

Computational Methods in der politischen Kommunikationsforschung

Methodische Vertiefung: Computational Methods mit R und
RStudio

Julian Unkel

Contents

Einführung	5
Seminarinformationen	5
Ablauf des Kurses	5
Motivation und Ziele des Seminars	6
Hinweise zur Nutzung des Online-Kurses	8
I Eine kurze Einführung in R	11
1 Installation und erste Schritte	13
1.1 R installieren	13
1.2 RStudio installieren	13
1.3 Die Benutzeroberfläche von RStudio	14
1.4 RStudio anpassen	17
1.5 Übungsaufgaben	20
2 Objekte und Datenstrukturen	21
2.1 Objektnamen	23
2.2 Objekttypen	24
2.3 Datenstrukturen	27
2.4 Übungsaufgaben	40

Einführung

Zuletzt aktualisiert: 2020-04-22 08:40:44. Dies ist ein *Work-in-Progress* und wird laufend aktualisiert.

Seminarinformationen

- Dozent: Julian Unkel, unkel@ifkw.lmu.de
- Zeit und Ort: Donnerstags, 12-14 Uhr, Øe 057 (bis auf weiteres findet das Seminar rein digital statt)
- Moodle: <https://moodle.lmu.de/course/view.php?id=8250>
- Zoom: <https://lmu-munich.zoom.us/j/6679571494>

Ablauf des Kurses

Aufgrund der aktuellen Situation wird dieses Seminar in einen Online-Kurs überführt. Alle Seminarinhalte werden in Textform aufbereitet und nach und nach diesem Online-Kurs hinzugefügt. Auf Basis des Kurses sollen die Seminarinhalte selbstständig und mit weitestgehend eigenem Lerntempo erarbeitet werden.

In jedem Kapitel werden hierzu zunächst die wesentlichen Konzepte und Inhalte erläutert. Jedes Kapitel schließt mit einigen Übungsaufgaben, die über Moodle abgegeben werden müssen. Deadlines für die Übungsaufgaben werden ebenfalls über Moodle kommuniziert, Lösungen im Anschluss an die Deadlines im Kurs hinzugefügt.

Jeden Donnerstag zum regulären Seminartermin findet von 12-14 Uhr eine Online-Sprechstunde via Zoom statt. Hier können Fragen zu den Seminarinhalten, Übungsaufgaben etc. gestellt und diskutiert werden.

In Moodle stehen zudem zwei Foren zur Verfügung, in dem Sie 1) allgemeine Fragen zu R und RStudio sowie 2) spezifische Fragen / alternative Lösungen zu den Übungsaufgaben (vor-)stellen und diskutieren können. Scheuen Sie sich bitte nicht, auch selbst auf Fragen und Probleme von Kommiliton*innen einzugehen.

Neben den regulären Übungsaufgaben werden Sie bisweilen auch optionale,

besonders knifflige Aufgaben vorfinden, die ich in der Sprache meiner Ahnen als *Käpseles-Aufgaben* kennzeichnen werde. Diese sind nicht verpflichtend, können Ihnen aber als Gradmesser dienen, ob Sie die jeweiligen Inhalte auch eigenständig und in leicht abgewandelter Form anwenden können.

Motivation und Ziele des Seminars

Das Ziel des Kurses ist es, methodische Kenntnisse zur Anwendung computationaler Methoden zu vermitteln. Hierzu werden wir uns zunächst allgemein mit der Datenbearbeitung und -analyse mit der statistischen Programmiersprache R auseinandersetzen. Es folgen dann spezifischere Verfahren der computationalen Datenerhebung und -analyse.

Dabei stehen insbesondere folgende Inhalte im Vordergrund:

- Einführung in *R* und die Arbeit mit *RStudio*
- Datenmanagement in *R*
- Computationale Datenerhebung mit *R*
- Datenvisualisierung mit *R*
- Automatisierte Inhaltsanalyse mit *R*

Zudem wird darauf eingegangen, wie mittels R und RStudio Kommunikationsforschung transparent, nachvollziehbar und reproduzierbar gestaltet werden kann.

Es werden keine Vorkenntnisse in R vorausgesetzt; die Inhalte der Veranstaltung *15424 Datenanalyse* werden als bekannt vorausgesetzt.

Bevor es jedoch ans Eingemachte geht, ein paar Worte zur Motivation hinter diesem Seminar: Warum lohnt es sich überhaupt, eine Programmiersprache für die quantitativ-wissenschaftliche Arbeit zu lernen? Und warum ausgerechnet R?

Warum also eine Programmiersprache für Datenanalyse lernen?

Wenn Sie bisher Daten statistisch ausgewertet haben, etwa im Rahmen von Forschungsseminaren oder der Bachelorarbeit, wird das in der Regel mit einem Programm mit grafischer Oberfläche erfolgt sein, etwa mit *Microsoft Excel* oder mit *IBM SPSS*. Diese Programme haben viele Vorteile: sie sind meist auf spezifische Funktionen zugeschnitten, in ihrer Aufmachung an typische Computersoftware angepasst und entsprechend intuitiv zu bedienen - ein paar Klicks, und schon gibt SPSS eine Regressionstabelle mit allen relevanten Informationen aus. Für die meisten Anwendungsfälle im KW-Studium bieten genannten Programme leicht zu erlernende und umzusetzende Lösungen an. Programmiersprachen haben hingegen eine zweifellos höhere Einstiegshürde, deren Bewältigung für viele Anwendungsfälle in der Kommunikationswissenschaft auf den ersten Blick keinen größeren Nutzen verspricht.

Velleicht ist im Studium aber auch schon eine Situation aufgetreten, in der SPSS keine Hilfe bot. Eine Effektstärke für einen Mittelwertvergleich? Die bietet SPSS zwar in Form von η_p^2 für die ANOVA an, nicht jedoch Cohen's d für den t-Test. Sie haben zusammen mit Komilliton*innen eine Inhaltsanalyse geplant und möchten vorab einen Intercoderreliabilitätstest durchführen? SPSS kennt weder die Reliabilität nach Holsti noch Krippendorff's α . Und auch wenn SPSS Grafiken ausgeben kann, so hat es doch einen Grund, warum man diese selten in wissenschaftlichen Veröffentlichungen (und hoffentlich auch in studentischen Arbeiten) findet.

Benötigt man also eine Funktion, die in der gewählten Softwarelösung nicht vorhanden ist, so muss man auf eine andere ausweichen. Programmiersprachen bieten hier deutlich mehr Flexibilität - ist die gewünschte Funktion nicht vorhanden, so schreibt man sie eben selbst (bzw. hat dies in aller Regel schon jemand anderes, der ebenfalls vor diesem Problem stand, für Sie getan). Dies gilt natürlich umso mehr, je weniger standardisiert die zu analysierenden Daten und gewählten Analyseverfahren sind. Beschäftigen wir uns beispielsweise mit Onlinetexten oder digitalen Spurendaten, dann liegen diese oftmals nicht in vorstrukturierter Form vor, müssen erst über Schnittstellen abgerufen, automatisiert heruntergeladen und/oder für die weitere Nutzung aufbereitet werden. Computationale Analyseverfahren wie beispielsweise Verfahren zur automatisierten Inhaltsanalyse werden beständig weiterentwickelt und angepasst. Die Flexibilität, die skriptbasierte Datenanalyse bietet, ist daher einer der Hauptgründe, warum nicht nur in der Wissenschaft, sondern auch in anderen professionellen Kontexten, etwa der Markt- und Medienforschung, wo Lösungen für vielseitige datenanalytische Fragen gesucht werden, die Bedeutung von Programmiersprachen zur Datenanalyse zunimmt.

Zugleich ist der Einstieg in das Programmieren deutlich einfacher geworden. Für viele Programmiersprachen stehen sogenannte Integrierte Entwicklungsumgebungen (IDEs) zur Verfügung, die mittels grafischer Benutzeroberflächen, intuitiver Bedienung und Hilfswerkzeugen (z. B. der automatischen Vervollständigung von Funktionsnamen) den Umgang mit Programmiersprachen deutlich erleichtern und komfortabler gestalten.

Ein weiterer entscheidender Vorteil der *programmatischen*¹ Datenanalyse ist, dass Skripte und Code alle Analyseschritte nachvollziehbar, transparent und reproduzierbar gestalten (entsprechend wurden Sie in der Datenanalyse-Ausbildung vermutlich auch dazu angehalten, in SPSS stets die Syntax zu nutzen). Einmal durchgeführte Arbeiten können somit jederzeit und problemlos von anderen und auch Ihnen selbst wiederholt und angepasst werden.

Schließlich können auch karrieretechnische Überlegungen eine Rolle spielen. Viele Unternehmen setzen für datenanalytische Tätigkeiten die Kenntnis einer

¹d. h. skript- bzw. codebasiert; im Englischen wird *programmatically* verwendet, um auszudrücken, dass etwas 'durch Code' und nicht durch Klicken von Knöpfen in einem Computerprogramm erfolgt ist, im Deutschen ist diese Wortbedeutung außerhalb von Informatikkreisen (noch) kaum geläufig; siehe auch diese Diskussion zur Wortbedeutung.

einschlägigen Programmiersprache inzwischen zwingend voraus. Und natürlich spiegelt sich das auch im Gehalt wider: das Vergleichsportal *PayScale* gibt beispielweise für *Data Analysts*, die die statistische Programmiersprache R beherrschen, ein um rund 5.000 US-Dollar höheres Jahresdurchschnittsgehalt an als für diejenigen Data Analysts, die mit SPSS arbeiten.

