DR. CLAUDIUS GRÄBNER

R FÜR DIE SOZIO-ÖKONOMISCHE FORSCHUNG

Contents

Willkommen

Dieses Skript ist als Begleitung für die Lehrveranstaltung "Wissenschaftstheorie und Einführung in die Methoden der Sozioökonomie" im Master "Sozioökonomie" an der Universität Duisburg-Essen gedacht.

Es enthält grundlegende Informationen über die Funktion der Programmiersprache R (R Core Team, 2018). Einige Kapitel beziehen sich unmittelbar auf bestimmte Vorlesungstermine, andere sind als optionale Zusatzinformation gedacht. Gerade Menschen ohne Vorkenntnisse in R sollten unbedingt die ersten Kapitel vor dem vierten Vorlesungsterm lesen und verstehen. Bei Fragen können Sie sich gerne an Claudius Gräbner wenden.

Die folgende Tabelle gibt einen Überblick über die Kapitel und die dazugehörigen Vorlesungstermine:

Kapitel	Zentrale Inhalte	Verwandter Vorlesungstermin
1: Vorbemerkungen	Gründe für R; Besonderheiten von R	Vorbereitung
2: Vorbereitung	Installation und Einrichtung von R und R Studio, Projektstrukturierung	Vorbereitung
3: Erste Schritte in R	Grundlegende Funktionen von R; Objekte in R; Pakete	Vorbereitung
4: Ökonometrie I	Implementierung von uni- und multivariaten linearen Regressionsmodellen	T4 am 06.11.19
5: Datenaquise und -management	Einlesen und Schreiben sowie Manipulation von Datensätzen; deskriptive Statistik	T8 am 11.12.19
6: Visualisierung	Erstellen von Grafiken	T8 am 11.12.19

Kapitel	Zentrale Inhalte	Verwandter Vorlesungstermin
7: Ökonometrie II	Mehr Konzepte der Ökonometrie	T9-10 am 8.&15.1.20
8: Ausblick	Ausblick zu weiteren Anwendungsmöglichkeiten	Optional
A: Einführung in Markdown	Wissenschaftliche Texte in R Markdown schreiben	Optional; relevant für Aufgabenblätter
B: Einführung in Git und Github	Verwendung von Git und Github	Optional

Das Skript ist *work in progress* und jegliches Feedback ist sehr willkommen. Dafür wird im Moodle ein extra Bereich eingerichtet.

Änderungshistorie während des Semesters

An dieser Stelle werden alle wichtigen Updates des Skripts gesammelt. Die Versionsnummer hat folgende Struktur: `major.minor.patch` *Neue Kapitel erhöhen die* `minor` *Stelle, kleinere, aber signifikante Korrekturen werden als Patches gekennzeichnet.*

Datum	Version	Wichtigste Änderungen
XX.XX.19	0.1.0	Erste Version veröffentlicht

Lizenz

1
Vorbemerkungen

1.1 Warum R?

Im folgenden gebe ich einen kurzen Überblick über die Gründe, die uns bewegt haben den Methodenkurs auf R aufzubauen. Die Liste ist sicherlich nicht abschließend (siehe auch Wickham (2019)).

