `x` erstellen möchten. Nach dieser *Funktionsdefinition* folgt der sog. Funktionskörper in geschweiften Klammern. Hier schreiben wir die Formel unserer Funktion in der ausführlichen Operatorenschreibweise. Anders als bei der mathematischen Schreibweise dürfen wir keine Operatoren weglassen.

```
f1 = function (x) {
    x ^ 2 - 3 * x
}

f2 = function (x) {
    4 * x + 2
}
```

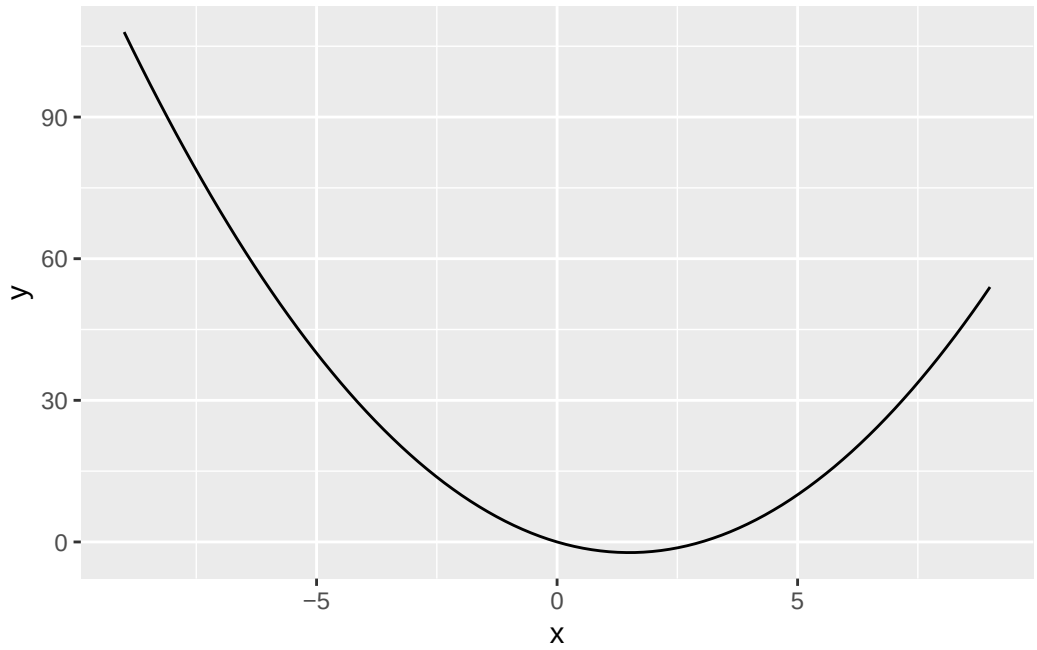
Damit `ggplot` "weiss", welches Intervall für `x` wir darstellen möchten, erzeugen wir eine Stichprobe mit einem Vektor `x`, der genau zwei Werte hat. Diesen Vektor weisen wir der Variable `Darstellungsbereich` zu. Wenn wir die Werte symmetrisch angeben, dann landet die 0 auf der x-Achse in der Mitte unseres Diagramms.

```
Darstellungsbereich = tibble(x = c(-9, 9))
```

Jetzt können wir unsere Funktion visualisieren. Wir übergeben die Stichprobe im der Variablen `Darstellungsbereich` an die `ggplot()`-Funktion und legen mit dem Aufruf der `aes()`-Funktion mit dem Vektor `x` die Grenzen für die x-Achse fest. Anschliessend rufen wir die Funktion `geom_line()` auf, um einen Graphen zu erzeugen. Weil wir keine Werte für die y-Achse haben, legen wir fest, dass wir die y-Werte aus einer Funktion berechnen wollen. Das erreichen wir mit dem Parameter `stat = "function"`. Sobald wir diesen Parameter angeben, erwartet die `geom_line()` Funktion eine Funktion zur Berechnung der y-Werte.

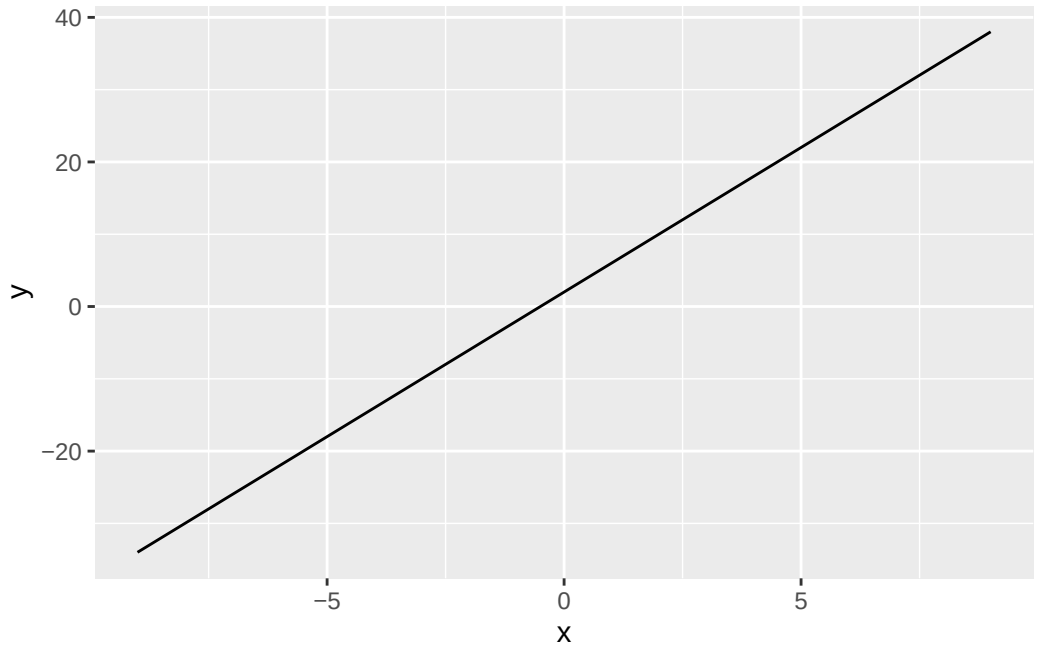
Diese Funktion übergeben wir mit dem Parameter `fun = f1`, wobei `f1` eine unserer vorab definierten Funktionen ist.

```
Darstellungsbereich |>
    ggplot(aes(x)) +
        geom_line(stat = "function", fun = f1)
```



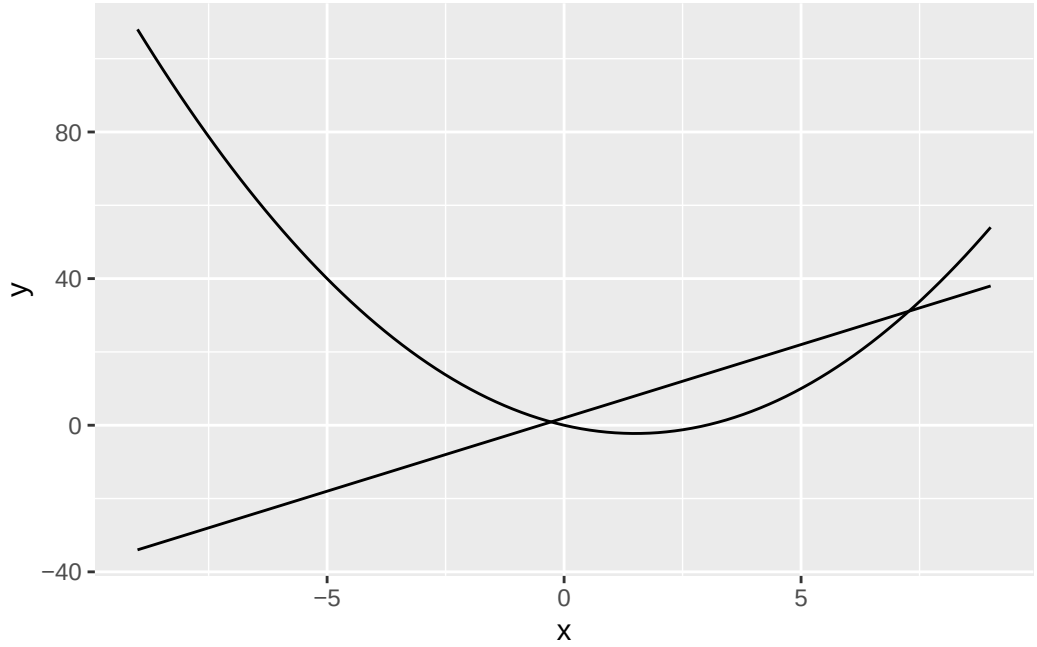
Diesen Schritt können wir für die Funktion `f2` wiederholen.

```
Darstellungsbereich |>
    ggplot(aes(x)) +
        geom_line(stat = "function", fun = f2)
```



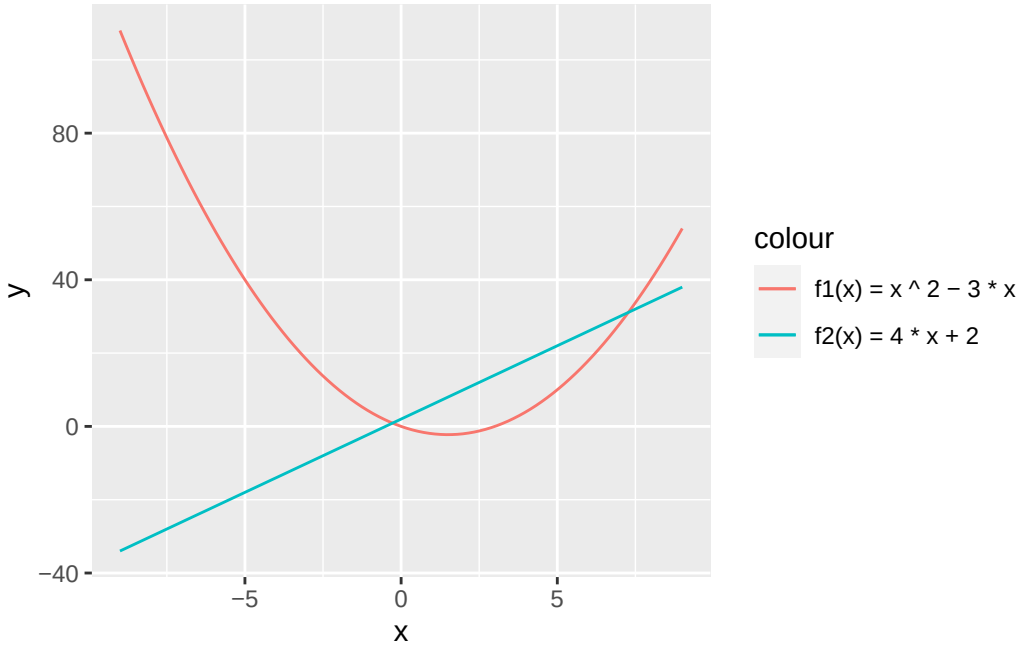
Weil wir mit `ggplot` Darstellungen überlagern können, dürfen wir die beiden Funktionen selbstverständlich auch in einem Diagramm darstellen.

```
Darstellungsbereich |>
    ggplot(aes(x)) +
        geom_line(stat = "function", fun = f1) +
        geom_line(stat = "function", fun = f2)
```



