Einführung in die Datenwissenschaft mit R

Inhaltsverzeichnis

Vorwort **3**

Copyright **4**

1. Einleitung **5**
1.1. Motivation und Ausgangslage . . . 5
1.2. Base R und Tidy R . . . 5
1.3. Organisation dieses Buchs . . . 6

2. Tool Chain **7**
2.1. R installieren . . . 7
2.1.1. Überprüfen der Installation . . . 10
2.1.2. Erste Schritte . . . 11
2.2. Grafische Oberflächen für R . . . 12
2.2.1. RStudio . . . 13
2.2.2. JupyterLab . . . 13
2.2.3. Visual Studio Code . . . 14
2.3. R-Bibliotheken installieren . . . 14

3. Hilfe bekommen **16**
3.1. `help()` . . . 16
3.1.1. Aufbau von Funktionsdokumentationen . . . 17
3.2. Vignettes . . . 19
3.3. Cheat Sheets . . . 19

4. R-Sprachelemente **22**
4.1. Syntaktische Symbole . . . 22
4.1.1. Werte . . . 22
4.1.2. Schlüsselworte . . . 22
4.1.3. Bezeichner . . . 23
4.2. Operationen . . . 24
4.2.1. Operatoren . . . 24
4.2.2. Blöcke . . . 26

I. Datenquellen **27**

5. Dokumentation **28**
5.1. Mathematische Formeln in Datendokumenten 30

6. Datentypen **32**
6.1. Fundamentale Datentypen 32
6.1.1. Undefinierte Werte 32
6.1.2. Zahlen 33
6.1.3. Zeichenketten 34
6.1.4. Wahrheitswerte 35
6.1.5. Faktoren 36
6.2. Datenstrukturen 36
6.2.1. Vektoren 37
6.2.2. Listen 38
6.2.3. Matrizen 40
6.2.4. Data-Frames 42

7. Importieren und Exportieren **44**
7.1. Daten importieren 44
7.1.1. Dateitypen 44
7.1.2. Dateien mit einer Spalte 46
7.1.3. Excel Arbeitsmappen 46
7.2. Daten exportieren 49
7.3. JSON-Daten 50
7.4. Festkodierte Daten 51

II. Mathematik der Daten **52**

8. Variablen, Funktionen und Operatoren **53**
8.1. Variablen 53
8.2. Funktionen 54
8.2.1. Operatoren 54
8.2.2. Funktionsketten 57
8.3. Eigene Funktionen erstellen 57
8.3.1. Parameter und Variablen 59
8.3.2. Datentypen überprüfen 59
8.3.3. Nebeneffekte 59
8.4. Bibliotheken 61
8.5. Bibliotheksmanagement 63
8.5.1. Projektvorbereitung 64
8.5.2. Bibliotheken installieren 64

8.5.3. Updates für Bibliotheken 64

9. Zeichenketten **65**

10. Faktoren **66**

11. Boole'sche Operationen **67**

12. Vektoroperationen **68**

13. Matrix-Operationen **69**

14. Indizieren und Gruppieren **70**
14.1. Indizieren 70
14.2. Gruppieren 70

15. Daten kodieren **71**
15.1. Mit Faktoren kodieren 71

16. Daten formen **72**

III. Deskriptive Datenanalyse **73**

17. Daten beschreiben **74**

18. Daten visualisieren **75**

Referenzen **76**

Vorwort

Work in Progress

Copyright

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).