- Die R Community gilt als besonders freundlich und hilfsbereit. Gerade weil viele Menschen, die R benutzen praktizierende Datenwissenschaftler*innen sind werden praktische Probleme breit und konstruktiv in den einschlägigen Foren diskutiert und es ist in der Regel leicht Lösungen für Probleme zu finde, sobald man selbst ein bestimmtes Level an Programmierkenntnissen erlangt hat.
  - Auch gibt es großartige Online Foren und Newsletter, die es einem einfacher und unterhaltsamer machen, seine R Kenntnisse stetig zu verbessern und zusätzlich viele neue Dinge zu lernen. Besonders empfehlen kann ich R-Bloggers, eine Sammlung von Blog Artikeln, die R verwenden und neben Inspirationen für die Verwendung von R häufig inhaltlich sehr interessant sind; rweekly, ein Newsletter, der ebenfalls interessante Infos zu R enthält sowie die R-Ladies Community, die sich besonders das Empowerment von Minderheiten in der Programmierwelt zur Aufgabe gemacht hat.
  - Selbstverständlich werden zahlreiche R Probleme auch auf StackOverflow disktuiert. Häufig ist das der Ort, an dem man Antworten auf seine Fragen findet. Allerdings ist es gerade am Anfang unter Umständen schwierig die häufig sehr fortgeschrittenen Lösungen zu verstehen.
- R ist eine offene und freie Programmiersprache, die auf allen bekannten Betriebssystemen läuft. Im Gegensatz zu Programmen wie SPSS und STATA, für die Universitäten jedes Semester viele Tausend Euro bezahlen müssen und die dann umständlich über

Serverlizenzen abgerufen werden müssen. Auch für Studierende sind die Preise alles andere als gering. R dagegen ist frei und inklusiv, und auch Menschen mit weniger Geld können sie benutzen. Gerade vor dem Hintergrund der Rolle von Wissenschaft in einer demokratischen und freien Gesellschaft und in der Kooperation mit Wissenschaftler*innen aus ärmeren Ländern ist dies extrem wichtig.

- R verfügt über ein hervorragendes Package System. Das bedeutet, dass es recht einfach ist, neue Pakete zu schreiben und damit die Funktionalitäten von R zu erweitern. In der Kombination mit der Open Source Charakter von R bedeutet das, dass R nie wirklich *out of date* ist, und dass neuere Entwicklungen der Statistik und Datenwissenschaften, und immer mehr auch in der VWL, recht zügig in R implementiert werden. Insbesondere wenn es um statistische Analysen, *machine learning*, Visualisierungen oder Datenmanagement und -manipulation geht: für alles gibt es Pakete in R und irgendjemand hat ihr Problem mit hoher Wahrscheinlichkeit schon einmal gelöst und Sie können davon profitieren.
  - R ist - zusammen mit Python - mittlerweile die *lingua franca* im Bereich Statistik und Machine Learning.
- Integration mit Git, Markdown, Latex und anderen Tools erlaubt einen integrierten Workflow, in dem Sie im Optimalfall euer Paper in der gleichen Umgebung schreiben wie den Code für eure statistische Analyse. Diesen Vorteil werden Sie bereits bei der Bearbeitung der Aufgabenzettel genießen können, da diese in teilweise in R Markdown zu lösen und abzugeben sind. Das bedeutet, dass Coding und Schreiben der Antworten im gleichen Dokument vorgenommen werden können. Auch dieses Skript wurde vollständig in R Markdown geschrieben.
- R erlaubt sowohl objektorientierte als auch funktionale Programmierung.
- Für besondere Aufgaben ist es recht einfach R mit high-performance Sprachen wie C, Fortran oder C++ zu integrieren.

1.2 Besonderheiten von R

R ist keine typische Programmiersprache, denn sie vor allem von Statistiker*innen benutzt und weiterentwickelt, und nicht von Programmierer*innen. Das hat den Vorteil, dass die Funktionen oft sehr genau auf praktische Herausforderungen ausgerichtet sind und es für alle typischen statistischen Probleme Lösungen in R gibt. Gleichzeitig hat dies auch dazu geführt, dass R einige unerwünschte Eigenschaften

aufweist, da die Menschen, die Module für R programmieren keine 'genuinen' Programmierer*innen sind.

Im folgenden möchte ich einige Besonderheiten von R aufführen, damit Sie im Laufe Ihrer R-Karriere nicht negativ von diesen Besonderheiten überrascht werden. Während es sich für Programmier-Neulinge empfiehlt die Liste zu einem späteren Zeitpunkt zu inspizieren sollten Menschen mit Erfahrungen in anderen Sprachen gleich einen Blick darauf werfen.