Warum **R** lernen?

Bisher wurde allgemein von Programmiersprachen gesprochen. In der Datenanalyse-Praxis sind viele unterschiedliche Programmiersprachen gängig, z. B. *Python*, *R*, *SQL* und *Julia*. Wir werden in den kommenden zwei Semestern mit R arbeiten. Dies hat einige Gründe:

- R ist eine speziell auf statistische und datenanalytische Anwendungen ausgelegte Programmiersprache (auch wenn die Anwendungsbereiche inzwischen darüber hinausgehen). Das bedeutet, dass viele gängige statistische Verfahren bereits in der Basis-Version vorhanden sind und ohne weitere Anpassungen genutzt werden können.
- In der *Scientific Community* ist R inzwischen sehr weit verbreitet und wird durch diese kontinuierlich weiterentwickelt. Das bedeutet auch, dass neue Verfahren, sowohl zur Datenerhebung als auch zur Datenanalyse, meist sehr schnell auch in R verfügbar sind.
- Zugleich gibt es durch die weite Verbreitung auch vielzählige Hilfsangebote. In Communities wie Stack Overflow und durch googeln werden Sie für nahezu jedes Problem, das sich Ihnen bei der Arbeit mit R stellt, schnell eine Lösung finden.
- R ist komplett kostenlos und für jedes Betriebssystem verfügbar.
- Mit *RStudio* steht eine ebenfalls kostenfreie IDE zur Verfügung, die die ehemals hohen Einstiegshürden erheblich senkt.
- R und RStudio decken durch Erweiterungen nahezu alle Schritte ab, die für die wissenschaftliche Arbeit erforderlich sind. Das reicht vom Datenabruf aus Befragungssoftware sowie der Datenerhebung durch Programmierschnittstellen oder Web Scraping über die Datenbearbeitung, -bereinigung und -analyse bis hin zur Erstellung von Manuskripten und publikationsfähigen Grafiken. Auch dieser Kurs ist komplett in RStudio erstellt.

Auch wenn sich R in einigen Aspekten von den oben genannten Programmiersprachen unterscheidet, so sind viele der Konzepte, die wir in den kommenden zwei Semestern lernen werden, auch in anderen Programmiersprachen gleich oder zumindest ähnlich umgesetzt. Ihnen wird es in Zukunft also auch leichter fallen, sich bei Bedarf in andere Programmiersprachen einzuarbeiten.

Hinweise zur Nutzung des Online-Kurses

- In der Onlineversion können Sie mit den Cursortasten \leftarrow und \rightarrow durch die Seiten des Kurses blättern.

- In der oberen Leiste finden Sie einen Download-Knopf, mit dem Sie sich die aktuelle Version des Kurses als *PDF* oder *EPUB* (für E-Reader) herunterladen können. Bitte achten Sie in diesem Fall darauf, regelmäßig die aktuellste und somit vollständigste Version herunterzuladen. Oben auf dieser Seite ist angegeben, wann der Kurs zuletzt aktualisiert wurde.
- Früher oder später wird etwas in Ihrem Code nicht so funktionieren, wie Sie sich das vorstellen oder wünschen. Hier greift die 15-Minuten-Regel: Versuchen Sie zunächst, 15 Minuten lang das Problem selbst zu lösen - in dem Sie das Problem in kleinere Schritte zerlegen, den Code nach Tippfehlern durchsuchen, nochmals Hilfsdokumente konsultieren etc. Sind Sie nach 15 Minuten noch nicht weitergekommen, fragen Sie um Hilfe - z. B. in unseren Moodle-Foren.
- Der Witz, wonach Programmieren zu 70% aus Googeln bestehe, hat einen wahren Kern. Es ist nicht verwerlich, im Internet nach Hilfestellungen und Lösungen zu suchen und Code-Schnipsel von anderen zu verwenden - ganz im Gegenteil, gezieltes Suchen stellt einen wesentlichen Teil der Problemlösekompetenz dar. Auch wenn es jedoch verlockend und einfach erscheinen mag, Code von StackOverflow und vergleichbaren Portalen zu kopieren, sollten Sie immer versuchen, den Code und damit die Lösung auch nachvollziehen zu können.

Beginnen wir mit der Installation von R und RStudio sowie ersten Schritten.



Figure 1: Illustration von @allison_horst: https://twitter.com/allison_horst

Part I

Eine kurze Einführung in R

Chapter 1

Installation und erste Schritte

In diesem Kapitel installieren wir die notwendige Software und machen uns mit der Benutzeroberfläche von RStudio vertraut.

1.1 R installieren

Zunächst benötigen wir natürlich R. Die aktuellste Version erhalten wir immer über CRAN (*Comprehensive R Archive Network*).

Unter “Download and Install R” wählen wir zunächst unser Betriebssystem. Im Falle von Windows wählen wir zusätzlich auf der folgenden Seite noch “base” (die Basisversion) aus. Es sollte dann ein Download-Link für die aktuellste Version erscheinen (3.6.3, Stand 9. April 2020). Der Installationsprozess selbst läuft wie bei anderer Software auch ab.

Neben einem *Interpreter*, einem Programm, das Code (in diesem Fall also Code, der in R geschrieben wurde) für unseren Computer in ausführbare Befehle übersetzt, umfasst die Installation von R auch schon eine (sehr) rudimentäre grafische Benutzeroberfläche, die aber nur wenig komfortabel und nutzerfreundlich ist. Als nächstes installieren wir daher noch RStudio.

1.2 RStudio installieren

RStudio ist eine grafische Benutzeroberfläche für R, die die Arbeit mit der Programmiersprache deutlich erleichtert. Auch RStudio ist für Privatanwender komplett kostenfrei nutzbar. Die aktuellste Version kann über <https://rstudio.com/products/rstudio/download/#download> heruntergeladen werden.

Für den Rest des Kurses arbeiten wir immer mit RStudio (und nicht direkt mit der Oberfläche von R). Öffnen wir also zum ersten Mal RStudio.

1.3 Die Benutzeroberfläche von RStudio

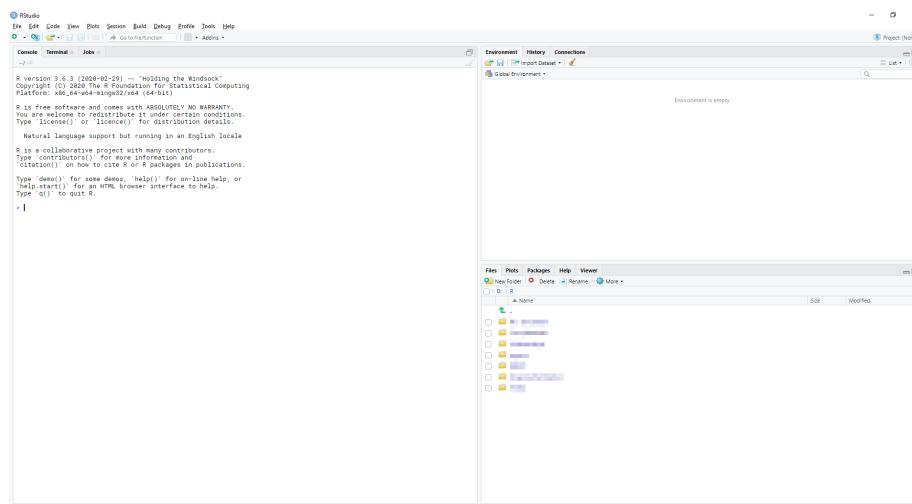


Figure 1.1: Die Benutzeroberflche von RStudio

Nach dem Starten von RStudio sollte sich das Programm Ihnen wie oben präsentieren - mit einer Dreiteilung in drei abgetrennte Bereiche. Wir beginnen mit dem großen, aktuell noch weitestgehend leeren Bereich auf der linken Seite, der Konsole.

1.3.1 Konsole

Die Konsole ist zugleich das Eingabe- und das Ausgabefenster von R bzw. RStudio. Befehle, die wir hier eingeben, werden durch Druck auf die **Eingabe/Enter**-Taste direkt ausgeführt. Die Konsole signalisiert uns, dass sie bereit ist, einen Befehl zu empfangen, durch ein vorangestelltes **>**. Wir können dies mit simplen Berechnungen ausprobieren:

(Zur Darstellung in diesem Kurs: die erste hellgraue Box umfasst hier und im Folgenden jeweils die Befehle, die wir eingeben - in diesem Fall also den Befehl 1 + 2. Die zweite hellgraue Box enthält dann immer die Ausgabe in der Konsole, gekennzeichnet durch zwei vorangestellte Rautensymbole ##.)