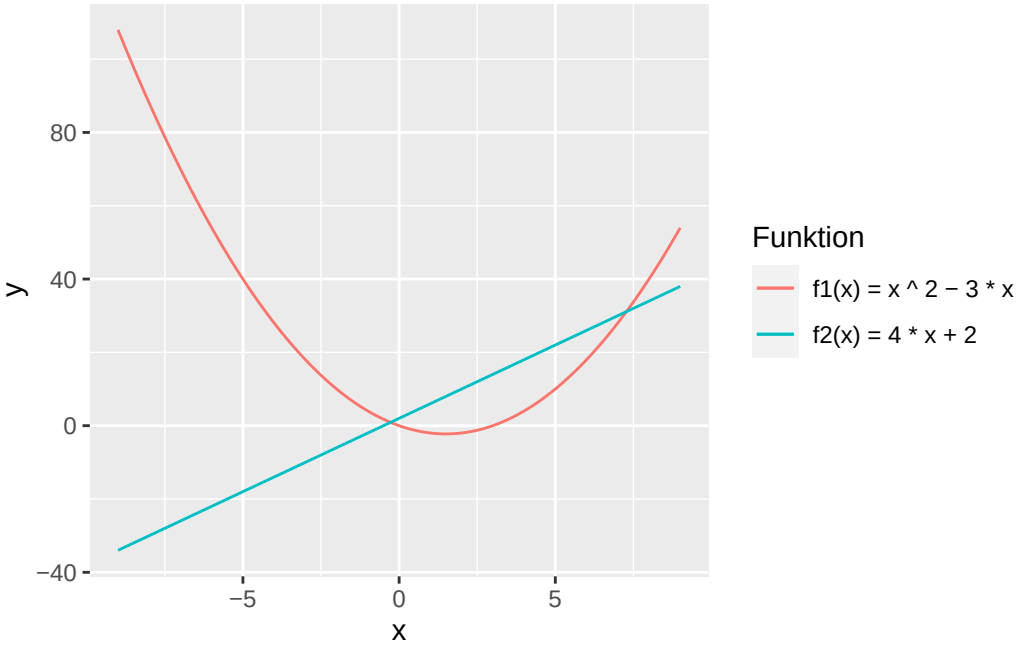
Wir wollen die beiden Graphen noch visuell hervorheben, damit wir wissen, welcher Graph zu welcher Funktion gehört. Dabei überlassen wir die Auswahl der Farben `ggplot`, womit wir sicherstellen, dass die Farben nicht zu ähnlich sind. Dazu verwenden wir den Trick, dass wir jeder Geometrie-Funktion ergänzende ästhetische Parameter übergeben dürfen.

```
Darstellungsbereich |>
    ggplot(aes(x)) +
        geom_line(stat = "function", fun = f1,
                  aes(colour = "f1(x) = x ^ 2 - 3 * x")) +
        geom_line(stat = "function", fun = f2,
                  aes(colour = "f2(x) = 4 * x + 2"))
```



Die Legende für unser Diagramm hat keine schöne Überschrift. Das passen wir noch schnell mit der `labs()`-Funktion(für *labels* bzw. *Beschriftungen*) an. Dort geben wir für den ästhetischen Parameter die richtige Beschriftung an. In unserem Fall ist das `colour`.

```
Darstellungsbereich |>
    ggplot(aes(x)) +
        geom_line(stat = "function", fun = f1,
                  aes(colour = "f1(x) = x ^ 2 - 3 * x")) +
        geom_line(stat = "function", fun = f2,
                  aes(colour = "f2(x) = 4 * x + 2")) +
        labs(colour = "Funktion")
```



18.3. Berechnete Visualisierungen

Wir haben im Abschnitt zu einfachen Visualisieren in R die Funktion `ggplot()` kennengelernt, um zwei Vektoren zu visualisieren.

Sehr häufig wir einen Vektor und wollen sehen, wie die Werte in diesem Vektor verteilt sind. Wir sollen also die Werte in dem Vektor für die Visualisierung **aggregieren**. Hierbei handelt es sich um eine so häufige Aufgabe, dass uns `ggplot()` diese Aufgabe abnimmt.

Für verschiedene Visualisierungen hat `ggplot()` vordefinierte Funktionen, mit denen Werte für die Visualisierung aufbereitet werden, wenn die Werte für eine Achse fehlen. Diese Funktionen müssen wir im Detail nicht kennen, denn `ggplot()` wählt diese automatisch für uns aus.

Wir lernen heute zwei wichtige berechnete Visualisierungen kennen.

1. Das Histogramm
2. Den Box-Plot

18.3.1. Histogramm

Definition 18.1. Als **Histogramm** werden Balkendiagramme bezeichnet, die die *Häufigkeiten* von gemessenen Werte darstellen.

Das übliche Balkendiagramm erzeugen wir mit der Funktion `geom_bar()`. Diese Funktion verwenden wir immer, wenn unsere gemessenen Werte nur auf bestimmte Werte fallen (können). Die `geom_bar()`-Funktion zählt für jeden gemessenen Wert die Anzahl der Datensätze, in denen dieser Wert vorkommt.

Gelegentlich sind unserer Werte so verteilt, dass nur selten zwei oder mehr Datensätze gleiche Werte haben. In solchen Fällen verwenden wir die Funktion `geom_histogram()`. Diese Funktion teilt den gesamten Wertebereich in gleichmässige Intervalle und zählt die Anzahl der Datensätze im jeweiligen Intervall.

Das folgende Beispiel veranschaulicht die Situation.

Wir verwenden die Stichprobe `digitales_umfeld1.csv`. In dieser Stichprobe gibt es den Vektor `tage`, der das Alter der beantwortenden Person in Tagen festhält. Dabei handelt es sich rein formell um *diskrete Werte*. Wenn wir die Verteilung dieser Werte in einem Histogramm für *diskrete Werte* darstellen würden, dann erhalten wir das folgende Histogramm:

```
digitales_umfeld = read_csv("befragung_digitales_umfeld/digitales_umfeld1.csv")
```

```
Rows: 135 Columns: 5
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): mobilgeraet
dbl (4): alter, tage, monate, digitalisiert

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.
```

```
digitales_umfeld |>
    ggplot(aes(x = tage)) +
        geom_bar()
```

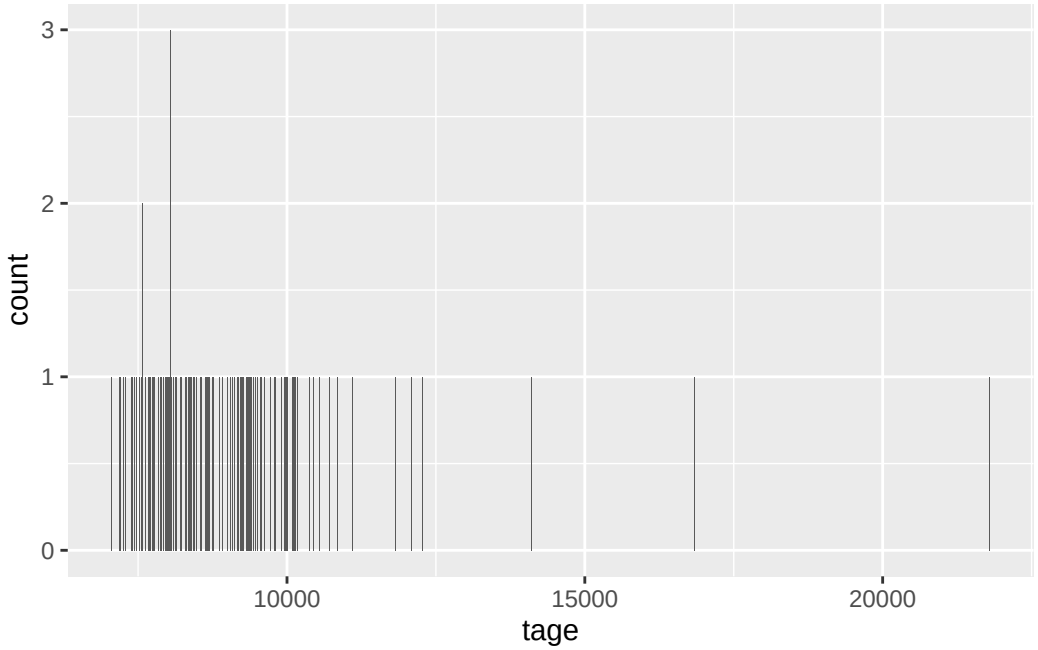


Abbildung 18.1.: Tage-Histogram Diskrete Werte

Auf diesem Histogramm kann man keine Verteilung erkennen. Es scheint, als ob alle Werte genau einmal vorkommen. Der Wertebereich der y-Achse deutet aber darauf hin, dass einzelne Werte bis zu drei Mal vorkommen. Diese Balken sind jedoch so dünn, dass sie nicht im Diagramm dargestellt werden können.

Die Werte in diesem Vektore verhalten sich also wie *kontinuierliche Werte*. Deshalb verwenden wir die Funktion `geom_histogram()`, um die Daten darzustellen.

```
digitales_umfeld |>
    ggplot(aes(x = tage)) +
        geom_histogram()
```

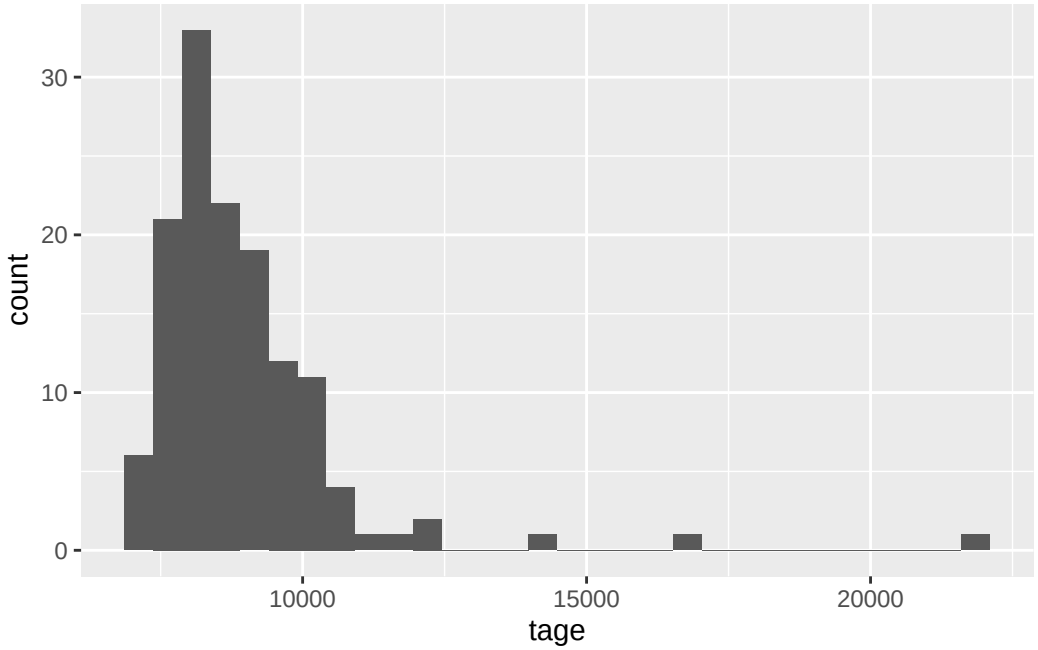


Abbildung 18.2.: Tage-Histogram

Aus diesem Histogramm können wir wesentlich besser die Verteilung des Alters in Tagen ablesen, weil der Wertebereich in grössere Segmente gegliedert wurde und die Datensätze in diesen Segmenten gezählt wurden.

> **Merke**
>
> Histogramme für *kontinuierliche Werte* erzeugen wir mit der Funktion `geom_histogram()`. Histogramme für *diskrete Werte* erzeugen wir mit der `geom_bar()`-Funktion.

18.3.1.1. Histogramme selbst berechnen

Gelegentlich haben uns bereits die Häufigkeiten für ein Histogramm als Teil unserer Stichproben. In solchen Fällen verwenden wir die Funktion `geom_col()`, um die Daten als Histogramm darzustellen. In diesem Fall müssen wir neben der x-Achse auch den Vektor mit den berechneten Werten für die y-Achse an `ggplot()`'s `aes()`-Funktion übergeben.

18.3.2. Box-Plot

Definition 18.2. Ein Box-Plot stellt die Verteilung eines Stichprobenvektors mit Hilfe von Quartilen dar.

Box-Plots werden mit der `geom_boxplot()` Funktion dargestellt.

Beim Box-Plot wird der Median als dicke Linie dargestellt. Der Interquartilsabstand wird als Rechteck (*Box*) um den Median visualisiert (2. und 3. Quartil). Dabei liegt die Hälfte der aller gemessenen Werte innerhalb der dargestelltn Box. Der gesamte Umfang wird durch Linien links (1. Quartil) und rechts (4. Quartil) vom Interquartilsabstand dargestellt. Manchmal werden Punkte an den äusseren Rändern dargestellt. Diese Punkte stellen sog. Ausreisser dar.