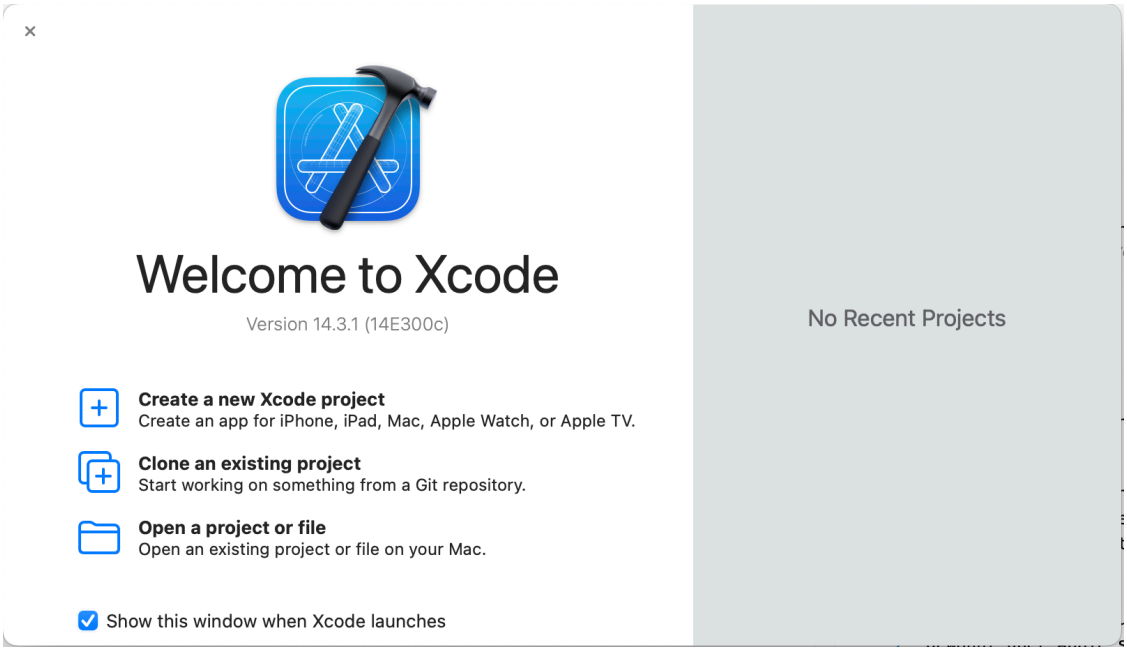

Abbildung 2.1.: XCode Start Dialog

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 Abgragen 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 Lizensbedingungen 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 Versionsinformationen 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 zustätzliche Werkzeuge und Ansichten.

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

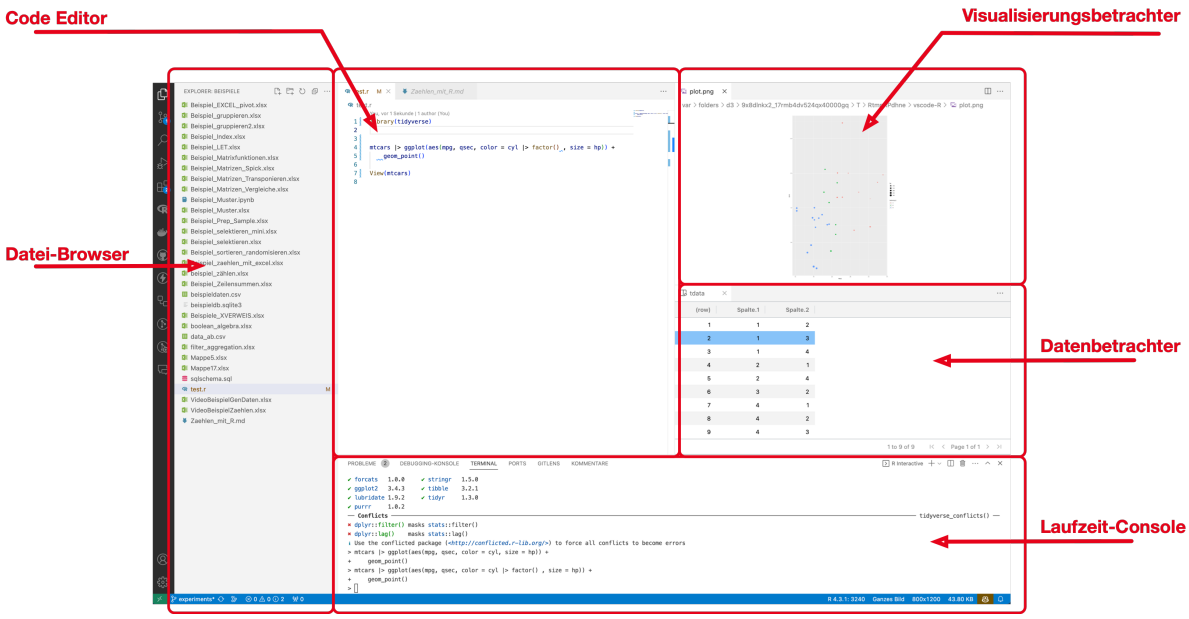

Abbildung 2.3.: Komponenten einer datenzentrischen Entwicklungsumgebung am Beispiel von Visual Studio Code

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.