1 + 2

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

Die Konsole spuckt also direkt das Ergebnis aus – in diesem Fall 3 – und wartet

auf den nächsten Befehl, wieder zu erkennen am >. Die [1] links neben dem Ergebnis gibt an, dass es sich hierbei um den ersten (und einzigen) Ausgabewert handelt. Wir werden aber noch zahlreiche Befehle kennenlernen, bei denen mehr als nur ein Wert ausgegeben wird.

Mit den Cursortasten \uparrow und \downarrow können wir in der Konsole durch bisher eingegebene Befehle schalten. Ein Druck auf \uparrow sollte also den ersten und bisher einzigen Befehl - 1 + 2 - anzeigen.

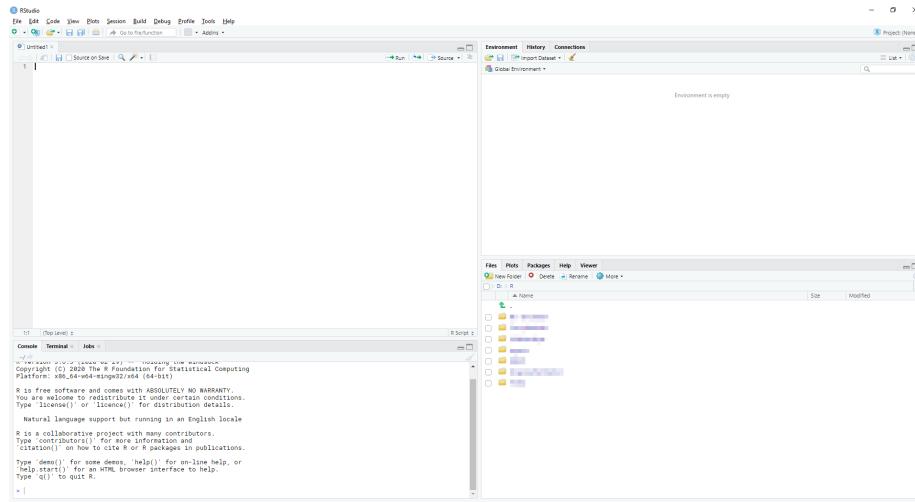
Ist ein Befehl noch nicht vollständig, signalisiert uns dies die Konsole durch ein vorangestelltes +. Wir können dies ausprobieren, in dem wir beispielsweise eine unvollständigen Additionsbefehl eingeben: 3 +. Die Konsole wartet nun auf den restlichen Befehl - in diesem Fall können wir eine weitere Zahl eingeben und den Befehl abschließen. Alternativ können wir den unvollständigen Befehl durch Druck der ESC-Taste abbrechen.

In der Praxis passiert dies vor allem, wenn in einem längeren Befehl eine Klammer) oder Anführungszeichen " fehlt. Sollte die Konsole also einmal die Arbeit verweigern, liegt das oft daran, dass noch ein unvollständiger Befehl vorhanden ist.

Prinzipiell könnten wir alle Arbeitsschritte über die Konsole ausführen. Das ist in der Praxis aber wenig sinnvoll, da wir im Normalfall längere und mehrere Befehle hintereinander ausführen und diese auch festhalten möchten. Wir arbeiten daher mit Skript-Dateien.

1.3.2 R-Skripte

Um eine neue Skriptdatei zu erstellen, klicken wir entweder links oben auf das Symbol mit der leeren Seite und dem grünen Plus und anschließend auf R-Script, auf *File - New File - R-Script* oder drücken die Tastenkombination **Strg/Cmd + Shift + N**. Es sollte sich im links-oberen Bildschirmviertel eine leere Skriptdatei öffnen und unser RStudio-Fenster somit in ein viergeteiltes Layout übergehen:



Hier können wir nun alle Befehle der Reihen der Reihe nach schreiben und gesammelt abspeichern. Einzelne Befehlszeilen lassen sich über die Tastenkombination **Strg/Cmd + Eingabe/Enter** ausführen. Das Ergebnis des Befehls erscheint dann in der Konsole. Wir können auch mehrere Zeilen auf einmal markieren und gemeinsam über dieselbe Tastenkombination ausführen. Schreiben wir z. B. mehrere Rechenoperationen hintereinander, so erscheinen deren Ergebnisse in der Ausführungsreihenfolge in der Konsole:

```
1 + 3
12 * 25
17 / 4
```

```
## [1] 4
## [1] 300
## [1] 4.25
```

Längere Skriptdateien werden schnell unübersichtlich. Wir können aber an jeder Stelle Kommentare einfügen, indem wir eine Raute **#** voran stellen - alles was in dieser Zeile *hinter* dem Symbol steht, wird von R beim Ausführen ignoriert, wir können also sowohl ganze Zeilen *auskommentieren* als auch hinter R-Befehlen eine kurze Erklärung hinzufügen. Außerdem können jederzeit Leerzeilen eingefügt werden, um das Skript etwas aufzulockern:

```
# Zunächst ein wenig Addition
2 + 5
6 + 12

# Dann ein wenig Multiplikation
21 * 35
2345 * 1.6

# Und zum Schluss etwas komplexere Rechenoperationen
```

```
(2 + 3) ^ 3 # Das ^ steht für Exponentiation, hier also 5 hoch 3
```

```
## [1] 7
## [1] 18
## [1] 735
## [1] 3752
## [1] 125
```

Skriptdateien können und sollten natürlich abgespeichert werden - entweder über *File - Save* oder die Tastenkombination **Strg/Cmd + S**. R-Skriptdateien erhalten die Dateiendung **.R**.

1.3.3 Environment

Im rechten oberen Bildschirmbereich öffnet sich standardmäßig das *Environment* an, in dem RStudio alle derzeit angelegten und somit verfügbaren Objekte anzeigt. Mit Objekten werden in uns im nächsten Kapitel genauer auseinandersetzen.

Weitere Registerkarten in diesem Bereich sind die *History* (eine Auflistung sämtlicher ausgeführter Zeilen der aktuellen Sitzung) sowie *Connections* und *Build*, die für uns aber vorerst keine Rolle spielen werden.

1.3.4 Files

Der rechte untere Bildschirmbereich zeigt standardmäßig einen Dateibrowser (*Files*) an, der das aktuelle Arbeitsverzeichnis zeigt. Auch damit setzen wir uns in den kommenden Kapiteln ausführlicher auseinander.

Weitere Registerkarten in diesem Bereich sind:

- *Plots*: hier werden Grafiken angezeigt, wenn wir diese in R erstellen.
- *Packages*: Eine Übersicht aller installierten Packages (kurz gesagt Sammlungen von R-Funktionen, die nicht in der Basisversion enthalten sind). Auch mit Packages beschäftigen wir uns in einem eigenen Kapitel.
- *Help*: Hier wird die Dokumentation einzelner Funktionen angezeigt, sobald wir diese anfordern. Diesen Bereich sehen wir uns an, sobald wir uns mit Funktionen beschäftigen.
- *Viewer*: Hier kann RStudio Webinhalte anzeigen, die mit R-Funktionen erstellt wurden. Dies wird vorab keine Rolle für uns spielen.

1.4 RStudio anpassen

Unter *Tools - Global Options* können wir RStudio nach unseren Wünschen anpassen. Die einzelnen Einstellungsmöglichkeiten sollen hier nicht ausführlich diskutiert werden; hier jedoch einige sinnvolle Einstellungen:

Im Bereich *General* ist es sinnvoll, zwei Anpassungen vorzunehmen. Zum einen können wir unter *R Sessions* ein *Default working directory* (also ein

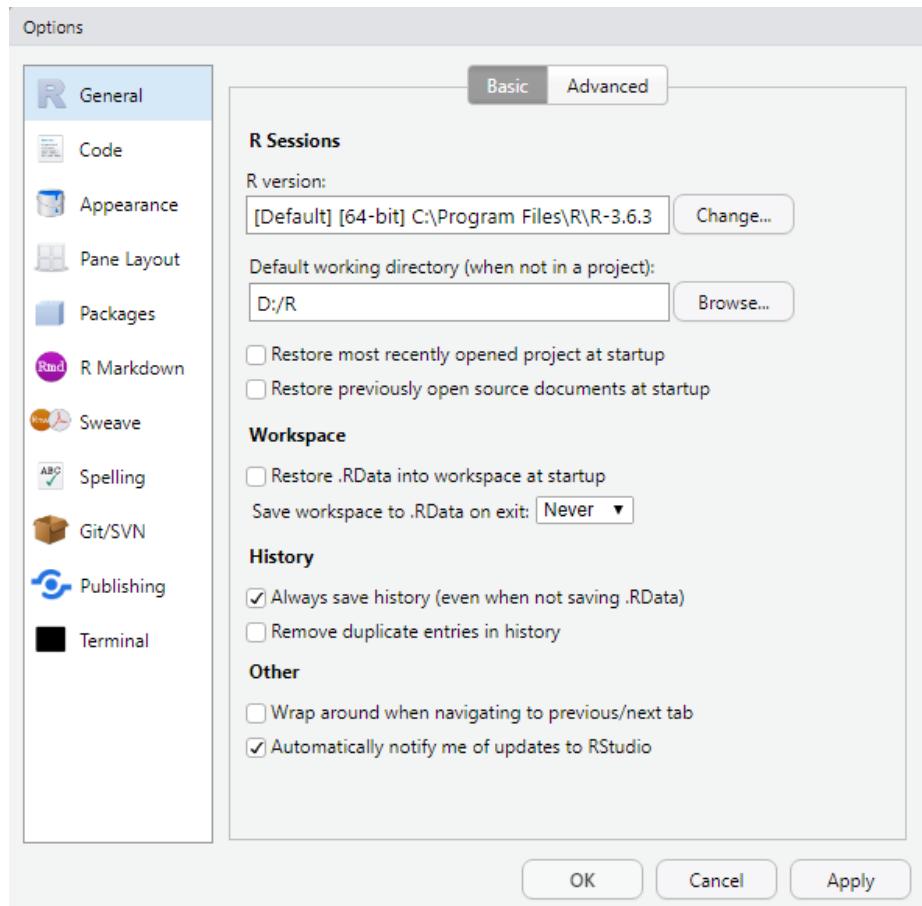


Figure 1.2: Global Options in RStudio

standardmäßiges Arbeitsverzeichnis) einstellen. Dieses Verzeichnis öffnet R dann beim Start automatisch. Hier bietet es sich an, einen eigenen Ordner anzulegen.