Wiederholen wir die Visualisierung für unsere Alterstage mit einem Boxplot, dann erhalten wir folgendes Ergebnis:

```
digitales_umfeld |>
    ggplot(aes(x = tage)) +
        geom_boxplot()
```

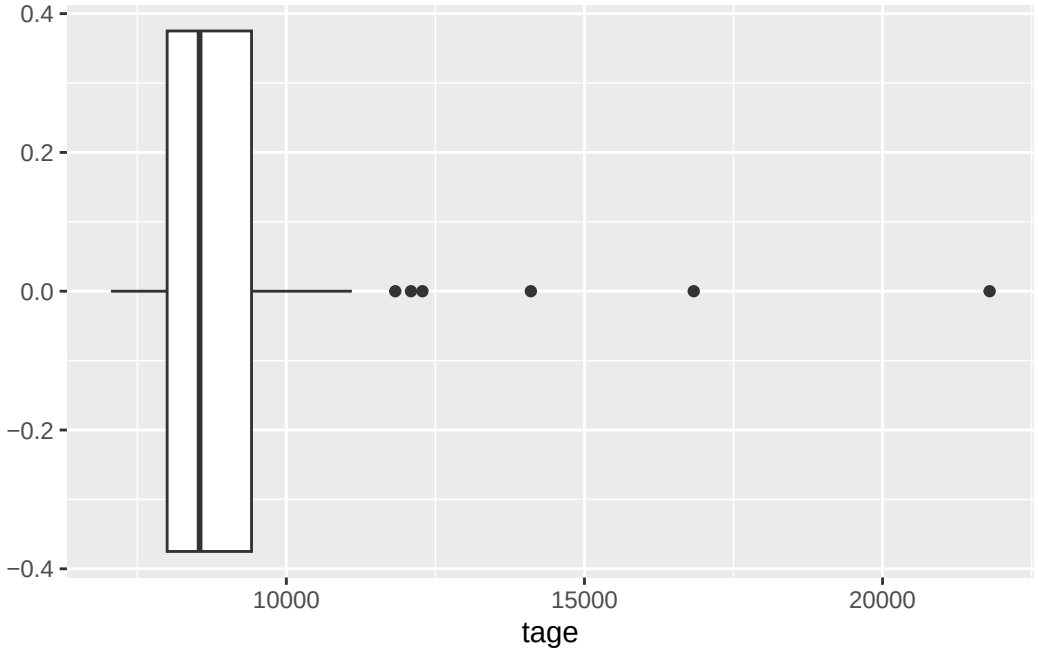


Abbildung 18.3.: Tage-Boxplot

Wir erkennen jetzt leicht, dass der Grossteil der Gruppe unter 10000 Tagen alt ist und dass es sechs Ausreisser gibt, die deutlich älter als der Grossteil der Gruppe sind.

18.3.3. Punkt- und Jitter-Diagramme

Die dritte wichtige visuelle Analysetechnik sind Punktwolken. Bei Punktwolken stellen wir die Werte von zwei Vektoren ähnlich einer Kreuztabelle gegenüber und überprüfen das gemeinsame Auftreten von Werten in den Vektoren unserer Messungen.

Für Punktwolken stehen zwei Funktionen zur Verfügung:

1. `geom_point()` für kontinuierliche Werte.
2. `geom_jitter()` für diskrete Werte.

Mit Punktdiagrammen werden zwei Merkmale mit kontinuierlichen Daten gegenübergestellt. Punktdiagramme bilden **Beziehungen zwischen Merkmalen** ab.

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

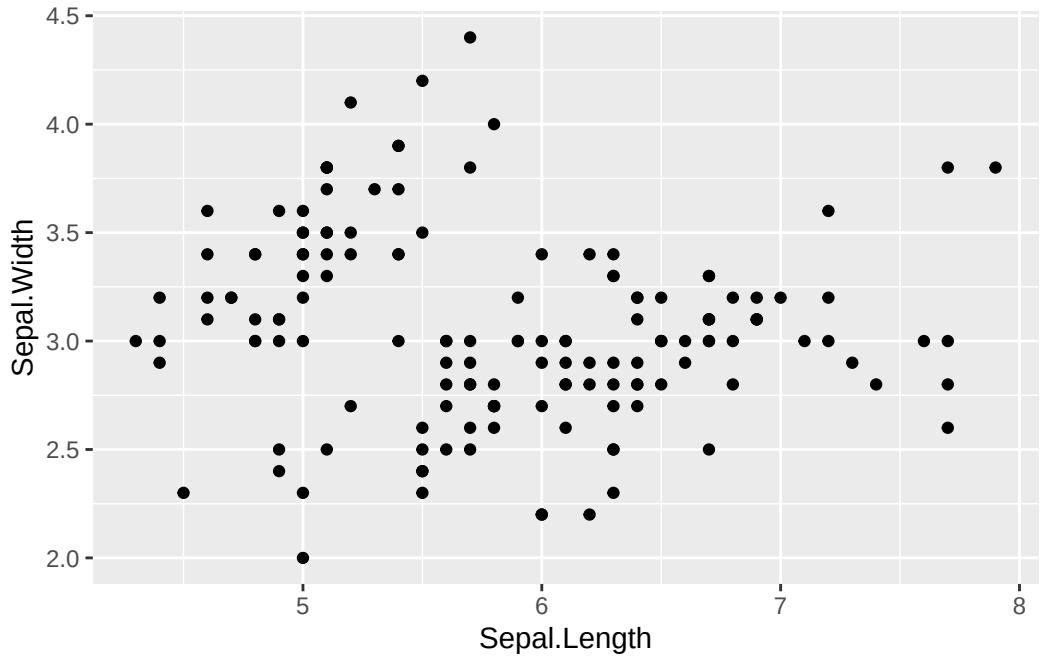


Abbildung 18.4.: Punktdiagramm der `iris`-Daten

```
daten = read_csv("befragung_digitales_umfeld/befragung.csv");

daten |>
    filter(q18_6 > 0 &
           q18_7  > 0 ) |>
    ggplot(aes(x = q18_6 ,
               y = q18_7)) +
        geom_point()
```

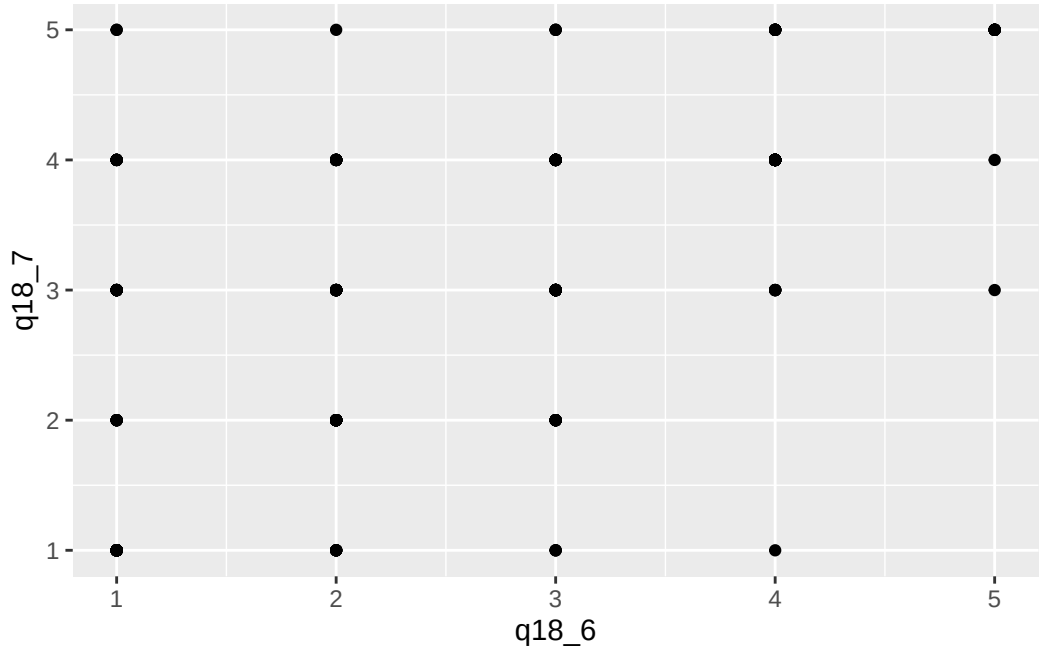


Abbildung 18.5.: Beispiel eines Punktdiagramms mit diskreten Daten

In diesem Beispiel sehen wir, dass alle Werte genau an den gleichen Punkten im Koordinatensystem liegen. Ein solcher Plot ist ein gutes Beispiel für *diskrete Werte*. Bei diskreten Werten fallen alle Messungen genau auf bestimmte Punkte im Wertebereich. Kontinuierliche Werte weichen oft ein wenig voneinander ab, sodass wir eine Wolke sehen würden.

Um Punktwolken für diskrete Werte zu erzeugen, verwenden wir die `geom_jitter()`-Funktion. Diese Funktion erzeugt einen kleinen Bereich um den diskreten (echten) Messwert und verteilt die einzelnen Datensätze mit einem zufälligen Abstand vom echten Messwert. Dadurch wird das gemeinsame Auftreten von Werten deutlich sichtbar, sofern es Gemeinsamkeiten gibt.

```
daten |>
    filter(q18_6 > 0 &
           q18_7  > 0 ) |>
    ggplot(aes(x = q18_6 ,
               y = q18_7)) +
        geom_jitter()
```

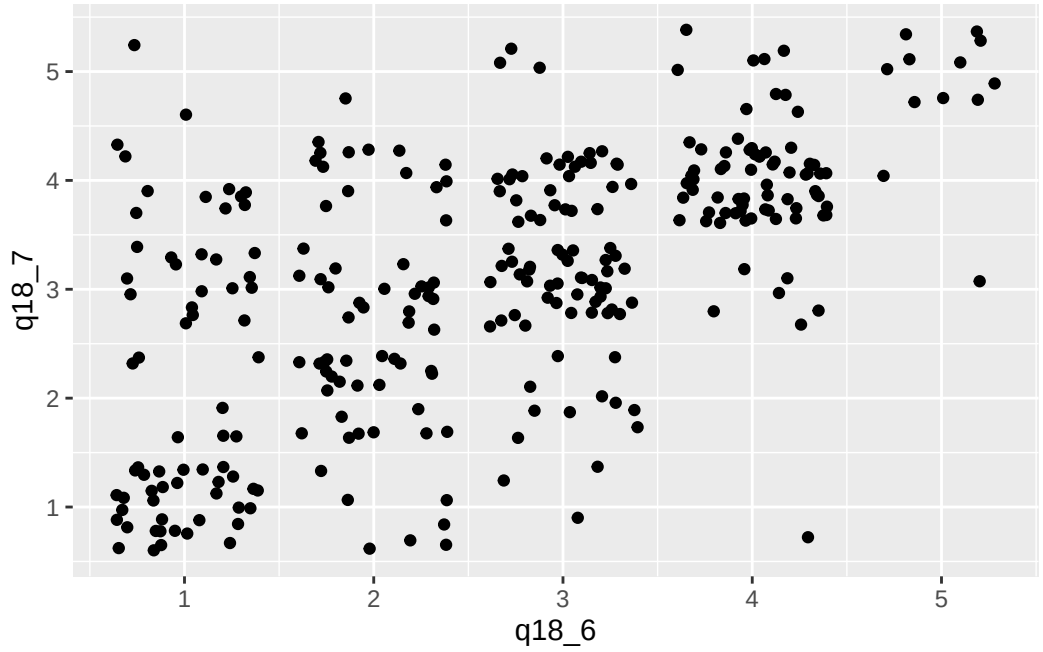


Abbildung 18.6.: Beispiel eines Jitter-Diagramms

Durch den leichten Versatz sind nun gehäufte Wertepaare leichter zu erkennen. Bei Jitter-Plots dürfen wir aber nie vergessen, dass die Punkte zwar Messungen repräsentieren, aber leicht vom echten Messpunkt versetzt dargestellt wurden.

18.3.4. Ausgleichsgeraden

Im vorigen Abschnitt können wir eine Häufung entlang der nach rechts aufsteigenden Diagonalen erkennen. Solche Häufungen in Punktwolken deuten auf *Korrelationen* hin.

Definition 18.3. Eine **Korrelation** bezeichnet das wiederholte Auftreten von Wertepaaren in Stichproben. Korrelationen deuten auf Zusammenhänge zwischen zwei Vektoren hin.