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).

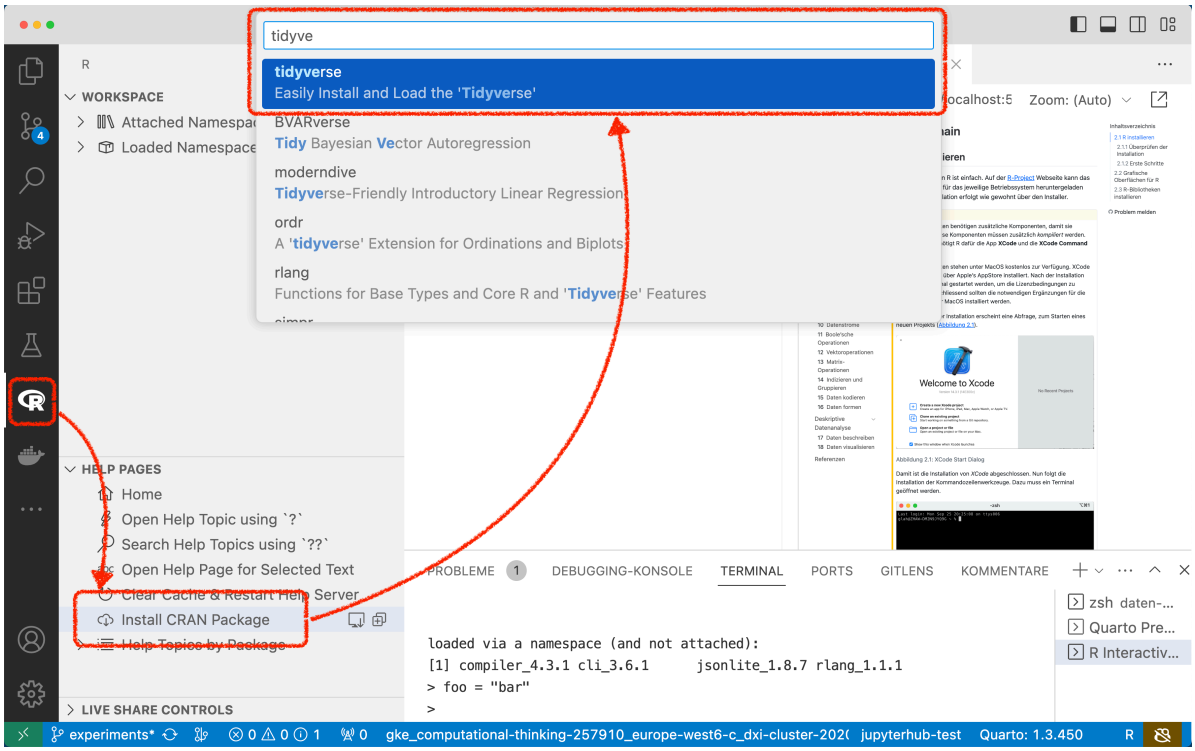

Abbildung 2.4.: Installation der `tidyverse`-Bibliothek in Visual Studio Code

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.