Unter *Workspace* entfernen Sie bitte, falls vorhanden, das Häkchen bei *Restore .RData into workspace at startup* und stellen *Save workspace to .RData on exit* auf *Never*. Zwar mag es praktisch erscheinen, dass RStudio automatisch die zuletzt bearbeitete *Session* wiederherstellt, das führt in der Praxis aber gerne zu Konflikten und Problemen – und letztlich ist es längerfristig auch sinnvoller, sich einen Arbeitsprozess anzueignen, bei dem Skripte schnell den jeweiligen Arbeitsstand wiederherstellen.

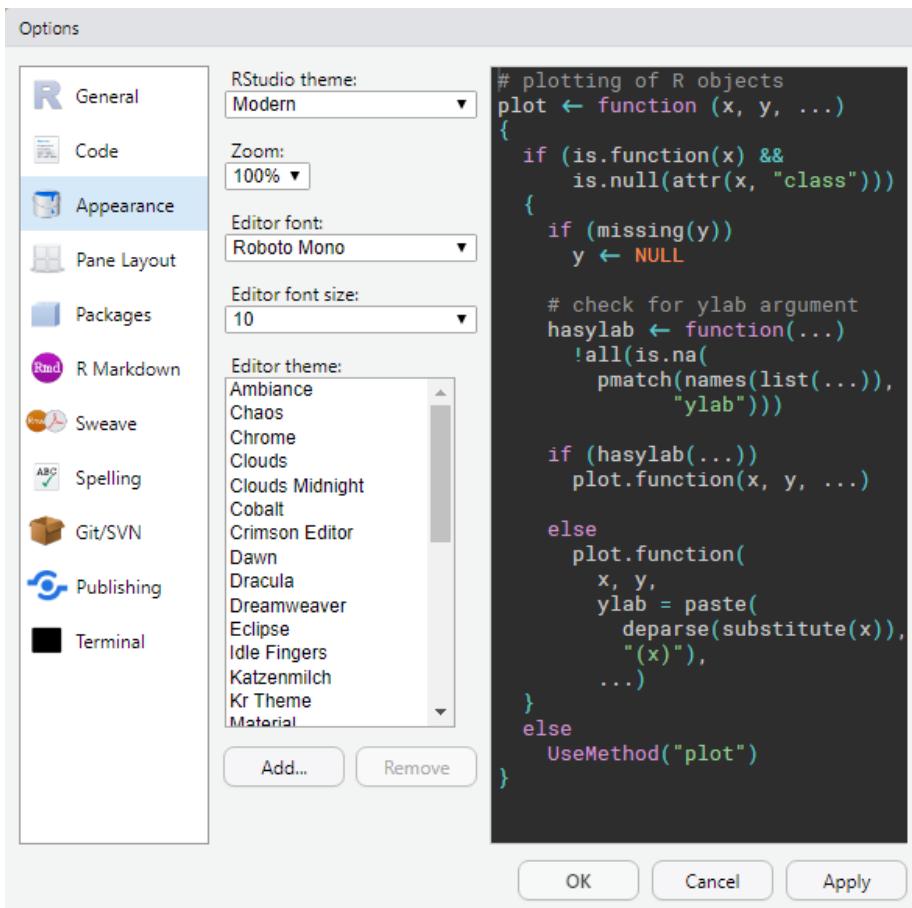


Figure 1.3: Anzeigeeinstellungen in RStudio

Eher Geschmackssache sind die Anzeigeeinstellungen, die Sie unter *Appearance* vornehmen können. Hier können Sie die Schriftart und -größe im Skripteditor einstellen sowie unter verschiedenen *Themes* (Farbschemata) wählen (darunter auch “dunkle” Themes, also solche, die hellen Text auf dunklem Hintergrund

bieten). Am besten, Sie probieren hier unterschiedliche Einstellungen aus, bis Sie ein subjektiv angenehmes Anzeigebild von RStudio gefunden haben.

1.5 Übungsaufgaben

Diese Aufgaben sollen Sie lediglich mit den grundlegendsten Funktionen von RStudio vertraut machen. Sie müssen daher keine Dateien abgeben.

Übungsaufgabe 1.1. Installieren Sie R (siehe 1.1) und RStudio (1.2) und nehmen Sie die Einstellungen unter 1.4 vor.

Übungsaufgabe 1.2. Öffnen Sie RStudio und führen Sie ein paar simple Berechnungen in der Konsole durch.

Übungsaufgabe 1.3. Erstellen Sie eine neue Skriptdatei und fügen dort mindestens sechs Berechnungen hinzu. Gliedern Sie die Skriptdatei durch einige Kommentare.

Chapter 2

Objekte und Datenstrukturen

Wir können R bzw. RStudio nun als Taschenrechner verwenden und uns arithmetische Operationen direkt in der Konsole ausgeben lassen:

```
1 + 2 # Addition  
1 - 2 # Subtraktion  
2 * 2 # Multiplikation  
4 / 2 # Division  
2 ^ 5 # Exponentiation
```

```
## [1] 3  
## [1] -1  
## [1] 4  
## [1] 2  
## [1] 32
```

Wirklich sinnvoll ist dies aber nicht: wir wollen Ergebnisse ja auch speichern und weiterverwenden können. Hierfür benötigen wir Variablen, also Namen, denen wir (veränderliche) Werte zuordnen können.

In R lassen sich Variablen erstellen, indem wir einer Zeichenkette (zu den Benennungsregeln kommen wir gleich) einen Wert mittels `<-` zuordnen (hierfür entweder die Zeichen `<` und `-` eingeben oder die Tastenkombination Alt/Option `+ -` drücken).¹ Dies erstellt ein *Objekt*² mit eben diesem Namen und der

¹Es ist prinzipiell auch möglich, die Zuordnung mittels einem `=` vorzunehmen. Da das `=` aber auch in anderen Kontexten benötigt wird, erzeugt dies Verwirrung, sodass wir Objekte immer mit `<-` zuordnen sollten.

²Die Begriffe Variable und Objekt können für unsere Zwecke weitestgehend synonym verwendet werden. Wir werden aber gleich noch sehen, dass so ziemlich alles in R ein Objekt sein kann.

entsprechenden Zuordnung:

```
x <- 2
```

Führen wir diesen Befehl aus, erstellen wir das Objekt `x` und ordnen den Wert `2` zu. Wir sollten diese neue Zuordnung zudem im rechten oberen *Environment*-Bereich sehen können.

Wir können nun mit diesem Objekt weiterrechnen:

```
x * 2
x + 5
x / 2
```

```
## [1] 4
## [1] 7
## [1] 1
```

Um den aktuellen Wert eines Objektes anzuzeigen, können wir auch einfach das Objekt ausführen:

```
x
```

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

Generell wird bei der Zuordnung immer zunächst der Teil rechts vom `<-` ausgeführt und dann zugeordnet. Wir können also auch komplexere Befehle ausführen und diese Zuordnen:

```
y <- (2 + 4) * (3 - 1) / 2
y
```

```
## [1] 6
```

Objekte sind veränderlich und können jederzeit neu zugeordnet werden - und dabei auch selbst bei der Zuordnung verwendet werden:

```
x <- 2
x
x <- 3
x
x <- x - 1
x
```

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

Schauen wir uns das einmal in einem etwas komplexeren Beispiel an - welchen Wert hat `b` am Ende dieser Befehlskette?

```
a <- 10
b <- a / 2
```

```
a <- b * 2 + a
b <- a - b
```

Die Antwort lautet 15. Gehen wir das der Reihe nach durch:

1. Zunächst ordnen wir **a** den Wert 10 zu.
2. Dann ordnen wir **b** den Wert $a / 2$ zu. Da **a** in diesem Schritt 10 zugeordnet ist, wird $10 / 2$ gerechnet. **b** entspricht nun also dem Wert 5.
3. Wir ordnen nun **a** den Wert $b * 2 + a$ zu. Der gesamte Teil rechts vom `<-` wird zuerst ausgeführt und dann zugeordnet, hier also $5 * 2 + 10$. **a** entspricht nun dem Wert 20, **b** weiterhin dem Wert 5.
4. Zuletzt ordnen wir **b** das Ergebnis von $a - b$ zu, was vor dieser Zuordnung $20 - 5$ bedeutet. **b** entspricht schlussendlich also 15.