Ähnlich wie beim Vergleichen mit Histogrammen ist es bei Punktwolken hilfreich, für die Wolke eine Referenz zur Orientierung zu haben. Das erreichen wir mit der `geom_smooth()`-Funktion. Die Methode `lm` steht für "lineares Modell". In diesem Modell versteckt sich das Wort Linie und deshalb *erzeugt ein lineares Modell immer eine Ausgleichsgerade*. Der graue Bereich zeigt uns die Spanne des Fehlerbereichs für diese Gerade. Bei einer linearen Korrelation sollte diese Gerade den Häufungen in unserem Plot ungefähr folgen.

```
daten |>
    filter(q18_6 > 0 &
           q18_7  > 0 ) |>
    ggplot(aes(x = q18_6 ,
```

```
              y = q18_7)) +
        geom_jitter() +
        geom_smooth(method=lm)
```

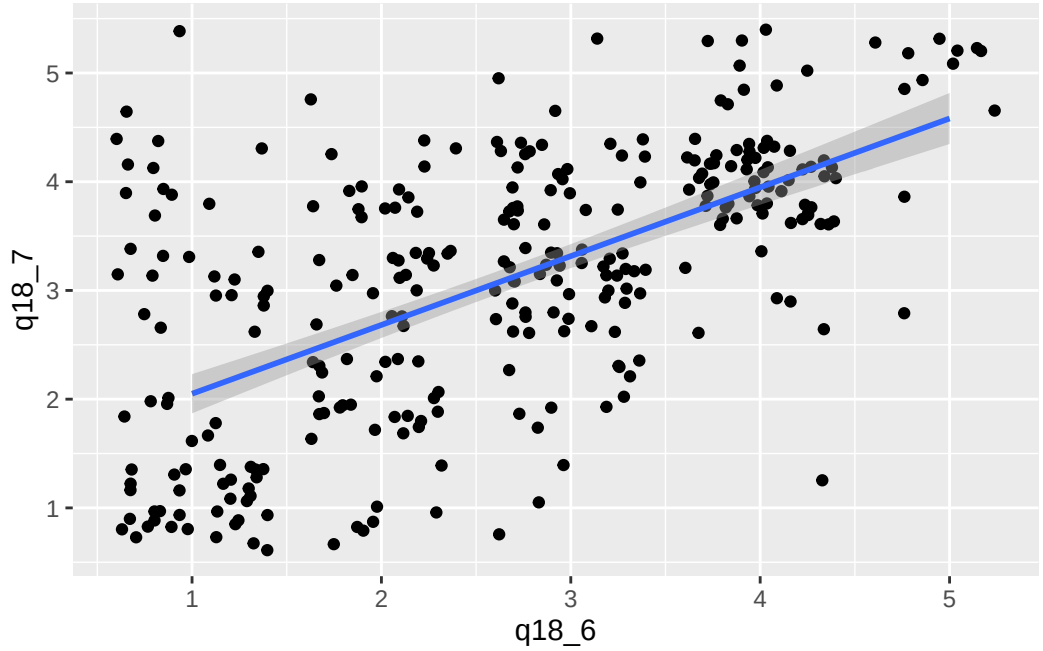


Abbildung 18.7.: Beispiel eines Jitter-Diagramms mit Ausgleichsgerade

Das folgende Beispiel zeigt eine Punktwolke, bei der die Wertepaare zufällig über den gesamten Wertebereich gestreut sind. In diesem Fall ist eine Korrelation kaum wahrscheinlich.

```
daten |>
    filter(q18_3 > 0 &
           q18_29  > 0 ) |>
    ggplot(aes(x = q18_3 ,
               y = q18_29)) +
        geom_jitter() +
        geom_smooth(method="lm")
```

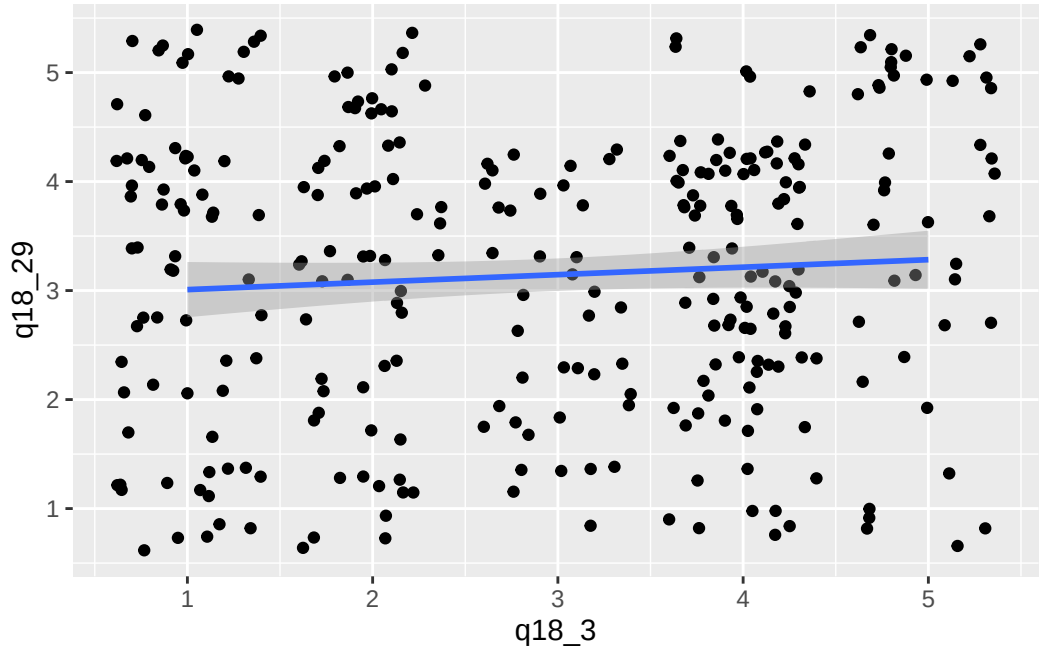


Abbildung 18.8.: Beispiel eines Jitter-Diagramms mit flacher Ausgleichsgerade

Im Beispiel ist die Ausgleichsgerade fast waagerecht. Wenn eine Ausgleichsgerade fast waagerecht ist, dann liegt in der Regel auch keine *Korrelation* vor.

Es gibt auch nicht-lineare Korrelationen. In diesem Fall sehen wir Häufungen in bestimmten Teilen unserer Punktwolken oder unsere Punkte folgen einer Kurve. Solche Korrelationen müssten einer entsprechenden "Ausgleichskurve" folgen. Eine solche Ausgleichskurve erzeugen wir mit `loess` als Ausgleichsmethode.

Wenn eine Ausgleichskurve fast gerade ist, dann sollten wir eine lineare Korrelation annehmen. Ein Beispiel für eine fast gerade Ausgleichskurve zeigt uns der nächste Plot.

```
daten |>
    filter(q18_3 > 0 &
           q18_29  > 0 ) |>
    ggplot(aes(x = q18_3 ,
               y = q18_29)) +
        geom_jitter() +
        geom_smooth(method = "loess")
```

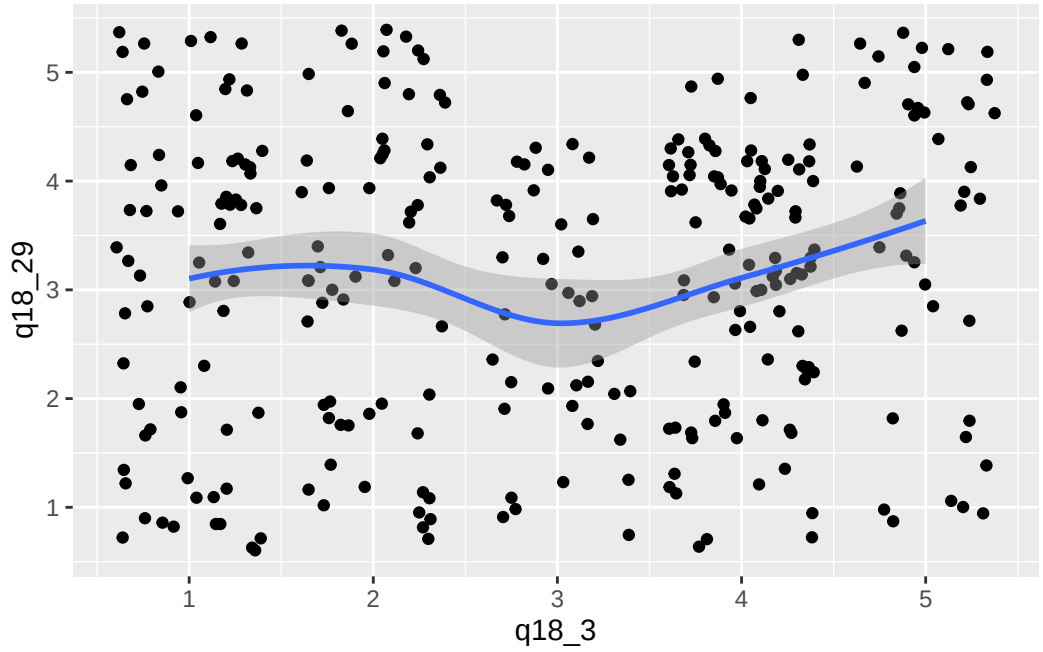


Abbildung 18.9.: Beispiel eines Jitter-Diagramms mit nicht linearer Ausgleichslinie

18.4. Mehrdimensionale Plots mit `aes`

18.4.1. Farbkodierung

Mit der Farbkodierung können zusätzliche Merkmale in einer Visualisierung kodiert werden. ggplot unterscheidet für die Farbkodierung zwischen diskreten und kontinuierlichen Daten. Bei diskreten Daten verwendet R Farben, die sich gut voneinander unterscheiden lassen. Bei kontinuierlichen Daten werden Farbverläufe zwischen mehreren Farben verwendet. Ausserdem hängt die Färbung von der Art der Visualisierung ab. Flächige Darstellungselemente, wie die Balken von Balkendiagrammen, wird eine *Füllfarbe* gesetzt. Die Füllfarbe wird über die Ästhetik `fill` kontrolliert. Darstellungselemente wie Linien oder Punkte wird eine *Linienfarbe* gesetzt. Die Linienfarbe wird über die Ästhetik `colour` festgelegt.

> **Merke**
>
> Die Färbung von Punkten wird über die Linienfarbe gesteuert.

```
iris |>
    ggplot(aes( x = Sepal.Length,
                y = Sepal.Width,
                colour = Species)) +                    (1)
        geom_point()
```

① Farbliche Hervorhebung

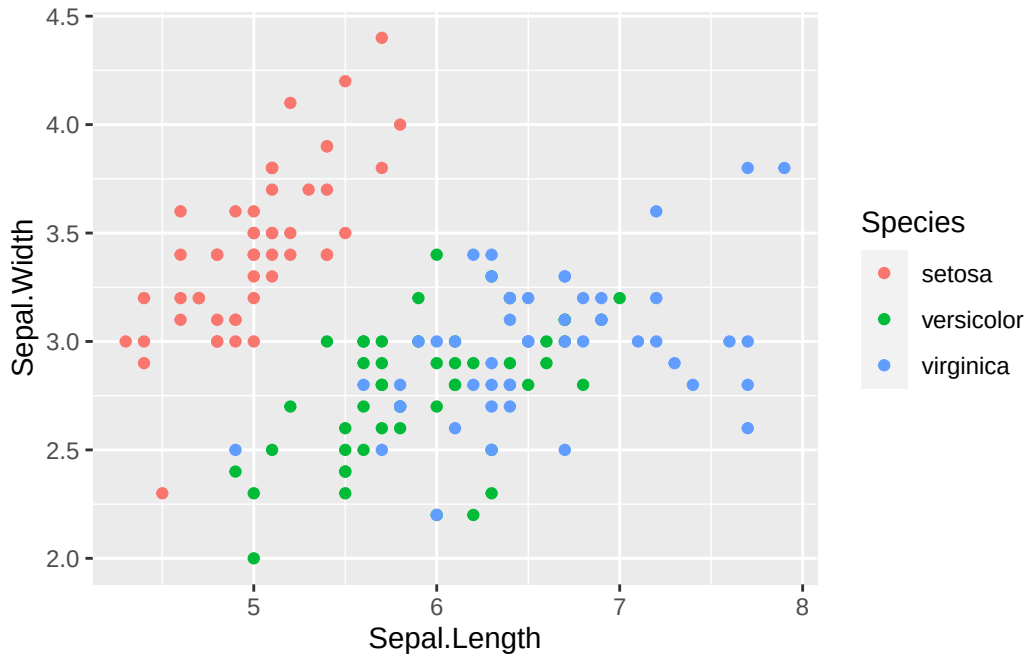


Abbildung 18.10.: Punktdiagramm der `iris`-Daten mit farblich hervorgehobenen Spezieswerten

Balkendiagramme und Histogramme benötigen eine Flächenfärbung.

```
iris |>
    ggplot(aes( x = Sepal.Length,
                fill = Species)) +                                    ①
        geom_histogram()
```