- R wird dezentral über viele benutzergeschriebene Pakete ('libraries' oder 'packages') konstant weiterentwickelt. Das führt wie oben erwähnt dazu, dass R quasi immer auf dem neuesten Stand der statistischen Forschung ist. Gleichzeitig kann die schiere Masse von Paketen auch verwirrend sein, insbesondere weil es für die gleiche Aufgabe häufig deutlich mehr als ein Paket gibt. Das führt zwar auch zu einer positiven Konkurrenz und jede*r kann sich ihren Geschmäckern gemäß für das eine oder andere Paket entscheiden, es bringt aber auch mögliche Inkonsistenzen und schwerer verständlichen Code mit sich.

- Im Gegensatz zu Sprachen wie Python, die trotz einer enormen Anzahl von Paketen eine interne Konsistenz nicht verloren haben gibt es in R verschiedene 'Dialekte', die teilweise inkonsistent sind und gerade für Anfägner durchaus verwirrend sein können. Besonders die Unterscheidungen des `tidyverse`, einer Gruppe von Paketen, die von der R Studio Company sehr stark gepusht werden und vor allem zur Verarbeitung von Datensätzen gedacht sind, implementieren viele Routinen des 'klassischen R' ('base R') in einer neuen Art und Weise. Das Ziel ist, die Arbeit mit Datensätzen einfacher und leichter verständlich zu machen, allerdings wird die recht aggressive 'Vermarktung' und die teilweise inferiore Performance des Ansatzes auch kritisiert.[1]

- Da viele der Menschen, die R Pakete herstellen keine Programmierer sind, sind viele Pakete von einem Programmierstandpunkt aus nicht sonderlich effizient oder elegant geschrieben. Gleichzeitig gibt es aber auch viele Ausnahmen zu dieser Regel und viele Pakete werden über die Zeit hinweg signifikant verbessert.

- R an sich ist nicht die schnellste Programmiersprache, insbesondere wenn man seinen Code nicht entsprechend geschrieben hat. Auch bedarf eine R Session in der Regel recht viel Speicher. Hier sind selbst andere High-Level Sprachen wie Julia oder Python deutlich performanter, auch wenn Pakete wie data.table diesen Nachteil häufig abschwächen. Zudem ist er für die meisten Probleme, die Sozioökonom*innen in ihrer Forschungspraxis bearbeiten, irrelevant.

[1] Zum einen bin ich ein großer Fan von vielen tidyverse Paketen, gleichzeitig ist der Fokus von R Studio auf diese Pakete sehr gefährlich. Ich bin aber einer anderen Meinung was die Einsteigerfreundlichkeit vom `tidyverse` andgeht: meiner Meinung nach machen diese Pakete die Arbeit mit Datensätzen sehr einfach, und für kleine Datensätze (<500MB) benutze ich das `tidyverse` auch in meiner eigenen Forschung. Aufgrund der Einsteigerfreundlichkeit werden wir hier für die Arbeit mit Datensätzen trotz allem mit dem `tidyverse` arbeiten. Ich weise jedoch auf die kritische Diskussion im entsprechenden Kapitel des Skripts hin.

Alles in allem ist R jedoch eine hervorragende Wahl wenn es um quantitative sozialwissenschaftliche Forschung geht. Auch in der Industrie ist R extrem beliebt und wird im Bereich der *Data Science* nur noch von Python ernsthaft in den Schatten gestellt. Allerdings verwenden die meisten Menschen, die in diesem Bereich arbeiten, ohnehin beide Sprachen, da sie unterschiedliche Vor- und Nachteile haben. Entsprechend ist jede Minute, die Sie in das Lernen von R investieren eine exzellente Investition, egal wo Sie in Ihrem späteren Berufsleben einmal landen werden.