Nochmals die wichtigsten Punkte zusammengefasst:

- Mit `<-` erstellen wir Objekte und ordnen diesen Werte zu.
- Alle Objekte sind veränderlich und können überschrieben werden.
- Bei einer Zuordnung wird der gesamte Teil rechts vom Zuordnungsfeil `<-` zuerst ausgeführt und dann die Zuordnung vorgenommen.

2.1 Objektnamen

Für die obigen Beispiele haben wir nur einzelne Buchstaben für Objekte verwendet. In der Praxis können und sollten wir längere Objektnamen verwenden. Dabei gelten folgende Regeln:

- Objektnamen können Groß- und Kleinbuchstaben, Ziffern sowie Punkte `.` und Unterstriche `_` beinhalten. Andere Sonderzeichen, Umlaute und Leerzeichen sind nicht gestattet.
- Objektnamen können mit einem Buchstaben oder einem `.` beginnen, nicht jedoch mit Ziffern oder `_`.
- Objektnamen sind *case-sensitive*, d. h. unterscheiden zwischen Groß- und Kleinschreibung. `myVar` und `myvar` sind also unterschiedliche Objekte.

Es ist sinnvoll, Objekten “sprechende” Namen zu geben, sodass andere (und auch Sie zu einem späteren Zeitpunkt) nachvollziehen können, was sich dahinter verbirgt, auch ohne den gesamten Code zu lesen.

```
# Gute Objektnamen
mittelwert <- 2.5
mein_alter <- 32
groesse_in_cm <- 175

# Schlechte Objektnamen
x1 <- 2.5
var2 <- 32
asdadasd <- 175
```

Es gibt außerdem unterschiedliche Konventionen, um mehrere Wörter in Objektnamen aneinanderzuhängen. `mein_alter` ist ein Beispiel für den sogenannten *snake_case*: Alle Wörter kleingeschrieben und durch einen Unterstrich `_` miteinander verbunden. Einen Überblick über verbreitete Konventionen der Objektbenennung gibt folgende Illustration:

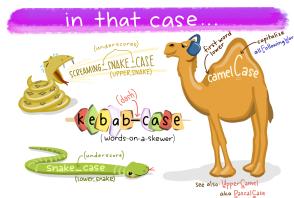


Figure 2.1: Illustration von @allison_horst: https://twitter.com/allison_horst

Was auch immer Sie wählen - wichtig ist vor allem, dass Sie einheitlich vorgehen.³

2.2 Objekttypen

Bisher haben wir lediglich Zahlen Objekten zugewiesen. Natürlich können Daten aber auch in anderen Formen vorliegen; wir sprechen daher von verschiedenen *Objekttypen*.⁴

2.2.1 Numerische Objekte

Zahlenwerte werden in R als `numeric` bezeichnet. Wir können hier zudem zwischen den Typen `integer` (ganze Zahlen) und `double` (Kommazahlen⁵) unterscheiden.

Grundsätzlich ordnet R Zahlen als `double` zu, auch wenn nur ganze Zahlen zugeordnet werden.

```
x <- 4
typeof(x) # Mit dieser Funktion können wir den Objekttyp anfordern

## [1] "double"
```

Um explizit den Typ `integer` anzufordern, muss Zahlenwerten ein L nachgestellt

³Aufmerksamen Leser*innen dürfte zudem aufgefallen sein, dass *kebab-case* in R nicht möglich ist.

⁴Tatsächlich ist die Sache etwas komplexer: es gibt in R einige wenige Kernobjekttypen, die wiederum mit bestimmten Attributen versehen werden können, um daraus zusätzliche Objekttypen abzuleiten. So kann etwa eine Zahlenfolge mit einem zusätzlichen Attribut als Datumsangabe interpretiert werden. Für unsere Zwecke spielt diese Unterscheidung jedoch keine Rolle.

⁵Der Typenbezeichnung leitet sich von Gleitkommazahlen mit doppelter Genauigkeit ab.

werden:⁶

```
x <- 4L
typeof(x)

## [1] "integer"
```

In der Praxis macht es aber kaum einen Unterschied, ob eine ganze Zahl als `integer` oder `double` abgespeichert wird – `integer` verbraucht weniger Speicherplatz, aber das wird erst bei *sehr* großen Datensätzen relevant. Wir können die Unterscheidung also guten Gewissens ignorieren und von numerischen Objekten sprechen.

2.2.2 Textobjekte

Wir können Objekten auch Text zuordnen - diese Objekte haben dann den Typ `character` (Textvariablen werden zudem häufig als “string” bezeichnet). Um ein `character`-Objekt zu erstellen, müssen wir die Zeichenkette in einfache '' oder doppelte "" Anführungszeichen setzen:

```
text1 <- "Guten Morgen!"
text2 <- 'Einfache Anführungszeichen sind sinnvoll, wenn im "Text" ebenfalls Anführungszeichen v'
```

Natürlich können auch Zahlen als Text gespeichert werden - werden dann aber natürlich auch als Text behandelt, sodass man nicht mehr mit ihnen rechnen kann.

```
zahl_als_text <- "123"
```

2.2.3 Logicals (logische Objekte)

Der dritte Kernobjekttyp heißt `logical` und kann nur zwei Werte annehmen: `TRUE` (wahr) oder `FALSE` (falsch). Logicals entstehen durch logische Vergleiche zweier Objekte, wobei u.a. folgende Operatoren verwendet werden können:

Table 2.1: Logische Operatoren in R

Operator	Vergleich	Beispiele
<code>==</code>	ist gleich	<code>1 == 1</code> (ergibt <code>TRUE</code>) <code>"a" == "b"</code> (ergibt <code>FALSE</code>)
<code>!=</code>	ist nicht gleich	<code>1 != 1</code> (ergibt <code>FALSE</code>) <code>"a" != "b"</code> (ergibt <code>TRUE</code>)
<code><</code>	ist kleiner als	<code>1 < 2</code> (ergibt <code>TRUE</code>) <code>2 < 2</code> (ergibt <code>FALSE</code>)
<code>></code>	ist größer als	<code>2 > 1</code> (ergibt <code>TRUE</code>) <code>2 > 2</code> (ergibt <code>FALSE</code>)
<code><=</code>	ist kleiner gleich	<code>1 <= 2</code> (ergibt <code>TRUE</code>) <code>2 <= 2</code> (ergibt <code>TRUE</code>)
<code>>=</code>	ist größer gleich	<code>2 >= 1</code> (ergibt <code>TRUE</code>) <code>2 >= 2</code> (ergibt <code>TRUE</code>)

⁶Warum L? Hier gibt es unterschiedliche Erklärungsansätze, die beispielsweise hier nachgelesen werden können

Die Zuordnung erfolgt wie bei anderen Objekten auch:

```
x <- "a" == "b"
x
## [1] FALSE
```

Logicals werden vor allem bei Wenn-Dann-Bedingungen benötigt, mit denen wir uns im übernächsten Kapitel auseinandersetzen werden.

2.2.4 Weitere Objekttypen

Diese drei Objekttypen (`numeric`, `character`, `logical`) bilden die Basis fast aller Objekte in R. Durch zusätzliche Attribute können jedoch noch zusätzliche Objekttypen erzeugen, die den Umgang mit bestimmten Daten erleichtern. Für kategoriale Variablen kennt R beispielsweise den Typ `factor`, für Datumsangaben den Typ `date`. Diese werden bei der Zuordnung nicht automatisch erkannt und müssen stattdessen durch bestimmte Funktionen erzeugt werden.

Erzeugen wir beispielsweise ein Objekt mit einer Zeichenfolge, die ein Datum repräsentiert (z. B. `date1 <- "2020-05-05"`, im Format `YYYY_MM_DD`, also Jahr-Monat-Tag), speichert R dies zunächst als `character` ab. Wir können aber R explizit sagen, dass er dies als Datum behandeln soll:

```
date2 <- as.Date("2020-05-05")
```

Dies hat nun u.a. den Vorteil, dass wir im Gegensatz zu `character`-Objekten auch arithmetische Operationen durchführen können, also beispielsweise zwei Datums-Objekte voneinander subtrahieren, um die zeitliche Differenz zu berechnen.

Wir werden uns im späteren Verlauf noch ausführlicher mit diesen spezielleren Objekttypen beschäftigen – bis jetzt nehmen Sie vor allem mit, dass sowohl kategoriale Variablen als auch Datumsangaben kein Problem für R darstellen.