① Farbliche Hervorhebung

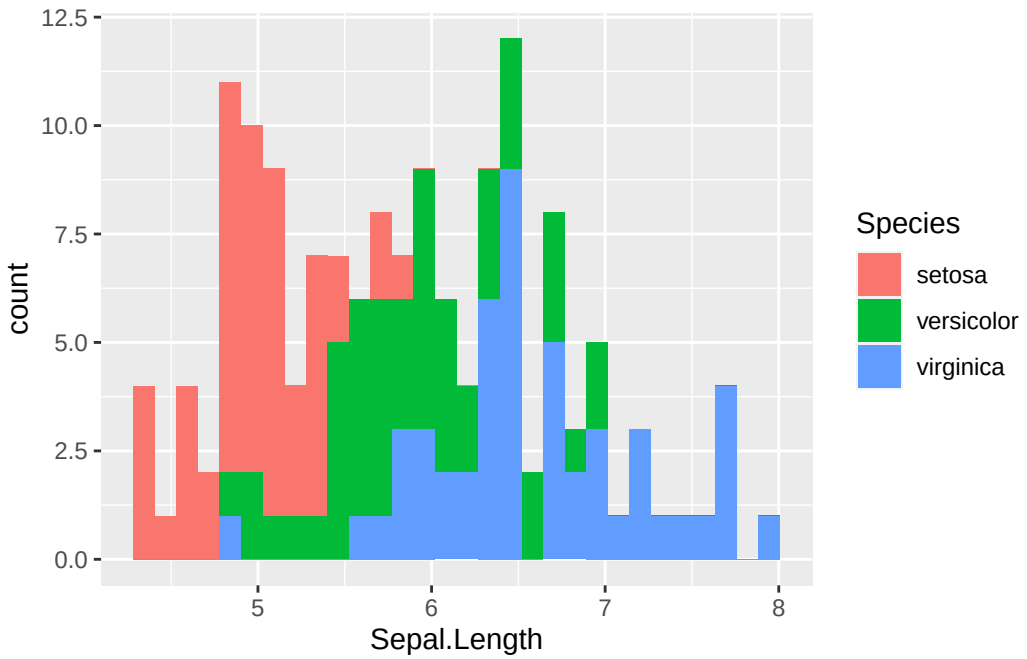


Abbildung 18.11.: Histogramm der `iris`-Daten mit farblich hervorgehobenen Spezieswerten

> **Praxis**
>
> Wenn Histogramme oder Balkendiagramme eine zusätzliche farbliche Kodierung erhalten, verbirgt eine gestapelte Balkendarstellung die Verteilungen. Mit dem Parameter `position` lässt sich die Positionierung der Balken steuern. Dies wird beispielsweise durch die Funktion `position_dodge()` vereinfacht.

```
iris |>
    ggplot(aes( x = Sepal.Length,
                fill = Species)) +                          ①
        geom_histogram(position = position_dodge())         ②
```

① Farbliche Hervorhebung
② Positionierung der hervorgehobenen Balken

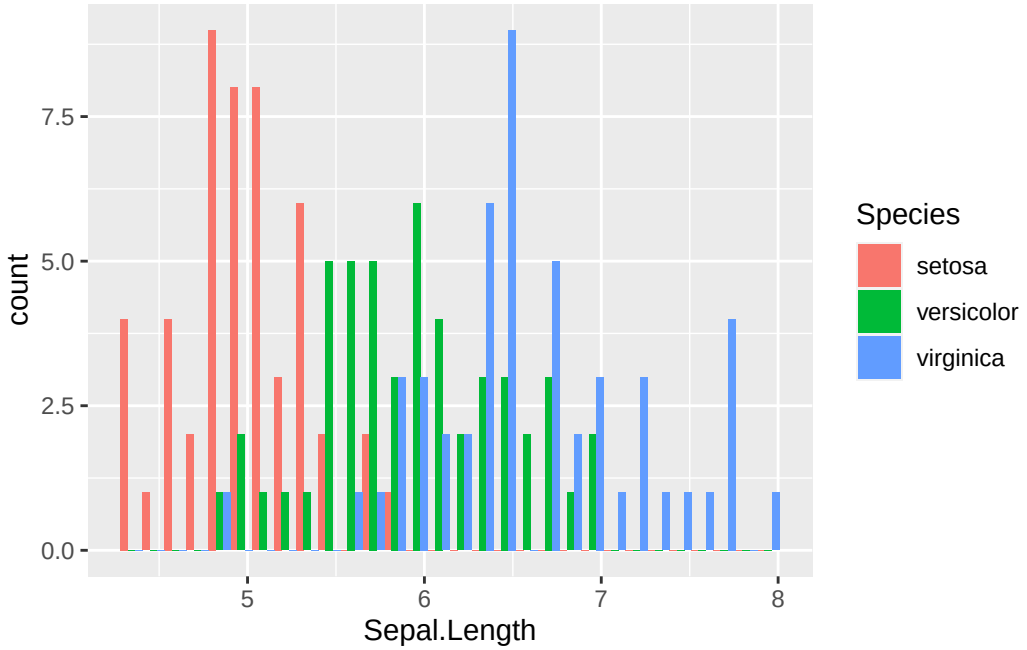


Abbildung 18.12.: Histogramm der `iris`-Daten mit farblich hervorgehobenen Spezieswerten

Die Ästhetik kann auch in den Geometriefunktionen angepasst werden. Wird hier für die Farbästhetik jeweils ein einzelner Wert angegeben, erzeugt `ggplot()` einen Vektor mit diskreten Werten und weist die Farben dynamisch zu.

```
Darstellungsbereich |>
    ggplot(aes(x)) +
        geom_line(stat = "function",
                  fun = f1,
                  aes(colour = "f1(x) = x ^ 2 - 3 * x")) +                    ①
        geom_line(stat = "function",
                  fun = f2,
                  aes(colour = "f2(x) = 4 * x + 2")) +                        ②
        labs(colour = "Funktion")
```

① Farbliche Hervorhebung der ersten Funktion
② Farbliche Hervorhebung der zweiten Funktion

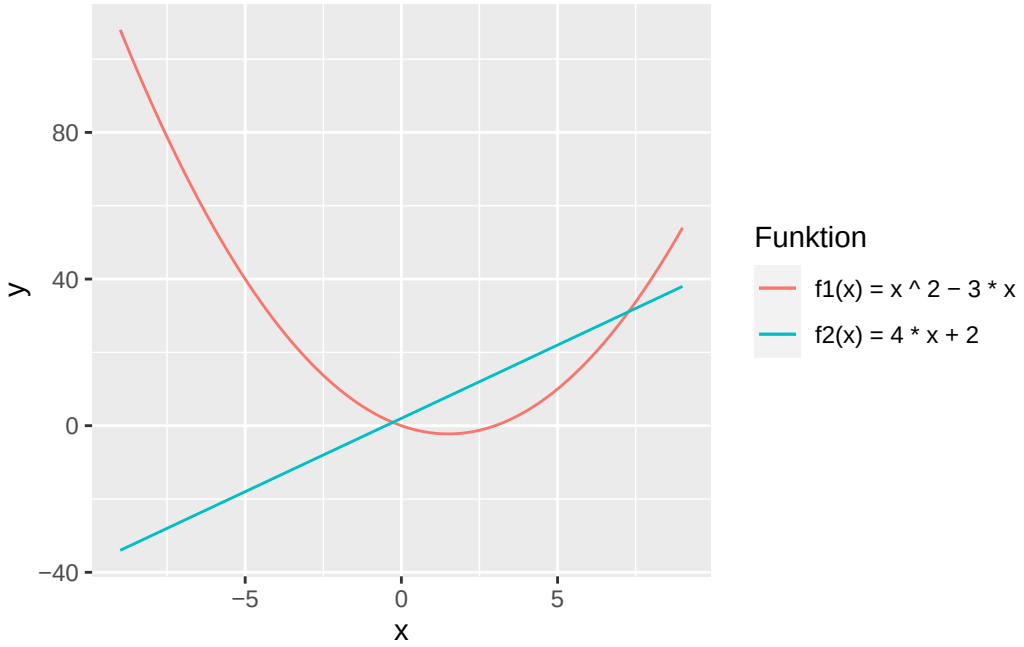


Abbildung 18.13.: Angepasste Ästhetik mit Einzelwerten in Geometriefunktionen

> **Warnung**
>
> `ggplot` unterstützt Farbcodes zur Konfiguration von Füll- und Linienfarben ausserhalb der normalen Ästhetikparameter. Mit diesen Farbcodes wird die normale Füll- oder Linienfarbe auf ***eine*** andere Farbe geändert. Diese Methode zur Anpassung von Farbe sollte **vermieden** werden, weil für solche Farben keine Legende erzeugt wird.

18.4.2. Grössenkodierung

> **Merke**
>
> Die Grössenkodierung eignet sich am Besten für *kontinuierliche Daten*.

> **Merke**
>
> Die Grössenkodierung lässt sich auf Linien oder Punkte anwenden. Flächen lassen sich nicht zusätzlich grössenkodieren, weil sie bereits Werte über die Grösse kodieren.

Die Grössenkodierung von Punkten erfolgt über die Ästhetik `size`. Auf diese Weise werden **Bubble-Charts** (Blasendiagramme) erstellt.

```
read_csv("geschlechter_schweizer_staedte.csv") ->
    geschlechteranteile_ch

geschlechteranteile_ch |>
    ggplot(aes(x = S, y = N, size = Gesamt)) +
        geom_point() +
        xlab("Männeranteil der ständigen Wohnbevölkerung") +
        ylab("Manneranteil der nicht-ständigen Wohnbevölkerung")
```

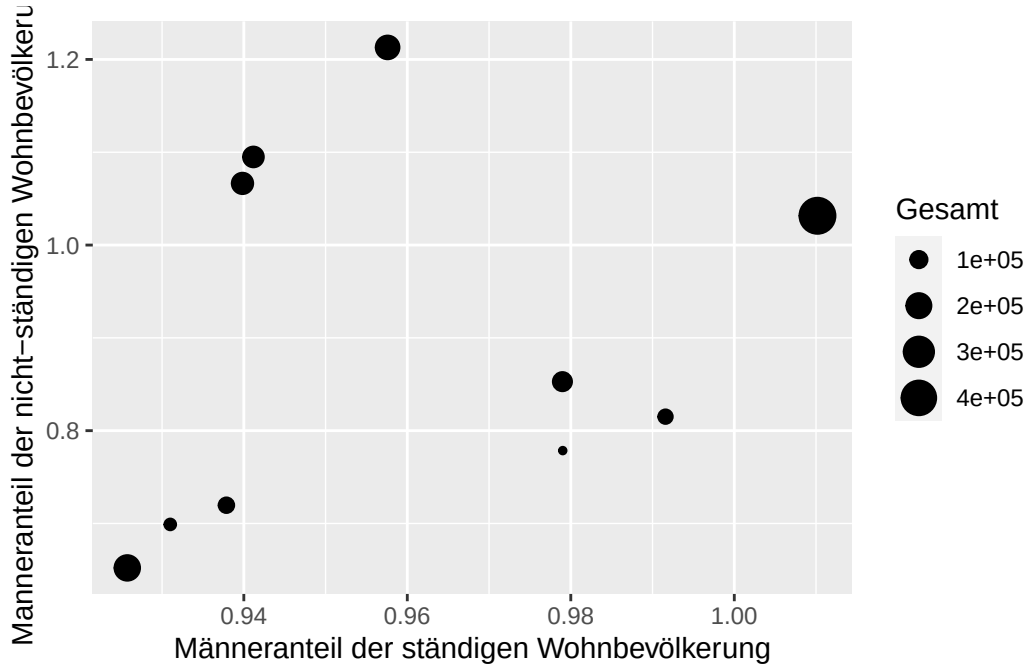


Abbildung 18.14.: Grössenkodierung am Beispiel des Männeranteils in Schweizer Städten

Die Grössenkodierung von Linien erfolgt über die Ästhetik `linewidth`. Die Linienbreite funktioniert analog zur Punktgrösse im Punktdiagramm.

> **Merke**
>
> Die Ästhetik `size` wird in Kombination mit `geom_point()`, `geom_text()` oder `geom_label()` verwendet.
> Die Ästhetik `linewidth` wird in Kombination mit `geom_line()` verwendet.

18.4.3. Formkodierung

Die Formkodierung kodiert die Werte einer Variable über die Form (engl. `shape`).

> **Merke**
>
> Die Formkodierung eignet sich nur für *diskrete Daten*!

> **Merke**
>
> Die Formkodierung kann nur ein Punkt- oder Jitter-Diagramm ergänzen.

Die Formkodierung erfolgt über die Ästhetik `shape`.

```
mtcars |>
    mutate(am = am |> factor()) |>
    ggplot(aes(
        x = mpg,
        y = hp,
        shape = am                                    (1)
    )) +
    geom_point()
```

(1) Formkodierung über das Merkmal `am`.