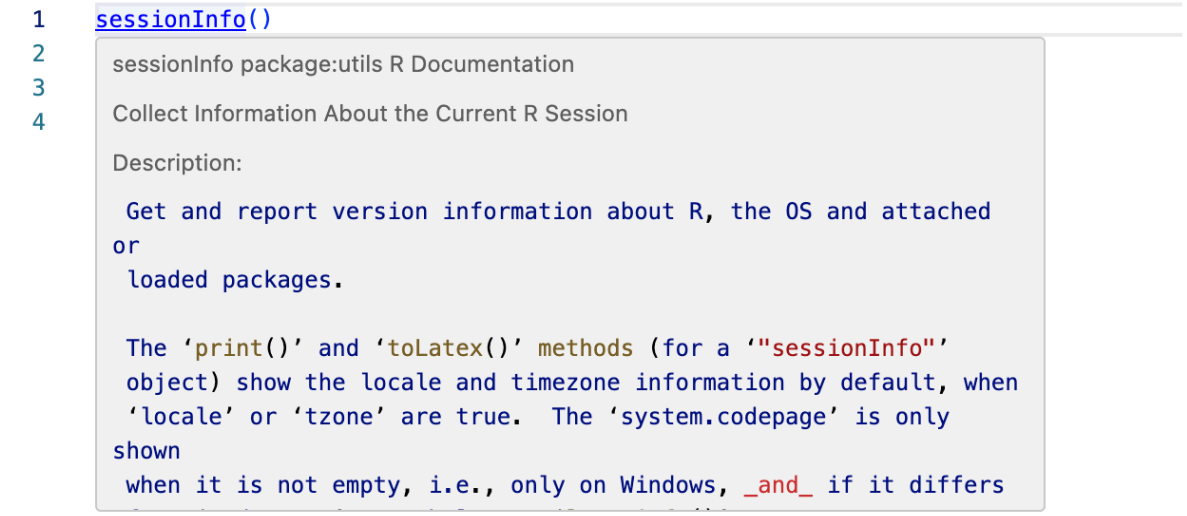

Abbildung 3.1.: Inline Anzeige einer R-Funktionsdokumentation in Visual Studio Code

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:

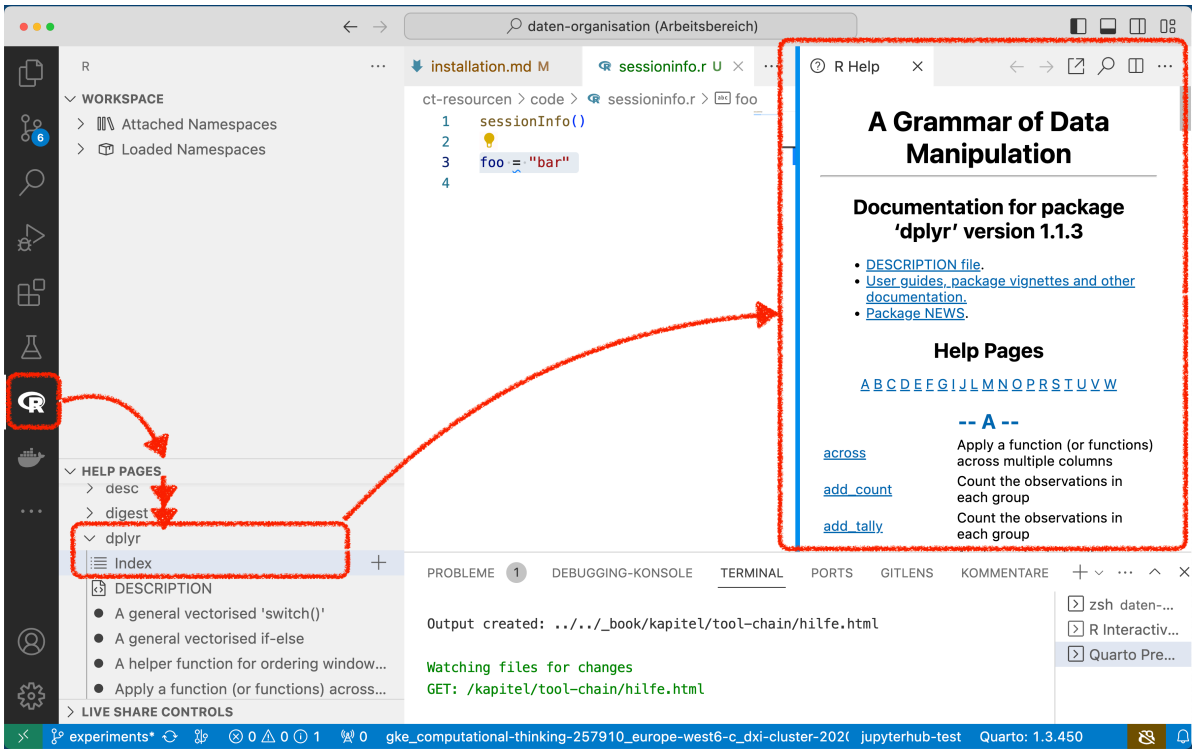

Abbildung 3.2.: Index der Dokumentation für die Bibliothek `dplyr`

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

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.