Das wichtigste am Programmieren ist in jedem Fall Spaß und die Bereitschaft zu und die Freude an der Zusammenarbeit mit anderen. Denn das hat R mit anderen offenen Sprachen wie Python gemeinsam: Programmieren und das Lösen von statistischen Fragestellungen sollte immer ein kollaboratives Gemeinschaftsprojekt sein!

2
Einrichtung

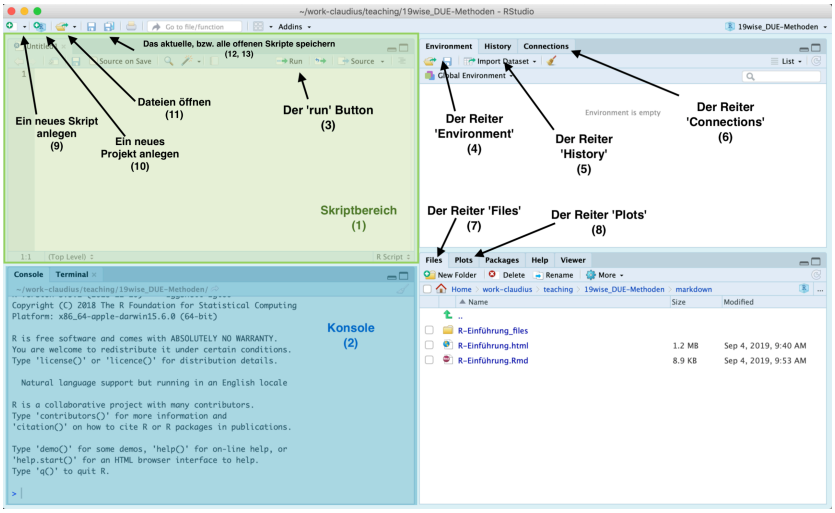
2.1 Installation von R und R-Studio

Die Installation von R ist in der Regel unproblematisch. Auf der R homepage wählt man unter dem Reiter 'Download' den Link 'CRAN' aus, wählt einen Server in der Nähe und lädt sich dann die R Software herunter. Danach folgt man den Installationshinweisen.

Im zweiten Schritt muss noch das Programm 'R-Studio' installiert werden. Hierbei handelt es sich um eine grafische Oberfläche für R, welche uns die Arbeit enorm erleichtern wird. Das Programm kann hier heruntergeladen werden. Bitte darauf achten 'RStuio Desktop' zu installieren.

2.2 Die R Studio Oberfläche

Nach dem Installationsprozess öffnen wir R Studio zum ersten Mal. Der folgende Screenshot zeigt die verschiedenen Elemente der Oberfläche, deren Funktion im folgenden kurz erläutert wird. Vieles ergibt sich hier aber auch durch *learning by doing*. Im folgenden werden nur die Bereiche der Oberfläche beschrieben, die am Anfang unmittelbar relevant für uns sind.



- Der **Skriptbereich** (1) ist ein Texteditor wie Notepad - nur mit zusätzlichen Features wie Syntax Highlighting für R, sodass es uns leichter fällt R Code zu schreiben. Hier werden wir unsere Skripte verfassen.

- Die **Konsole** (2) erlaubt es uns über R direkt mit unserem Computer zu interagieren. R ist eine Programmiersprache. Das bedeutet, wenn wir den Regeln der Sprache folgen und uns in einer für den Computer verständlicher Art und Weise ausdrücken, versteht der Computer was wir von ihm wollen und führt unsere Befehle aus. Wenn wir in die Konsole z.B. `2+2` eingeben, dann ist das valider R code. Wenn wir dann Enter drücken versteht der Computer unseren Befehl und führt die Berechnung aus. Die Konsole ist sehr praktisch um den Effekt von R Code direkt zu beobachten. Wenn wir etwas in der Console ausführen wollen, das wir vorher im **Skriptbereich** geschrieben haben, können wir den Text markieren und dann auf den Button `Run` (3) drücken: dann kopiert R Studio den Code in die Konsole und führt ihn aus.