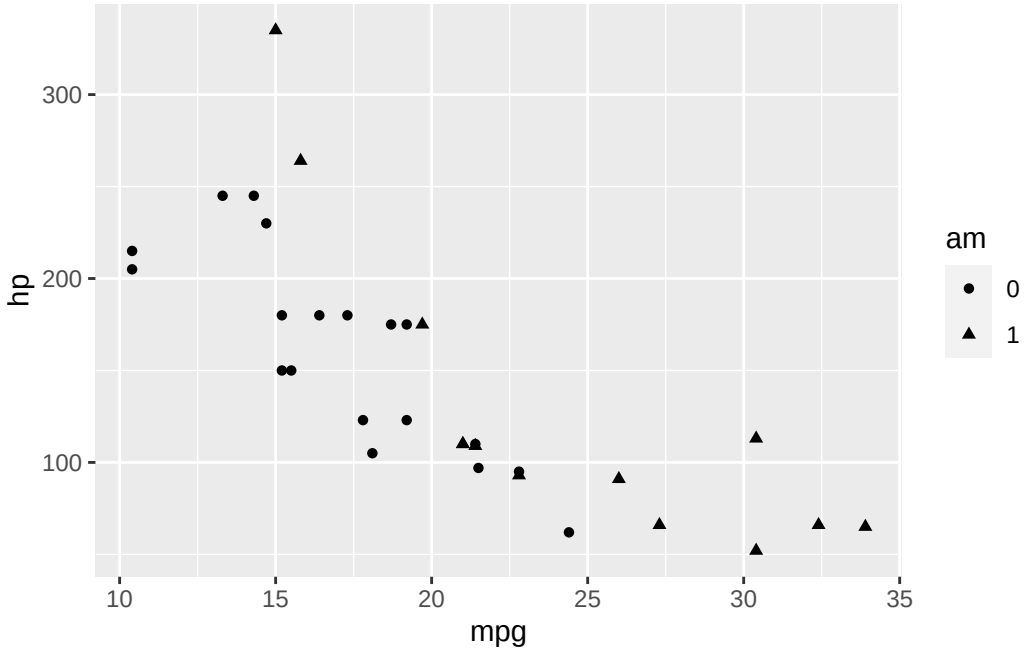


Abbildung 18.15.: Anwendung von `facet_wrap()`

Eine Variante der Formkodierung ist die Linienart. Die Kodierung der Linienart erfolgt über die Ästhetik `linetype`. Diese Formatierung prinzipiell mit allen Diagrammentypen

verwendet werden, die Linien darstellen. Für Boxplots oder Balkendiagramme sollte eine Kodierung über die Linienart vermieden werden.

```
iris |>
    ggplot(aes(x = Sepal.Length,
               y = Sepal.Width,
               linetype = Species)) +
        geom_point() +
        geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

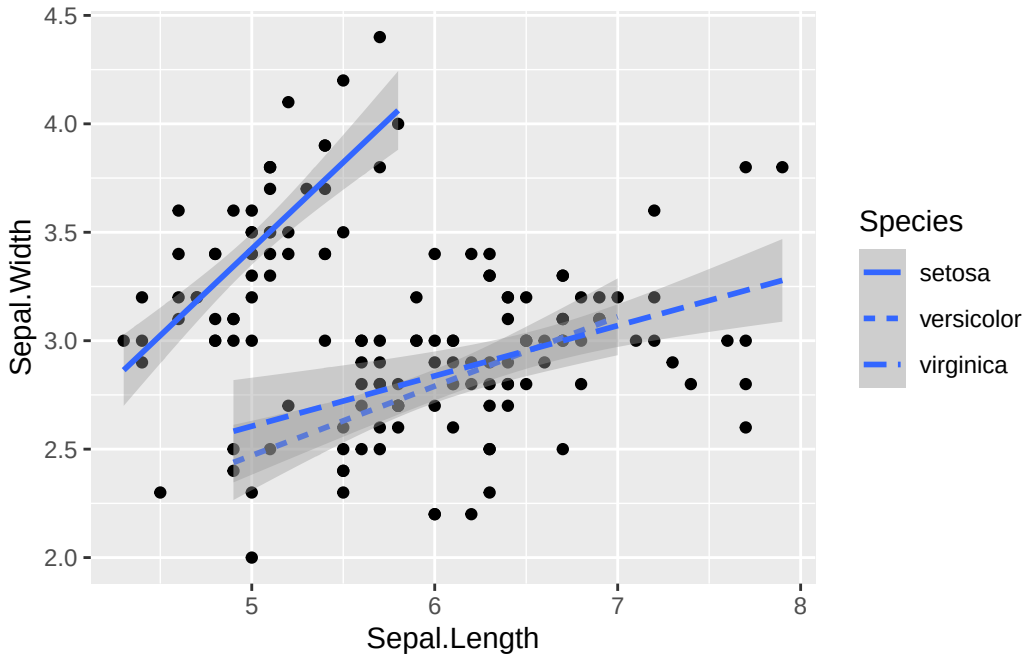


Abbildung 18.16.: Kodierung über Linienarten

Eine weitere Variante der Formkodierung sind Textmarkierungen. Textmarkierungen werden über die Ästhetik `label` zugewiesen. Für eine Textmarkierung wird der Wert des gewählten Merkmals im Diagramm angezeigt.

```
geschlechteranteile_ch |>
    ggplot(aes(x = S, y = N, label = Ort)) +
        geom_text() +
        xlab("Männeranteil der ständigen Wohnbevölkerung") +
        ylab("Manneranteil der nicht-ständigen Wohnbevölkerung")
```

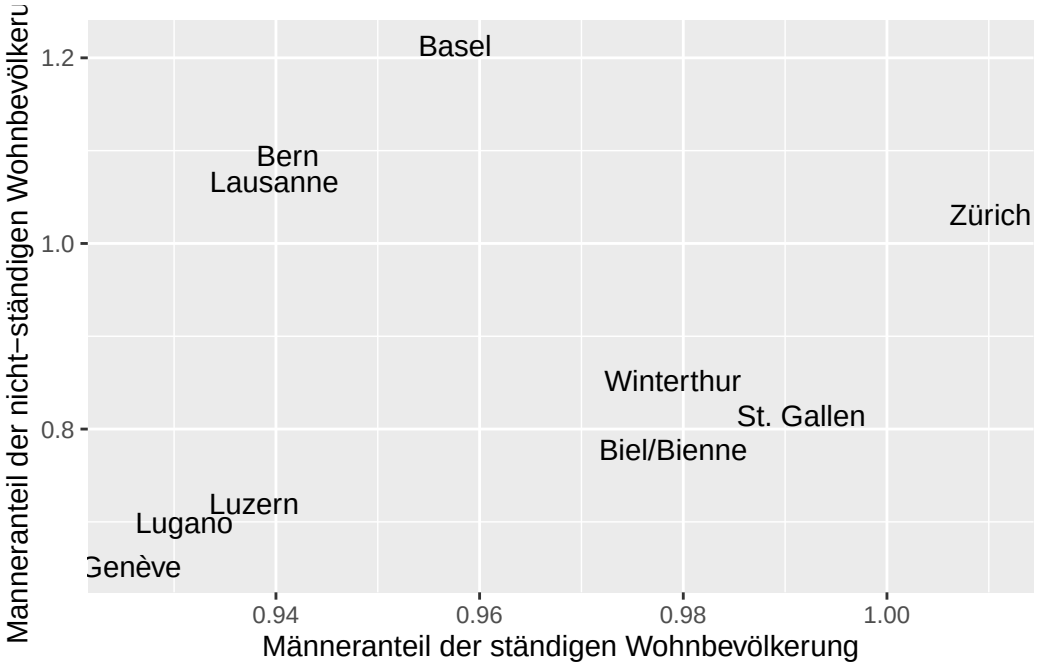


Abbildung 18.17.: Textmarkierungen über den Männeranteil an der ständigen und nicht-ständigen Wohnbevölkerung Schweizer Städte

18.4.4. Facetted Plots

Ein **Facetted Plot** gliedern die Darstellung von Werten in mehrere Teildiagramme. Diese Teildiagramme enthalten in der Regel nur die Werte

> **Merke**
>
> Facetten können nur mit *diskreten Daten* erstellt werden!

Facetten sind **keine** Ästhetik, sondern eine Steuerung.

```
mtcars |>
    ggplot(aes(mpg, hp)) +
    geom_point() +
    facet_wrap(~ gear)                                    ①
```

① Facetten folgen den Werten des Merkmals `gear`.

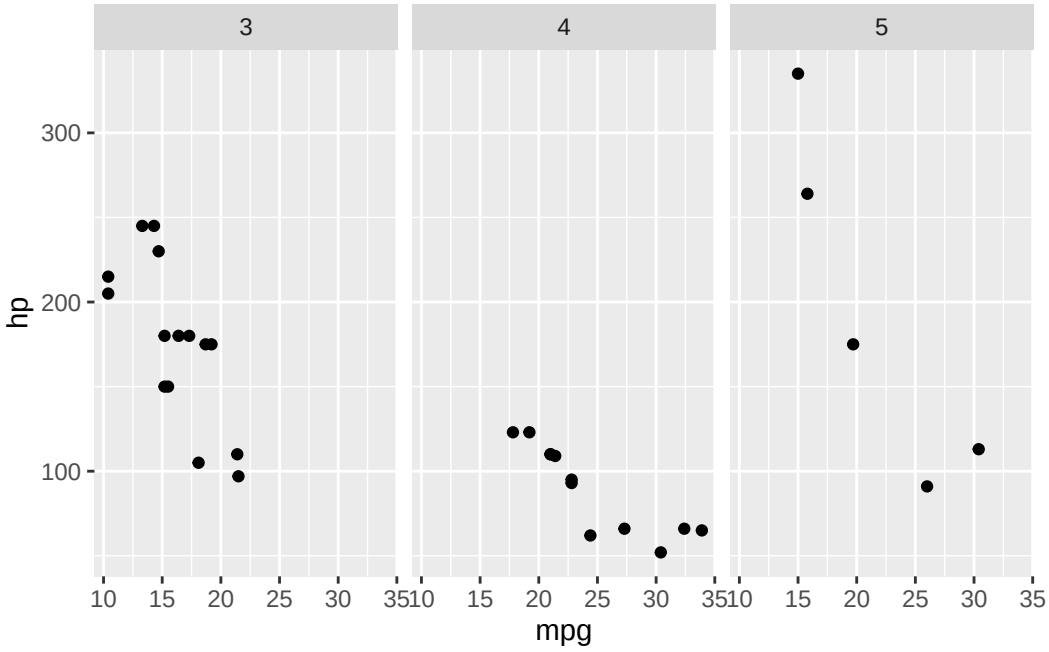


Abbildung 18.18.: Anwendung von `facet_wrap()`

Eine Variante von `facet_wrap()` ist `facet_grid()`. Diese Funktion erstellt eine visuelle Matrix entlang der Werte zweier diskreter Merkmale.

```
mtcars |>
    ggplot(aes(mpg, hp)) +
    geom_point() +
    facet_grid(carb ~ gear)                                    ①
```

① Facetten über die Werte der Merkmale `gear` (horizontal) und `carb` (vertikal).

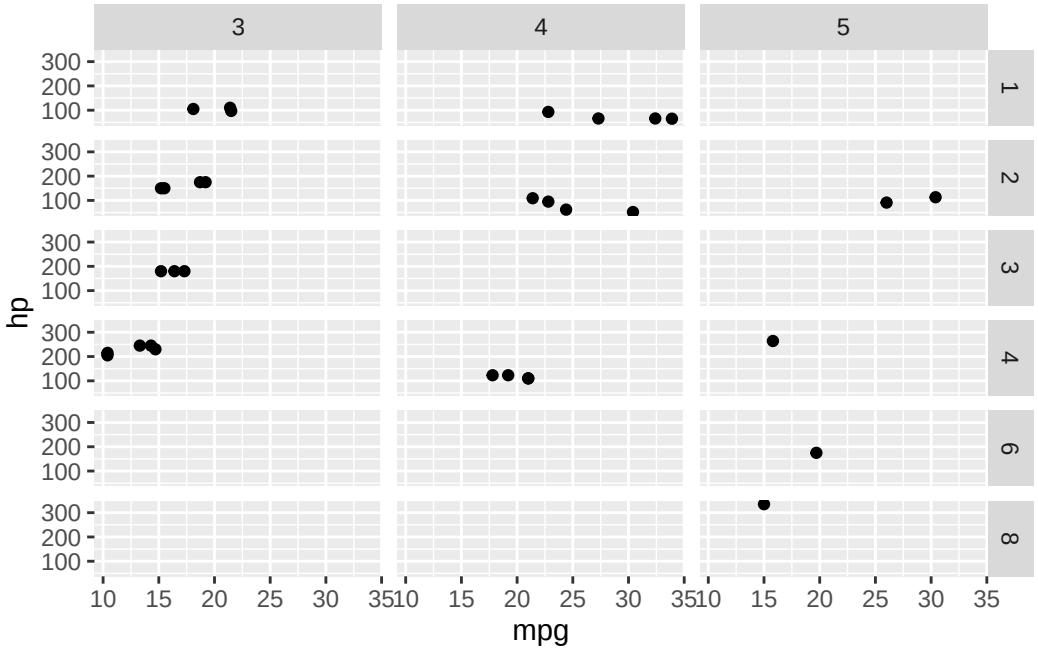


Abbildung 18.19.: Anwendung von `facet_grid()`

18.5. Spezielle Visualisierungen

18.5.1. Donut-Diagramme

> **!** Wichtig
>
> Tortendiagramme und Donut-Diagramme werden oft falsch interpretiert, weil Kreisflächen schwerer verglichen werden können als die Höhen von Balken. Sie sollten nur zur Illustration, aber nie zur Argumentation verwendet werden.

R kann auch Plots erstellen, die nur einen Datenvektor umfassen. In diesem Fall wird der zweite Vektor für die y-Achse aus den Werten des Vektors berechnet. Diese Möglichkeit haben wir schon bei der Erstellung von Histogrammen kennengelernt.

Nehmen wir das folgende Beispiel: Wir erstellen ein Stichprobenobjekt mit einem Vektor `q00_demo_gen`, der die Werte `1 : Keine Angabe`, `2 : Weiblich` und `3 : Männlich` enthält.