2.2.5 Fehlende Werte

Fehlende Werte werden in R als `NA` angegeben. Es ist sinnvoll, fehlende Werte immer explizit als `NA` zu kennzeichnen und nicht etwa durch einen negativen Wert bei numerischen Variablen (z. B. `-9`) oder durch einen leeren String bei Textvariablen (""), damit sichergestellt ist, dass Funktionen den fehlenden Wert auch entsprechend als einen solchen behandeln.

2.2.6 Objekttypen ändern

Bisweilen wird es relevant sein, Objekttypen zu ändern - etwa weil Zahlen fälschlicherweise als Text eingelesen wurden. Hierfür bietet R Funktionen an, die allesamt nach dem Schema `as.[Objekttyp]()` aufgebaut sind: mit `as.numeric()` wandeln wir Objekte in numerische Objekte um (genauer gesagt

in `double`), mit `as.character()` in Textobjekte und, wie im vorigen Abschnitt gesehen, mit `as.Date()` in ein Datumsobjekt. Der Fachbegriff hierfür lautet *Coercion*, wir zwingen R also dazu, ein Objekt als einen bestimmten Typ zu behandeln, auch wenn R automatisch einen anderen Typus bestimmt hätte.

```
x1 <- "25"
x1
typeof(x1)
x2 <- as.numeric("25")
x2 # Beachten Sie, dass in der Ausgabe nun die Anführungszeichen fehlen
typeof(x2)

## [1] "25"
## [1] "character"
## [1] 25
## [1] "double"
```

Natürlich klappt das nur, solange die Umwandlung auch sinnvoll durchführbar ist – in allen anderen Fällen wird eine Warnung ausgegeben und es werden fehlende Werte erzeugt.

```
x <- as.numeric("Dieser Text kann nicht sinnvoll als Zahl interpretiert werden")

## Warning: NAs introduced by coercion
x

## [1] NA
```

2.3 Datenstrukturen

Bisher haben wir einem Objekt immer nur einen einzigen Wert zugeordnet. Der Fachbegriff hierfür lautet *Skalar* und beschreibt somit die einfachst mögliche Datenstruktur, eben dass einem Objekt nur ein einziger Wert zugeordnet wurde. Objekte können in R jedoch auch mehrere Werte enthalten und somit komplexere Datenstrukturen erzeugen.

Im Folgen betrachten wir daher die vier wichtigsten komplexeren Datenstrukturen in R. Diese unterscheiden sich zum einen in ihrer Dimensionalität (also ob sie eindimensional sind) und zum anderen, ob sie homogene (also nur dieselben) oder heterogene (also unterschiedliche) Objekttypen beinhalten können:

Table 2.2: Datenstrukturen in R

Datenstruktur	Dimensionalität	Objekttypen
Vektor	eindimensional	homogen
Liste	eindimensional	heterogen
Matrix	zweidimensional	homogen

Datenstruktur	Dimensionalität	Objekttypen
Dataframe	zweidimensional	heterogen

2.3.1 Vektoren

Vektoren sind Objekte, die mehrere Werte desselben Typs beinhalten. Wir erzeugen Vektoren über die Funktion `c()` (von *concatenate*, also verketten). Die einzelnen Elemente des Vektors werden durch Kommas , getrennt.

```
gerade_zahlen <- c(2, 4, 6, 8)
gerade_zahlen
ungerade_zahlen <- c(1, 3, 5, 7, 9)
ungerade_zahlen
simpsons <- c("Homer Simpson", "Marge Simpson", "Bart Simpson", "Lisa Simpson", "Maggie Simpson")
```

```
## [1] 2 4 6 8
## [1] 1 3 5 7 9
## [1] "Homer Simpson"  "Marge Simpson"  "Bart Simpson"   "Lisa Simpson"   "Maggie Simpson"
```

Wir können auch Vektoren über `c()` mit einander verketten:

```
zahlen <- c(gerade_zahlen, ungerade_zahlen)
zahlen
```

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

Beachten Sie, dass die Verkettung immer in der angegebenen Reihenfolge erfolgt – R sortiert die Elemente also nicht automatisch.

2.3.1.1 Vektorelemente auswählen

Um bestimmte Elemente eines Vektors auszuwählen, können wir die gewünschten Elemente in eckigen Klammern [] hinter einem Vektor definieren. Hier geben wir nur das zweite Element des oben erzeugten `zahlen`-Vektors aus:

```
zahlen[2]
```

```
## [1] 4
```

Um mehrere Elemente eines Vektors auszugeben, benötigen wir wiederum einen Vektor mit den gewünschten Positionen – hier geben wir uns beispielsweise das erste, dritte und fünfte Element aus:

```
zahlen[c(1, 3, 5)]
```

```
## [1] 2 6 1
```

2.3.1.2 Vektorelemente benennen

Elemente in Vektoren können benannt werden, indem beim Erstellen die Namen der Elemente mit einem `=` angegeben werden:

```
homer <- c(nachname = "Simpson", vorname = "Homer", wohnort = "Springfield")
homer

##      nachname      vorname      wohnort
##      "Simpson"     "Homer"    "Springfield"
```

Benannte Elemente können dann auch über den Namen ausgewählt werden:

```
homer["wohnort"]

##      wohnort
## "Springfield"
```

Auch dies funktioniert mit mehreren Elementen gleichzeitig:

```
homer[c("vorname", "nachname")]

##      vorname      nachname
##      "Homer"     "Simpson"
```

Alternativ können Elementnamen im Nachhinein über die Funktion `names()` hinzugefügt werden:

```
marge <- c("Simpson", "Marge", "Springfield")
names(marge) <- c("nachname", "vorname", "wohnort")
marge

##      nachname      vorname      wohnort
##      "Simpson"     "Marge"    "Springfield"
```

2.3.1.3 Mit Vektoren rechnen

Mit numerischen Vektoren können arithmetische Berechnungen durchgeführt werden:

```
zahlen + 1
zahlen * 2

## [1]  3  5  7  9  2  4  6  8 10
## [1]  4  8 12 16  2  6 10 14 18
```

Berechnungen werden dabei der Reihe nach für jedes einzelne Vektorelement durchgeführt. Es ist auch möglich, Vektoren gleicher Länge zu addieren, subtrahieren etc. – im Falle einer Addition wird dann das erste Element des ersten Vektors zum ersten Element des zweiten Vektors addiert, dann das zweite Element des ersten Vektors zum zweiten Element des zweiten Vektors usw.:

```
x1 <- c(1, 3, 5)
x2 <- c(2, 3, 4)
x1 * x2

## [1] 2 9 20
```

2.3.1.4 Nützliche Vektorfunktionen

Abschließend einige nützliche Funktionen für den Umgang mit Vektoren:

- `length()` gibt die Anzahl der Elemente eines Vektors aus.
- Die unter 2.2.6 eingeführten *Coercion*-Funktionen (`as.numeric()`, `as.character()` usw.) können auch auf Vektoren angewendet werden und wandeln so jedes Vektorelement um.
- Für numerische Vektoren stehen zahlreiche statistische Funktionen bereit, z. B. zur Berechnung der Summe (`sum()`), des arithmetischen Mittels (`mean()`) und der Standardabweichung (`sd()`)

```
x <- c(10, 24, 32, 999)
length(x)
sum(x)
mean(x)
sd(x)

## [1] 4
## [1] 1065
## [1] 266.25
## [1] 488.5846
```

Oftmals benötigen wir aufsteigende Zahlenfolgen für Vektoren, beispielsweise für laufende Nummern. Dies lässt sich über die Funktion `:` abkürzen, die einen Vektor **Startwert:Endwert** erstellt:

```
eins_bis_zehn <- 1:10
eins_bis_zehn

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

2.3.2 Listen

Listen ähneln zunächst Vektoren und werden mit der Funktion `list()` erzeugt. Auch die Benennung von Listenelementen erfolgt analog zu Vektoren entweder beim Erstellen der Liste mit `=` oder im Nachhinein mit `names()`:

```
munich_facts <- list(name = "München", bundesland = "Bayern", bezirk = "Oberbayern")
munich_facts

## $name
## [1] "München"
```

```
##  
## $bundesland  
## [1] "Bayern"  
##  
## $bezirk  
## [1] "Oberbayern"
```

Wir sehen aber bereits am Konsolenoutput, dass die Darstellung von Vektoren abweicht: anstatt alle Elemente nebeneinander angezeigt zu bekommen, werden die einzelnen Elemente untereinander angezeigt.

Das röhrt daher, dass Listen deutlich mächtiger und flexibler sind als Vektoren. Nicht nur können wir in den einzelnen Elementen Objekttypen mischen, wir sind auch nicht auf einzelne Werte (Skalare) als Elemente beschränkt. Tatsächlich kann so gut wie jedes Objekt ein Listenelement sein – also auch Vektoren, ganze Datensätze und sogar Listen (die wiederum eigene Listen enthalten können – wir können hier also Daten prinzipiell endlos verschachteln). Erweitern wir dazu die Liste von oben:

```
munich_facts <- list(  
  namen = c(hochdeutsch = "München", englisch = "Munich", bairisch = "Minga"),  
  bundesland = "Bayern",  
  gruendungsjahr = 1158,  
  daten = list(  
    einwohner = 1471508,  
    geographie = c(flaeche_in_km2 = 310.7, hoehe_NHN_in_m = 519)  
  )  
)  
munich_facts  
  