- Für den Bereich oben rechts haben wir in der Standardkonfiguration von R Studio drei Optionen, die wir durch Klicken auf die Reiter auswählen können. Der Reiter **Environment** (4) zeigt uns alle bisher definierten Objekte an (mehr dazu später). Der Reiter **History** (5) zeigt an, welchen Code wir in der Vergangenheit ausgeführt haben. Der Reiter **Connections** (6) braucht uns aktuell nicht zu interessieren.

- Auch für den Bereich unten rechts haben wir mehrere Optionen: Der Bereich **Files** (7) zeigt uns unser Arbeitsverzeichnis mit allen Ordnern und Dateien an. Das ist das gleiche, was wir auch über

den File Explorer unserer Betriebssystems sehen würden. Der Bereich **Plots** (8) zeigt uns eine Vorschau der Abbildungen, die wir durch unseren Code produzieren. Die anderen Bereiche brauchen uns aktuell noch nicht zu interessieren.

- Wenn wir ein neues R Skript erstellen wollen, können wir das über den Button **Neu** (9) erledigen. Wir klicken darauf und wählen die Option 'R Skript'. Mit den alternativen Dateiformaten brauchen wir uns aktuell nicht beschäftigen.
- Der Botton **Neues Projekt anlegen** (10) erstellt eine neues R Studio Projekt - mehr dazu in Kürze.
- Der Button **Öffnen** (11) öffnet Dateien im Skriptbereich.
- Die beiden Buttons **Speichern** (12) und **Alles speichern** (13) speichern das aktuelle, bzw. alle im Skriptbereich geöffnenten Dateien.

Die restlichen Buttons und Fenster in R Studio werden wir im Laufe der Zeit kennenlernen.

Es macht Sinn, sich einmal die möglichen Einstellungsmöglichkeiten für R Studio anzuschauen und ggf. eine andere Darstellungsversion zu wählen.

2.3 Einrichtung eines R Projekts

Im folgenden werden wir lernen wie man ein neues R Projekt anlegt, R Code schreiben und ausführen kann.

Wann immer wir ein neues Programmierprojekt starten, sollten wir dafür einen eigenen Ordner anlegen und ein so genannten 'R Studio Projekt' erstellen. Das hilft uns den Überblick über unsere Arbeit zu behalten, und macht es einfach Code untereinander auszutauschen.

Ein Programmierprojekt kann ein Projekt für eine Hausarbeit sein, die Mitschriften für eine Vorlesungseinheit, oder einfach der Versuch ein bestimmtes Problem zu lösen, z.B. einen Datensatz zu visualisieren.

Die Schritte zur Erstellung eines solchen Projekts sind immer die gleichen:

1. Einen Ordner für das Projekt anlegen.
2. Ein R-Studio Projekt in diesem Ordner erstellen.
3. Relevante Unterordner anlegen.

Wir beschäftigen uns mit den Schritten gleich im Detail, müssen vorher aber noch die folgenden Konzepte diskutieren: (1) das Konzept eines *Arbeitsverzeichnisses* (*working directory*) und (2) die Unterscheidnug zwischen *absoluten* und *relativen* Pfaden.

2.3.1 Arbeitsverzeichnisse und Pfade

Das **Arbeitsverzeichnis** ist ein Ordner auf dem Computer, in dem R standardmäßig sämtlichen Output speichert und von dem aus es auf Datensätze und anderen Input zugreift. Wenn wir mit Projekten arbeiten ist das Arbeitsverzeichnis der Ordner, in dem das R-Projektfile abgelegt ist, ansonsten ist es euer Benutzerverzeichnis. Wir können uns das Arbeitsverzeichnis mit der Funktion `getwd()` anzeigen lassen. In meinem Fall ist das Arbeitsverzeichnis das folgende:

```
#> [1] "/Users/claudius/work-claudius/general/paper-projects/packages/SocioEconMethodsR"
```

Wenn ich R nun sagen würde ein File unter dem Namen `test.pdf` speichern, würde es am folgenden Ort gespeichert werden:

```
#> [1] "/Users/claudius/work-claudius/general/paper-projects/packages/SocioEconMethodsR/test.pdf"
```

R geht in einem solchen Fall immer vom Arbeitsverzeichnis aus. Da wir im vorliegenden Fall den Speicherort relativ zum Arbeitsverzeichnis angegeben haben, sprechen wir hier von einem **relativen Pfad**.