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

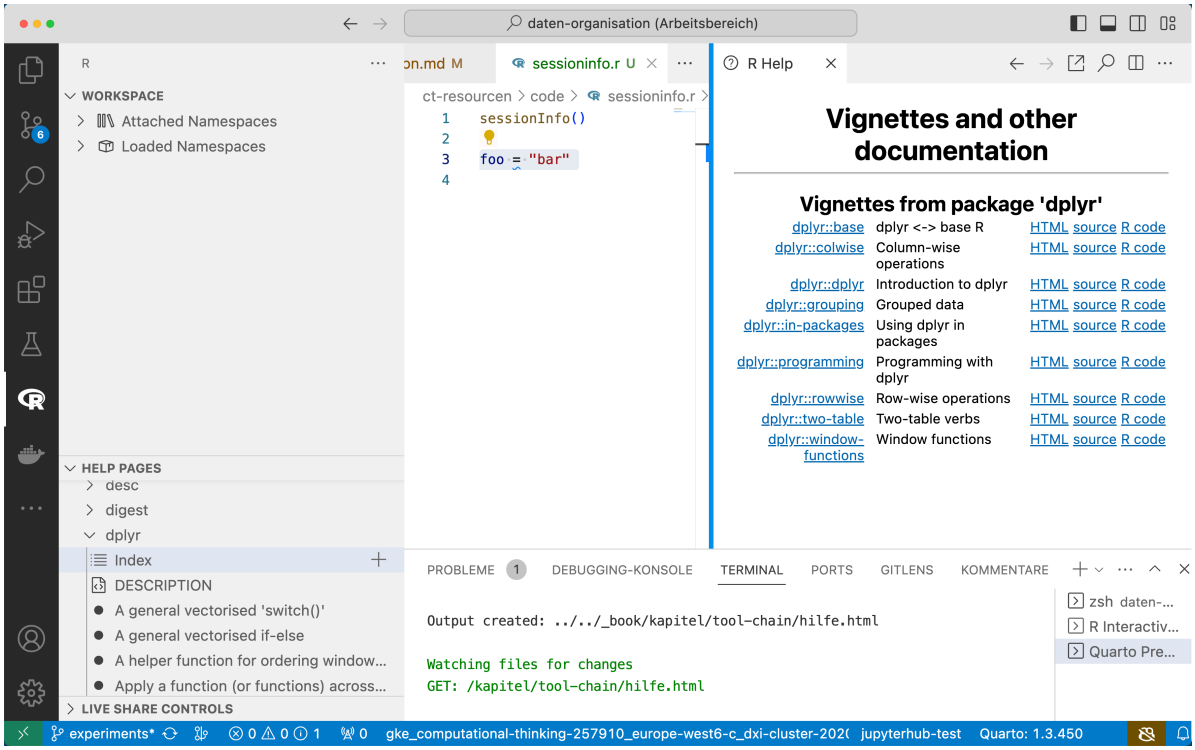

Abbildung 3.3.: Index der `dplyr` Anleitungen

> 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 "merwürdiger Name"`.

4.2. Operationen

Definition 4.1. Eine **Operation** ist ein syntaktisches Konstrukt, das von einer Programmiersprache als ausgeführbar 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 beiden 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"}
```

```r
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)**.**

```r
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)**.**

```r
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)**.**

```r
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)**.**

```r
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.

```r
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 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 einem Vektor erstellen)**.**

```r
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)**.**

```
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_delim()` oder `read_csv2()`	`read.csv2()`

Format	tidy R	Base R
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 = local(decimal_mark = "."))
> ```
>
> **Dieser Aufruf funktioniert jedoch nicht!**
> Hier greift die Funktion `read_delim()`. Wird dieser Funktion nur ein Dateiname übergeben, dann prüft die Funktion auf die verschiedenen Trennzeichen. Dieser (undokumentierte) Algorithmus erkennt Schweizer CSV-Dateien korrekt.
>
> ```
> read_delim(datei_name)
> ```
>
> **Dieser Aufruf importiert die Werte wie erwartet.**

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. Die Funktion `read_delim()` liest diese Dateien meistens korrekt ein. 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.

> **i** 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.

> **i** 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_delim("Bestellungen_Excel.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, muss die gesamte Datei zuerst eingelesen werden und dann an den JSON-Parser `fromJSON()` übergeben werden.

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

```
library(jsonlite)

Daten = fromJSON(read_file("beispiele/daten.json"))
```

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

```
write_file(toJSON(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. 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.