## $namen  
## hochdeutsch      englisch      bairisch  
##   "München"      "Munich"      "Minga"  
##  
## $bundesland  
## [1] "Bayern"  
##  
## $gruendungsjahr  
## [1] 1158  
##  
## $daten  
## $daten$einwohner  
## [1] 1471508  
##  
## $daten$geographie  
## flaeche_in_km2  hoehe_NHN_in_m  
##           310.7          519.0
```

Mit ihrer Flexibilität stellen Listen in R die Basis für nahezu alle komplexeren Datenstrukturen – auch bei Datensätze, Regressionsmodellen etc. handelt es sich um Listen, die mit bestimmten Attributen versehen wurden.

2.3.2.1 Listenelemente auswählen

Auch die Auswahl von Listenelementen funktioniert ähnlich wie bei Vektoren über die numerische Position oder den Elementnamen:

```
munich_facts[1]

## $namen
## hochdeutsch    englisch    bairisch
##   "München"    "Munich"    "Minga"

munich_facts["daten"]

## $daten
## $daten$einwohner
## [1] 1471508
##
## $daten$geographie
## flaeche_in_km2 hoehe_NHN_in_m
##                 310.7          519.0
```

Vielelleicht ist Ihnen bereits das Dollarsymbol \$ vor den Elementnamen aufgefallen – dieses verweist auf eine Funktion, mit der Listenelemente noch komfortabler ausgewählt werden können:

```
munich_facts$bundesland
```

```
## [1] "Bayern"
```

Nicht nur sparen Sie sich ein paar Zeichen bei der Eingabe, RStudio macht Ihnen auch automatisch Vorschläge, sobald Sie das Dollarzeichen eingetippt haben (im Beispiel also ab `munich_facts$`), welche Elemente sie auswählen können. Auch tiefer verschachtelte Elemente können so ausgewählt werden:

```
munich_facts$daten$einwohner
```

```
## [1] 1471508
```

2.3.2.2 Nützliche Listenfunktionen

Auch für Listen stehen einige nützliche Funktionen zur Verfügung, die die Arbeit mit ihnen erleichtern.

`length` gibt wie auch schon bei Vektoren die Anzahl der Elemente aus (wobei auch komplexere Elemente, also z. B. Vektoren, Listen etc., jeweils als ein Element gezählt werden):

```
length(munich_facts)
```

```
## [1] 4
```

Spezifisch für Listen relevant ist die Funktion `str()`, die zusätzliche Informationen über die Struktur der Liste ausgibt, was gerade bei verschachtelten Listen den Überblick erleichtert:

```
str(munich_facts)
```

```
## List of 4
## $ namen      : Named chr [1:3] "München" "Munich" "Minga"
##   ..- attr(*, "names")= chr [1:3] "hochdeutsch" "englisch" "bairisch"
## $ bundesland  : chr "Bayern"
## $ gruendungsjahr: num 1158
## $ daten      :List of 2
##   ..$ einwohner : num 1471508
##   ..$ geographie: Named num [1:2] 311 519
##   ... ..- attr(*, "names")= chr [1:2] "flaeche_in_km2" "hoehe_NHN_in_m"
```

Wir sehen, dass die Liste `munich_facts` aus 4 Elementen besteht `List of 4`. Das erste Element trägt den Namen `namen` ist ein benannter character-Vektor (abgekürzt durch `chr`) mit 3 Elementen (Angabe `[1:3]`) usw.

Durch die komplexe Struktur sind Listen jedoch nicht so einfach zu handhaben. Mittels der Funktion `unlist()` können Listen daher in Vektoren (inkl. Elementnamen, soweit vorhanden) umgewandelt werden:

```
munich_facts_vector <- unlist(munich_facts)
munich_facts_vector
```

##	namen.hochdeutsch	namen.englisch	namen.bairisch
##	"München"	"Munich"	"Minga"
##	bundesland	gruendungsjahr	daten.einwohne
##	"Bayern"	"1158"	"1471508"
##	daten.geographie.flaeche_in_km2	daten.geographie.hoehe_NHN_in_m	
##	"310.7"	"519"	

2.3.3 Matrizen

Matrizen sind Vektoren, die in eine zweidimensionale Struktur, also Zeilen und Spalten, überführt werden und können mit der Funktion `matrix()` erstellt werden. Hierzu ist zusätzlich noch die Anzahl an Zeilen, in die der Vektor aufgeteilt werden soll, nötig:

```
x <- 1:10 # Zahlen von 1 bis 10 als Vektor
m <- matrix(x, nrow = 2) # Vektor in Matrix mit zwei Zeilen aufteilen
m
```

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

Alternativ können wir mehrere Vektoren mit den Funktionen `cbind()` spaltenweise (von `column`) und `rbind()` zeilenweise (von `row`) zu einer Matrix “zusammenkleben”.

```
x1 <- 1:4
x2 <- 5:8
x3 <- 0:3
m <- cbind(x1, x2, x3)
m

##      x1 x2 x3
## [1,] 1  5  0
## [2,] 2  6  1
## [3,] 3  7  2
## [4,] 4  8  3
```

2.3.3.1 Matrizen benennen

Auch Matrizen können benannt werden. Da wir nun aber eine zweidimensionale Struktur haben, können wir entsprechend auch Zeilen und Spalten einzeln benennen. Hierfür gibt es die Funktionen `rownames()` und `colnames()`, die analog zu `names()` verwendet werden:

```
colnames(m) <- c("spalte_1", "spalte_2", "spalte_3")
rownames(m) <- c("zeile_a", "zeile_b", "zeile_c", "zeile_d")
m

##      spalte_1 spalte_2 spalte_3
## zeile_a      1      5      0
## zeile_b      2      6      1
## zeile_c      3      7      2
## zeile_d      4      8      3
```

2.3.3.2 Nützliche Matrixfunktionen

`length()` funktioniert auch für Matrizen und gibt die Anzahl aller Elemente an. Interessieren wir uns dagegen für die Anzahl an Zeilen und Spalten, gibt die Funktion `dim()` Aufschluss, die einen Vektor mit der Anzahl der Zeilen und der Anzahl der Spalten ausgibt:

```
length(m)
dim(m)
```

```
## [1] 12
## [1] 4 3
```

Die Funktion `t()` transponiert die Matrix, dreht die Matrix also um 90 Grad und vertauscht somit Zeilen und Spalten:

```
t(m)

##      zeile_a zeile_b zeile_c zeile_d
## spalte_1      1      2      3      4
## spalte_2      5      6      7      8
## spalte_3      0      1      2      3
```

2.3.4 Dataframes

Mit Matrizen kommen wir der Datensatzstruktur, wie wir sie von SPSS oder Excel kennen, schon recht nahe. Allerdings repräsentieren Matrizen Vektoren und sind daher auf einen Objekttyp beschränkt. Diese Einschränkung hebt die Datenstruktur *Dataframe* auf, mit der gleich lange Vektoren unterschiedlichen Typs kombiniert werden können. Wir erstellen Dataframes mit der gleichnamigen Funktion `data.frame()`:

```
beatles_data <- data.frame(
  name = c("John", "Paul", "George", "Ringo"),
  surname = c("Lennon", "McCartney", "Harrison", "Starr"),
  born = c(1940, 1942, 1943, 1940)
)
beatles_data

##      name   surname born
## 1  John    Lennon 1940
## 2  Paul  McCartney 1942
## 3 George   Harrison 1943
## 4 Ringo     Starr 1940
```

Wenn wir mit tabellarischen Daten arbeiten, geschieht das also in der Regel mit Dataframes. Natürlich wäre es nicht zielführend, wenn wir diese immer von Hand erstellen müssten. Es gibt daher Funktionen, mit denen wir externe Dateien (z. B. CSV-, Excel- und sogar SPSS-Dateien) als Dataframes in R laden können. Wie wir externe Dateien laden, schauen wir uns zu einem späteren Zeitpunkt genauer an.

2.3.4.1 Beispiel-Dataframes

R enthält einige eingebaute Beispiel-Dataframes, mit denen Funktionen demonstriert und geübt werden können. Keiner davon hat auch nur einen geringen KW-Bezug, aber damit wir nun auch ohne große Erstellungs-Arbeit mit Dataframes arbeiten können, nutzen wir diese dennoch. Wir werden aber bald auch mit für Sie relevanteren Daten arbeiten, versprochen.

Diese Beispiel-Datensätze sind direkt als Objekte hinterlegt und können durch Eingabe des Objektnamens genutzt werden. Wir arbeiten nun mit dem Datensatz

`iris`, der Blütenblatt- und Kelchblatt-Daten zu je 50 Exemplaren dreier Spezies von Schwertlilien (englisch *iris*) umfasst. Wie gesagt, keinerlei KW-Bezug, aber immerhin ein Grund, um den Text kurz durch einige Blumenfotos aufzulockern.



Figure 2.2: Drei Schwertlilien-Spezies im `iris`-Datensatz, von links nach rechts: Borsten-Schwertlilie (*iris setosa*), verschiedenfarbige Schwertlilie (*iris versicolor*) und Virginia-Schwertlilie (*iris virginica*). Fotos: Radomił Binek, Danielle Langlois und Eric Hunt.

2.3.4.2 Arbeiten mit Dataframes

Dataframes basieren auf Listen und Vektoren (genau genommen ist ein Dataframe eine Liste von gleich langen Vektoren, die zweidimensional dargestellt wird). Entsprechend können wir eine Vielzahl der Funktionen, die wir bei Listen und Vektoren kennengelernt haben, auch auf Dataframes anwenden.

In der Regel haben wir Datensätze, die mehr als nur ein paar Fälle umfassen. Sie in der Konsole ausgeben zu lassen, ist daher nur bedingt sinnvoll. Besser ist es, Informationen über die Struktur des Datensatzes über Funktionen abzufragen.

Die `length()`-Funktion gibt bei Dataframes die Anzahl der Spalten zurück (wir erinnern uns: Dataframes sind Listen, deren Elemente die Spaltenvektoren sind; entsprechend ermittelt `length()` daher die Anzahl der Vektoren). Die Anzahl der Zeilen – mithin die Anzahl der Fälle – gibt die Funktion `nrow()` aus. Beide Werte gemeinsam können wir erneut über `dim()` ausgeben.