```
daten2 = tibble(
    q00_demo_gen = c("2 : Weiblich", "2 : Weiblich", "3 : Männlich",
"2 : Weiblich", "2 : Weiblich", "2 : Weiblich", "2 : Weiblich",
"2 : Weiblich", "2 : Weiblich", "3 : Männlich", "3 : Männlich",
"3 : Männlich", "3 : Männlich", "2 : Weiblich", "2 : Weiblich",
```

```
"3 : Männlich", "3 : Männlich", "2 : Weiblich", "3 : Männlich",
"2 : Weiblich", "2 : Weiblich", "2 : Weiblich", "2 : Weiblich",
"3 : Männlich", "2 : Weiblich", "3 : Männlich", "2 : Weiblich",
"3 : Männlich", "3 : Männlich", "2 : Weiblich", "2 : Weiblich",
"3 : Männlich", "2 : Weiblich", "3 : Männlich", "2 : Weiblich",
"3 : Männlich", "2 : Weiblich", "2 : Weiblich", "3 : Männlich",
"2 : Weiblich", "2 : Weiblich", "2 : Weiblich", "2 : Weiblich",
"3 : Männlich", "3 : Männlich", "2 : Weiblich", "2 : Weiblich",
"3 : Männlich", "3 : Männlich", "2 : Weiblich", "3 : Männlich",
"3 : Männlich", "2 : Weiblich", "2 : Weiblich", "2 : Weiblich",
"3 : Männlich", "2 : Weiblich", "3 : Männlich", "3 : Männlich",
"3 : Männlich", "2 : Weiblich", "3 : Männlich", "3 : Männlich",
"2 : Weiblich", "2 : Weiblich", "3 : Männlich", "2 : Weiblich",
"2 : Weiblich", "2 : Weiblich", "2 : Weiblich", "3 : Männlich",
"3 : Männlich", "3 : Männlich", "2 : Weiblich", "3 : Männlich",
"3 : Männlich", "3 : Männlich", "3 : Männlich", "3 : Männlich",
"3 : Männlich", "3 : Männlich", "3 : Männlich", "2 : Weiblich",
"3 : Männlich", "2 : Weiblich", "2 : Weiblich", "2 : Weiblich",
"2 : Weiblich", "3 : Männlich", "3 : Männlich", "3 : Männlich",
"3 : Männlich", "3 : Männlich", "3 : Männlich", "2 : Weiblich",
"2 : Weiblich", "3 : Männlich", "3 : Männlich", "2 : Weiblich",
"3 : Männlich", "3 : Männlich", "2 : Weiblich", "2 : Weiblich",
"2 : Weiblich", "3 : Männlich", "3 : Männlich", "2 : Weiblich",
"2 : Weiblich", "2 : Weiblich", "2 : Weiblich", "2 : Weiblich",
"2 : Weiblich", "2 : Weiblich", "2 : Weiblich", "3 : Männlich",
"3 : Männlich", "2 : Weiblich", "3 : Männlich", "2 : Weiblich",
"3 : Männlich", "3 : Männlich", "3 : Männlich", "3 : Männlich",
"3 : Männlich", "3 : Männlich", "3 : Männlich", "3 : Männlich",
"2 : Weiblich", "3 : Männlich", "3 : Männlich", "2 : Weiblich",
"3 : Männlich", "3 : Männlich", "2 : Weiblich", "3 : Männlich",
"2 : Weiblich", "3 : Männlich", "2 : Weiblich", "2 : Weiblich",
"3 : Männlich", "2 : Weiblich", "3 : Männlich", "3 : Männlich",
"2 : Weiblich", "3 : Männlich", "3 : Männlich", "3 : Männlich",
"3 : Männlich", "2 : Weiblich", "3 : Männlich", "2 : Weiblich",
"2 : Weiblich", "3 : Männlich", "2 : Weiblich", "2 : Weiblich",
"3 : Männlich", "2 : Weiblich", "3 : Männlich", "3 : Männlich",
"2 : Weiblich", "3 : Männlich", "3 : Männlich", "3 : Männlich",
"3 : Männlich", "2 : Weiblich", "2 : Weiblich", "2 : Weiblich",
"2 : Weiblich", "1 : Keine Angabe", "3 : Männlich", "3 : Männlich",
"2 : Weiblich", "2 : Weiblich", "2 : Weiblich", "2 : Weiblich",
"3 : Männlich", "3 : Männlich", "3 : Männlich", "3 : Männlich",
"3 : Männlich", "3 : Männlich", "3 : Männlich", "3 : Männlich",
```

```
"2 : Weiblich", "3 : Männlich", "2 : Weiblich", "3 : Männlich",
"2 : Weiblich")
)
```

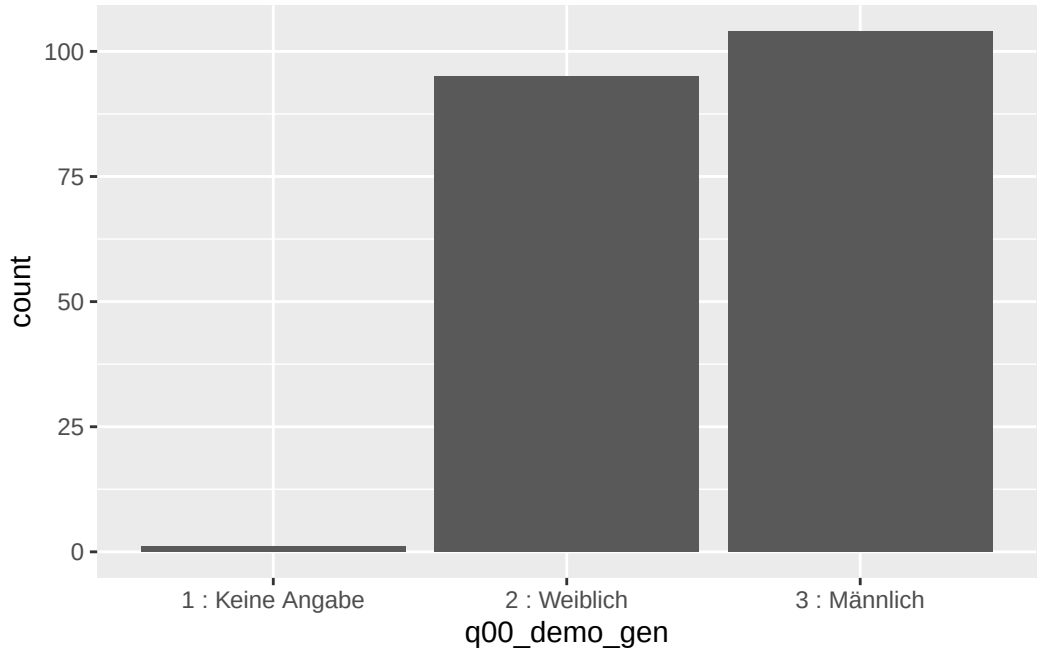
Uns interessiert nun: ***Wie oft kommen die drei möglichen Werte in unserer Stichprobe vor?***

Wir können die Werte mit `count()` selbst berechnen oder `ggplot` die Arbeit überlassen. Anstelle der `geom_col()`-Funktion verwenden wir nun die `geom_bar()`-Funktion. `geom_bar()` erwartet einen Vektor für die x-Achse und berechnet für die y-Achse das Auftreten der Werte, so wie wir es mit der `count()`-Funktion auch bestimmen würden.

```
daten2 |>
    count(q00_demo_gen)
```

```
# A tibble: 3 x 2
  q00_demo_gen          n
  <chr>             <int>
1 1 : Keine Angabe      1
2 2 : Weiblich         95
3 3 : Männlich        104
```

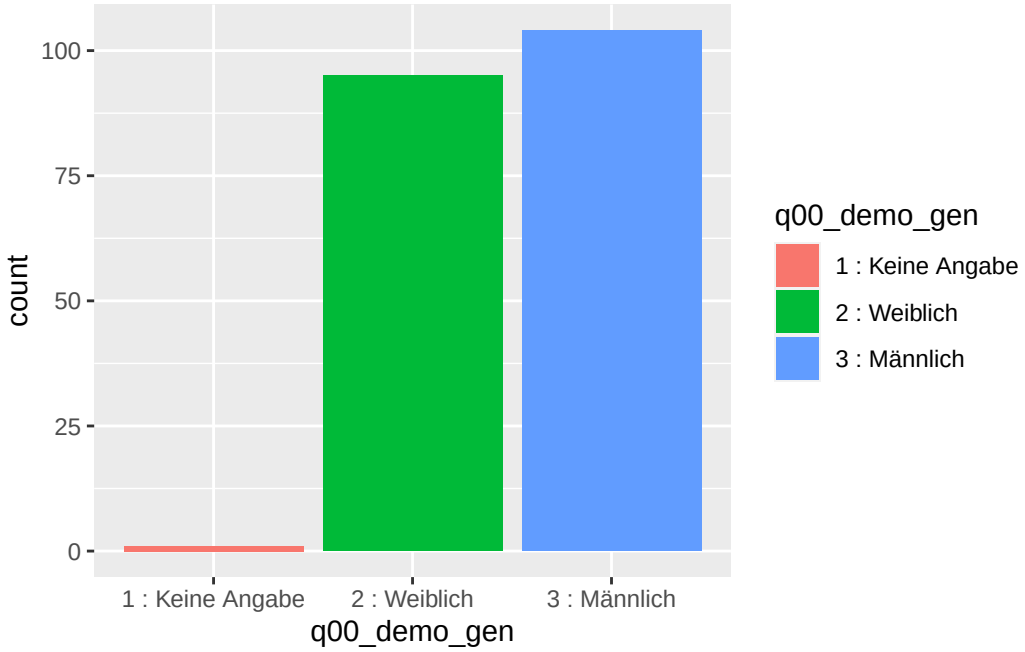
```
daten2 |>
    ggplot(aes(x = q00_demo_gen)) +
        geom_bar()
```



Mit diesem Plot können wir die Unterschiede in unserer Werteverteilung leichter erkennen.

Für Präsentationen ist so ein Plot aber nicht wahnsinnig attraktiv. Färben wir den Plot also ein. Das machen wir, indem wir den Vektornamen auch für die Füllung der Balken verwenden. `ggplot` wählt nun für jeden Wert in diesem Vektor eine eigene Farbe aus. Dadurch färben sich unsere Balken ein.

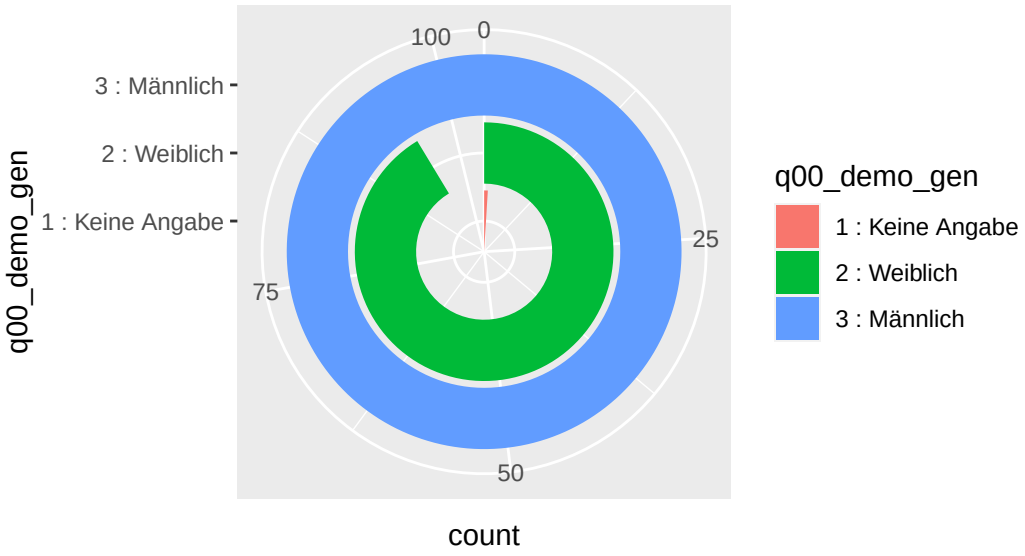
```
daten2 |>
    ggplot(aes(x=q00_demo_gen, fill = q00_demo_gen)) +
        geom_bar()
```



Für Histogramme verwenden wir normalerweise ein kartesisches Koordinatensystem. Wir können aber auch ein anderes Koordinatensystem wählen. Eine Variante sind **polare Koordinaten**. Mit einem polaren Koordinatensystem erreichen wir kreisförmige Darstellungen. Wir müssen dazu die Dimension, die auf das Zentrum des Diagramms zeigt, festlegen und als Parameter übergeben. In unserem Fall ist das die y-Dimension.

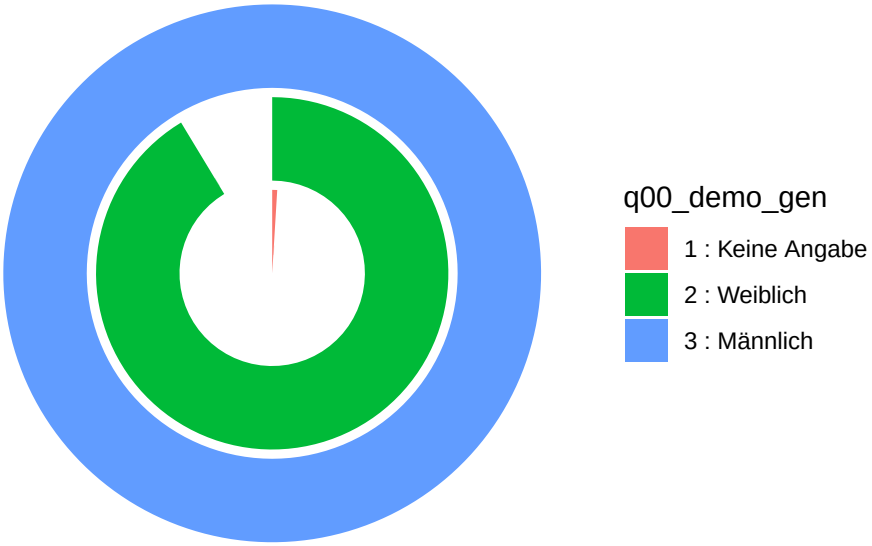
Wir stellen unsere Daten in einem polaren Koordinatensystem dar, indem wir mit der `coord_polar()`-Funktion ggplot mitteilen, dass wir ein anderes Koordinatensystem wünschen.