> **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 R gibt eine direkte Funktionsdefinition wie jeden anderen Wert direkt aus.

8.2.1. 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 8.1.: 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
`%/%`	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

Operator	Beschreibung	Art
`>=`	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
`?`	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 8.2 (Exakte *lineare* Beziehung zwischen Daten)**.**

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

Beispiel 8.3 (Beziehung zwischen Daten mit Interaktion als Modell)**.**

```
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 8.4 (+-Operator als Funktionsbezeichner)**.**

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

```
3
```

8.2.1.1. Zuweisungsoperatoren

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 8.2.2) eingesetzt.

8.2.1.2. Ausführungsoperator

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.

8.2.1.3. 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 8.5 (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 8.6 (Dokumentationssuche nach Operatoren)**.**

```
??operator
```

8.2.2. 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 8.7 (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.

8.3. 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 8.8 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 8.8 (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.

Beispiel 8.9 weist der Funktion aus Beispiel 8.8 den Bezeichner `quadrat_minus_eins` zu. Dieser Bezeichner kann anschliessend als Funktion verwendet werden (s. Beispiel 8.10).

Beispiel 8.9 (Eine Funktion mit Bezeichner deklarieren)**.**

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

Beispiel 8.10 (Eine selbstdeklarierte Funktion aufrufen)**.**

```
quadrat_minus_eins(2)
```

3

8.3.1. 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.

8.3.2. 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 8.11).

Beispiel 8.11 (Eine Funktion mit Typenprüfung deklarieren)**.**

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

8.3.3. 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 8.12 (Geltungsbereich von Variablen in Funktionen)**.**

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

# Anwendung
f(2)
var1
```

```
3 # Ergebnis von f(2)
1 # Ergebnis von var1
```

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

Beispiel 8.13 (Funktion mit Nebeneffekt)**.**

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

# Anwendung
f(2)
var1
```

```
3 # Ergebnis von f(2)
3 # Ergebnis von var1
```

> **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***.

8.4. 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 8.14 (Funktionen mit der `library()` Funktion einbinden)**.**

```
library(ggplot2)

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 8.15 (Eine Funktion direkt ansprechen)**.**

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

> **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 8.16 zeigt wie die Funktion `read_delim()` direkt angesprochen werden kann, wenn die `tidyverse`-Bibliotheken nicht mit `library(tidyverse)` eingebunden wurden. `read_delim()` wird in der Bibliothek `readr` definiert. Entsprechend kann die Funktion nur über `readr::read_delim()` aufgerufen werden.

Beispiel 8.16 (Funktion aus Unterbibliothek direkt ansprechen)**.**

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

8.5. Bibliotheksmanagement

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. 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.

8.5.1. 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.

8.5.2. 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.

8.5.3. Updates für Bibliotheken

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.

> **i** 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

> ⚠ Work in Progress

Zeichenketten zerlegen

String R

10. Faktoren

⚠ Work in Progress

Definition 10.1.

11. Boole'sche Operationen

Work in Progress

12. Vektoroperationen

Work in Progress

13. Matrix-Operationen

Work in Progress

14. Indizieren und Gruppieren

14.1. Indizieren

14.2. Gruppieren

15. Daten kodieren

Kodierungstabellen

15.1. Mit Faktoren kodieren

16. Daten formen

Work in Progress

Teil III.

Deskriptive Datenanalyse

17. Daten beschreiben

18. Daten visualisieren

Work in Progress

Referenzen

American Mathematical Society, & LATEX Project. (2020). *User's Guide for the amsmath Package*. http://mirrors.ctan.org/macros/latex/required/amsmath/amsldoc.pdf

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/

The Jupyter Book Community. (2023). *Jupyter Book*. https://jupyterbook.org