```
length(iris)
nrow(iris)
dim(iris)
```

```
## [1] 5
## [1] 150
```

```
## [1] 150 5
```

Wir sehen, dass der `iris`-Datensatz 5 Spalten (= Variablen) und 150 Zeilen (= Fälle) umfasst. Für einen Überblick bietet sich wie auch schon bei Listen die `str()`-Funktion an.

```
str(iris)
```

```
## 'data.frame': 150 obs. of 5 variables:
## $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
## $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
## $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
## $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
## $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

Der Datensatz umfasst also 4 numerische Variablen (jeweils Länge und Breite des Blüten- (*sepal*) und des Kelchblattes (*petal*)) sowie eine kategoriale Faktorvariable mit 3 Stufen, in der die Spezies des jeweiligen Exemplars festgehalten ist.

150 Fälle à 5 Variablen wären im Konsolenoutput sehr lang und unübersichtlich. Wollen wir dennoch in unsere Daten spähen, können wir uns mittels `head()` und `tail()` die ersten bzw. letzten Zeilen des Datensatzes ausgeben. Standardmäßig werden bei beiden Funktionen 6 Fälle angezeigt.

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

R bietet auch einen Viewer, mit dem der gesamte Datensatz ähnlich wie ein Tabellenblatt in Excel oder die Datenansicht in SPSS angezeigt wird. Hierzu führen wir die Funktion `View()` aus (großes V beachten), woraufhin in RStudio ein eigener Reiter mit der Datenansicht geöffnet wird.

```
View(iris)
```

In der Datenansicht können wir den Datensatz nach Variablen sortieren oder bestimmte Wertebereiche je Variable filtern; in der Fußzeile werden Strukturinformationen (Zeilen- und Spaltenanzahl) angezeigt. Fährt man mit dem Mauszeiger über die Kopfzeilen der Variablen, werden zudem weitere Informationen (Objekttyp und Wertebereich) eingeblendet.

Um einzelne Variablen (also Spalten bzw. die dahinterliegenden Vektoren) auszuwählen, können wir wie auch bei Listen das `$`-Zeichen nutzen:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.6	3.4	1.6	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
14	4.3	3.0	1.1	0.1	setosa
15	5.8	4.0	1.2	0.2	setosa
16	5.7	4.4	1.5	0.4	setosa
17	5.4	3.9	1.3	0.4	setosa
18	5.1	3.5	1.4	0.3	setosa
19	5.7	3.8	1.7	0.3	setosa
20	5.1	3.8	1.5	0.3	setosa

Showing 1 to 22 of 150 entries, 5 total columns

Figure 2.3: Die Datenansicht von R

```
iris$Sepal.Length
## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1 5.7 5
## [28] 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5
## [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1 6.3 6
## [82] 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6
## [136] 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

Das Resultat hat den Objekttyp Vektor. Wir können daher die uns bekannten Vektorfunktionen auf das Resultat anwenden. Um beispielsweise den Mittelwert der Kelchblattbreite zu erhalten, geben wir folgendes ein:

```
mean(iris$Petal.Width)
```

```
## [1] 1.199333
```

Auch die eckigen Klammern [] können genutzt werden, um flexibler nur bestimmte Teile des Datensatzes anzuzeigen. Dabei ist die zweidimensionale Struktur zu beachten – wir können zwei Werte bzw. Vektoren, getrennt durch ein ,, übergeben, die dann die Zeilen respektive die Spalten an wählt. Um etwa die ersten zehn Zeilen der Variablen Sepal.Length und Petal.Length auszuwählen, ist folgender Code nötig:

```
iris[1:10, c("Sepal.Length", "Petal.Length")]
```

```
##   Sepal.Length Petal.Length
## 1          5.1          1.4
```

```
## 2      4.9      1.4
## 3      4.7      1.3
## 4      4.6      1.5
## 5      5.0      1.4
## 6      5.4      1.7
## 7      4.6      1.4
## 8      5.0      1.5
## 9      4.4      1.4
## 10     4.9      1.5
```

Sollen lediglich Zeilen oder nur Spalten gefiltert werden, lassen wir den jeweiligen Wert vor (Zeilen) oder nach dem , (Spalten) leer. Folgender Code gibt die Zeilen 5-10 und alle Spalten aus:

```
iris[5:10,]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 7      4.6      3.4      1.4      0.3  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
```

Auch die Benennung von Dataframes läuft analog zu den bisherigen Objekttypen ab. Um etwa die Variablen einzudeutschen, können wir wieder `names()` nutzen⁷:

```
names(iris) <- c("bluetenblatt_laenge", "bluetenblatt_breite",
                 "kelchblatt_laenge", "kelchblatt_breite",
                 "spezies")
head(iris)
```

```
##   bluetenblatt_laenge bluetenblatt_breite kelchblatt_laenge kelchblatt_breite spezies
## 1      5.1              3.5          1.4          0.2  setosa
## 2      4.9              3.0          1.4          0.2  setosa
## 3      4.7              3.2          1.3          0.2  setosa
## 4      4.6              3.1          1.5          0.2  setosa
## 5      5.0              3.6          1.4          0.2  setosa
## 6      5.4              3.9          1.7          0.4  setosa
```

Falls Ihnen das nun umständlich erscheint – keine Sorge, wir werden, sobald wir uns ans richtige Datenmanagement begeben, Funktionen kennenlernen, die die obigen Schritte deutlich erleichtern. Für das Grundverständnis ist es aber wichtig, auch diese Art der Arbeit mit Dataframes kennenzulernen.

⁷Da Dataframes eine zweidimensionale Struktur darstellen, können wir auch die Funktionen `colnames()` und `rownames()`, die wir von den Matrizen kennen, verwenden, um die Spalten respektive Zeilen umzubenennen. `colnames()` und `names()` sind bei Dataframes äquivalent; Zeilennamen sind eher unüblich. In der Praxis wird daher meistens lediglich `names()` verwendet.

2.4 Übungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei und speichern diese als `ue2_nachname.R` ab. Antworten auf Fragen können Sie direkt als Kommentare in das Skript einfügen.

Übungsaufgabe 2.1. Objekttypen und Datenstrukturen I:

Erstellen Sie ein Objekt `myself`, das folgende Elemente enthält:

- `name`: Ihren Namen
- `born`: Ihr Geburtsjahr
- `from_bavaria`: Sind Sie in Bayern geboren?

Welche Datenstruktur ist hierfür am sinnvollsten? Welche Objekttypen haben die einzelnen Elemente?

Übungsaufgabe 2.2. Vektorfunktionen, Objekttypen und -umwandlung:

Führen Sie folgenden Code aus:

```
values <- c(1.2, 1.3, 0.8, 0.7, 0.7, 1.5, 1.1, 1.0, 1.1, 1.2, 1.1)
average <- mean(values)
above_average <- values > average
sum(above_average) / length(values)
```

Beschreiben Sie in eigenen Worten, was hier in jeder Zeile passiert. Was bedeutet das Resultat der letzten Codezeile? Warum wird hier überhaupt ein Resultat ausgeben? Schauen Sie sich hierzu an, was passiert, wenn Sie `above_average` in einen numerischen Vektor umwandeln.

Übungsaufgabe 2.3. Arbeiten mit Dataframes:

Im Beispiel-Datensatz `mtcars` sind einige Daten zu verschiedenen KfZ-Modellen hinterlegt. Beantworten Sie die folgenden Fragen zum Datensatz mittels Funktionen:

- Wie viele Variablen und Fälle befinden sich in dem Datensatz?
- Welche der drei Objekttypen (`numeric`, `character`, `logical`) kommen in dem Datensatz vor?
- Wie viele Zylinder haben die enthaltenen Fahrzeuge im Durchschnitt? (Zylinder: `cyl`)
- Erstellen Sie einen neuen Datensatz `cars_short`, der lediglich die Variablen `mpg` und `hp` enthält.