Alternativ können wir den Speicherort auch als **absoluten Pfad** angeben. In diesem Fall geben wir den kompletten Pfad, ausgehend vom Root Verzeichnis des Computers, an. Wir würden R also *explizit* auffordern, das File an foldengem Ort zu speichern:

```
#> [1] "/Users/claudius/work-claudius/general/paper-projects/packages/SocioEconMethodsR/test.pdf"
```

Wir werden hier **immer** relative Pfade verwenden. Relative Pfade sind fast immer die bessere Variante, da es uns erlaubt den gleichen Code auf verschiedenen Computern zu verwenden. Denn wiw man an den absoluten Pfaden erkennen kann, sehen diese auf jedem Computer anders aus und es ist dementsprechend schwierig, Code miteinander zu teilen.

Wir lernen mehr über dieses Thema wenn wir uns später mit Dateninput und -output beschäftigen.

2.3.2 Schritt 1: Projektordner anlegen

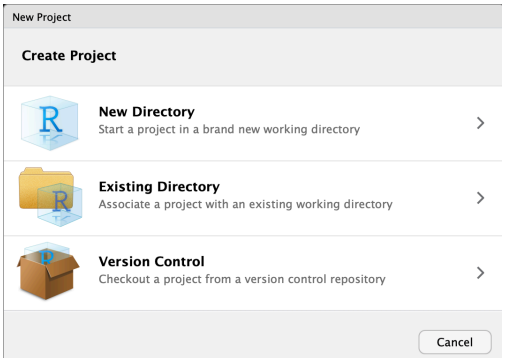
Zuerst müssen Sie sich für einen Ordner auf Ihrem Computer entscheiden, in dem alle Daten, die mit ihrem Projekt zu tun haben, also Daten, Skripte, Abbildungen, etc. gespeichert werden sollen und diesen Ordner gegebenenfalls neu erstellen. Es macht Sinn, einen solchen Ordner mit einem informativen Namen ohne Leer- und Sonderzeichen zu versehen, z.B. `SoSe19-Methodenkurs`.

Dieser Schritt kann theoretisch auch gemeinsam mit Schritt 2 erfolgen.

2.3.3 Schritt 2: Ein R-Studio Projekt im Projektordner erstellen

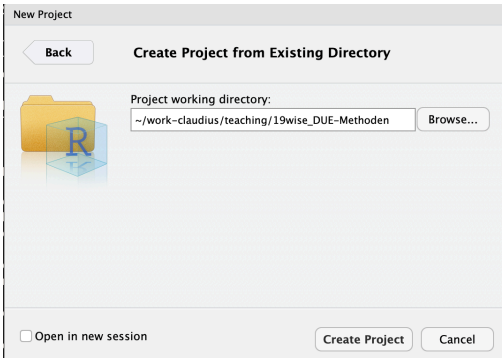
Wir möchten nun R Studio mitteilen den in Schritt 1 erstellten Ordner als R Projekt zu behandeln. Damit wird nicht nur dieses Ordner als Root-Verzeichnis festgelegt, man kann auch die Arbeitshistorie eines Projekts leich wiederherstellen und es ist einfacher, das Projekt auf verschiedenen Computern zu bearbeiten.