```
daten2 |>
    ggplot(aes(x=q00_demo_gen, fill = q00_demo_gen)) +
        geom_bar() +
        coord_polar("y")
```



Unser Plot hat jetzt unschöne Beschriftungen. Die werden wir mit einem Formatierungsthema los. `ggplot` hat verschiedenen Formatierungen als Thema vordefiniert. Eines davon ist das Thema `void`. Diese Formatierung entfernt alle Hintergründe, Achsen und Beschriftungen ausser Legenden.

```
daten2 |>
    ggplot(aes(x=q00_demo_gen, fill = q00_demo_gen)) +
        geom_bar() +
        coord_polar("y") +
        theme_void()
```



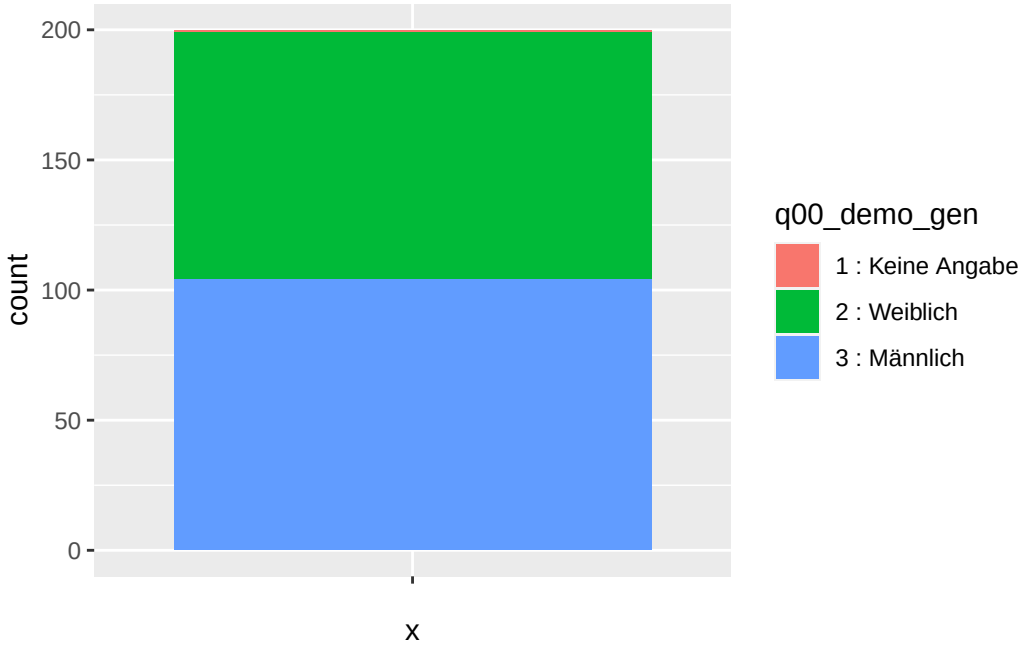
Das sieht doch gleich viel besser aus.

> **Hinweis**
>
> Solche ringartigen Visualisierungen werden als Zielscheiben- oder **Donut-Diagramme** bezeichnet.

18.5.2. Torten-Diagramme

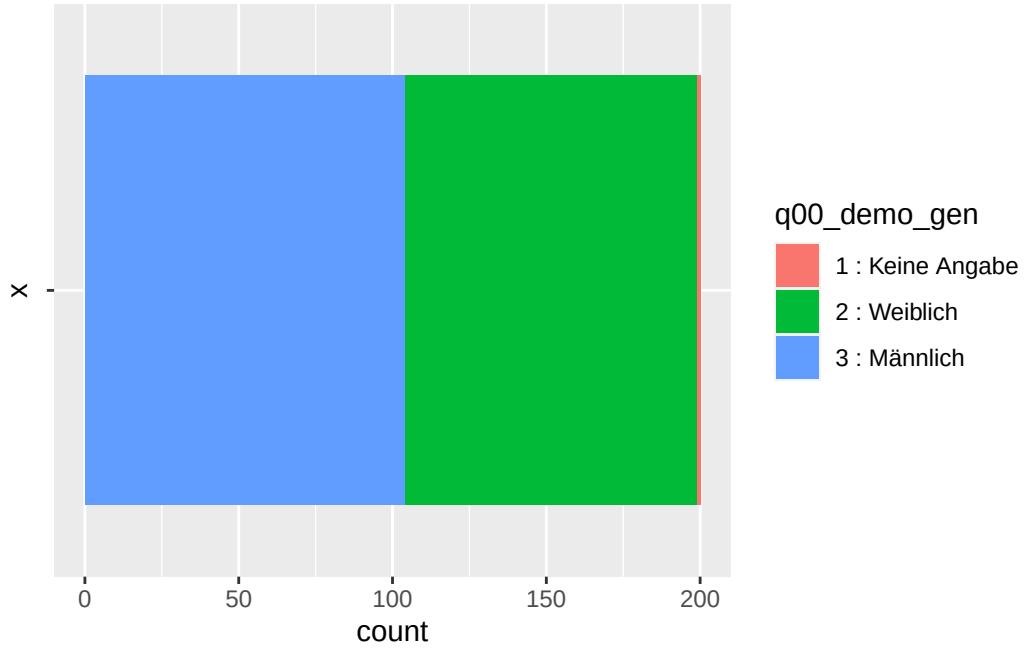
Für Torten- bzw. Kreisdiagramme müssen alle Balken eines Balkendiagramms übereinander gestapelt werden. Das erreichen wir, indem wir für die x-Achse einen konstanten Wert angeben. Z.B. nehmen wir dazu die leere Zeichenkette. So werden merkwürdige Beschriftungen im Diagramm vermieden. Damit wir die Balken auseinanderhalten können, färben wir sie ein.

```
daten2 |>
    ggplot(aes(x ="", fill = q00_demo_gen)) +
        geom_bar()
```



Mit der Funktion `coord_flip` werden die Achsen vertauscht und das Diagramm gedreht.

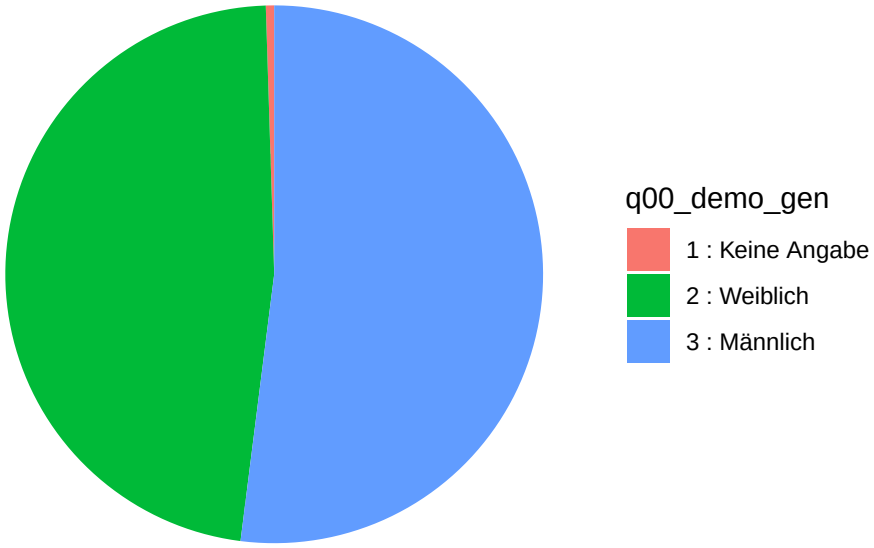
```
daten2 |>
    ggplot(aes(x ="", fill = q00_demo_gen)) +
        geom_bar() +
        coord_flip()
```



Wir erkennen nun deutlich, dass `ggplot` immer versucht möglichst viel Fläche zu nutzen.

Das Interessante an dieser Darstellung ist aber nicht dieses Format, sondern dass wir dieses Diagramm ebenfalls in einem polaren Koordinatensystem darstellen können.

```
daten2 |>
    ggplot(aes(x ="", fill = q00_demo_gen)) +
        geom_bar() +
        coord_polar("y") +
        theme_void()
```



Auf diese Weise erzeugen wir Tortendiagramme.

Referenzen

American Mathematical Society, & LATEX Project. (2020). *User's Guide for the amsmath Package*. http://mirrors.ctan.org/macros/latex/required/amsmath/amsldoc.pdf

Bates, D., Maechler, M., Jagan, M., Davis, T. A., Oehlschlägel, J., & Riedy, J. (2023). *R Package 'Matrix'* (Version 1.6-3). https://cran.r-project.org/package=Matrix

Grolemund, G. (2014). *Introduction to R Markdown*. https://rmarkdown.rstudio.com/articles_intro.html

Høgholm, M., & Madsen, L. (2022). *mathtools – Mathematical tools to use with amsmath*. https://ctan.org/pkg/mathtools?lang=en

Jupyter Development Team. (2015). *The Notebook file format*. https://nbformat.readthedocs.io/en/latest/format_description.html

Posit Software PBC. (2023). *quarto*. https://quarto.org/

Team, R. C., Maechler, M., Gentleman, R., Ihaka, R., Venables, W. N., & Smith, D. M. (2023). *An Introduction to R*. https://cran.r-project.org/doc/manuals/R-intro.html

The Jupyter Book Community. (2023). *Jupyter Book*. https://jupyterbook.org

Wickham, H. (2023a). *forcats: Tools for Working with Categorical Variables (Factors)*. https://forcats.tidyverse.org/

Wickham, H. (2023b). *stringr: Simple, Consistent Wrappers for Common String Operations*. https://stringr.tidyverse.org

Wickham, H., François, R., Henry, L., Müller, K., & Vaughan, D. (2023). *dplyr: A Grammar of Data Manipulation*. https://dplyr.tidyverse.org

Wickham, H., Vaughan, D., & Girlich, M. (2023). *tidyr: Tidy Messy Data*. https://tidyr.tidyverse.org