Um ein neues Projekt zu erstellen klicken Sie in R Studio auf den Button `Neues Projekt` (Nr. 10 in der obigen Abbildung) und Sie sollten folgendes Fenster sehen:

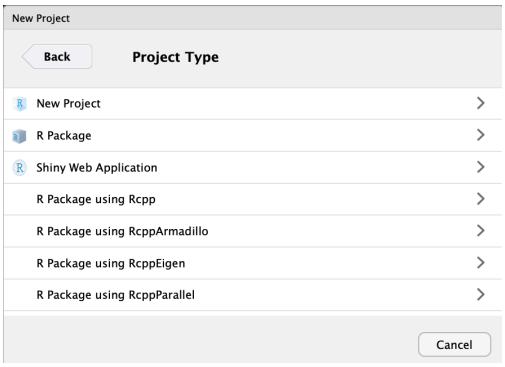


Falls Sie in Schritt 1 den Projektordner bereits erstellt haben wählen Sie hier `Existing Directory`, ansonsten erstellen Sie einen neuen Projektordner gleich mit dem Projektfile mit indem Sie `New Directory` auswählen.

Falls Sie `Existing Directory` gewählt haben, wählen Sie in folgendem Fenster einfach den vorher erstellten Ordner aus und klickt auf `Create Project`.



Falls Sie `New Directory` gewählt habt landen Sie auf folgendem Fenster:



Hier wählen Sie `New Project` aus, geben dem Projekt in folgenden Fenster einen Namen (das wird der Name des Projektordners sein), wählen den Speicherort für den Ordner aus und klicken auf `Create Project`.

In beiden Fällen wurde nun ein Ordner erstellt, in dem sich ein File `*.Rproj` befindet. Damit ist die formale Erstellung eines Projekts abgeschlossen. Es empfiehlt sich jedoch dringend gleich eine sinnvolle Unterordnerstruktur mit anzulegen.

2.3.4 Schritt 3: Relevante Unterordner erstellen

Eine sinnvolle Unterordnerstruktur hilf (1) den Überblick über das eigene Projekt nicht zu verlieren, (2) mit anderen über verschiedene Computer hinweg zu kollaborieren und (3) Kollaborationsplattformen wie Github zu verwenden und replizierbare und für andere nachvollziehbare Forschungsarbeit zu betreiben.

Die folgende Ordnerstruktur ist eine Empfehlung. In manchen Projekten werden Sie nicht alle hier vorgeschlagenen Unterordner brauchen, in anderen bietet sich die Verwendung von mehr Unterordnern an. Nichtsdestotrotz ist es ein guter Ausgangspunkt, den ich in den meisten meiner Forschungsprojekte auch so verwende.

Insgesamt sollten die folgenden Ordner im Projektordner erstellt werden:

- Ein Ordner `data`, der alle Daten enthält, die im Rahmen des Projekts verwendet werden. Hier empfiehlt es sich zwei Unterordner anzulegen: Einen Ordner `raw`, der die Rohdaten enthält, so wie sie aus dem Internet runtergeladen wurden. Diese Rohdaten sollten **niemals** verändert werden, ansonsten wird Ihre Arbeit nicht vollständig replizierbar werden und es kommt ggf. zu irreparablen Schäden. Alle Veränderungen der Daten sollten durch Skripte dokumentiert werdenn, die die Rohdaten als Input, und einen modifizierten Datensatz als Output generieren. Dieser modifizierte Datensatz sollte dann im Unterordner `tidy` gespeichert werden.

Beispiel: Sie laden sich Daten zum BIP in Deutschland von Eurostat und Daten zu Arbeitslosigkeit von AMECO herunter. Beiden Datensätze sollten im Unterordner `data/raw` gespeichert werden. Mit einem Skript lesen Sie beide Datensätze ein und erstellen den kombinierten Datensatz `macro_data.csv`, den Sie im Ordner `data/tidy` speichern und für die weitere Analyse verwenden. Dadurch kann jede*r nachvollziehen wie die von Ihnen verwendeten Daten sich aus den Rohdaten ergeben haben und Ihre Arbeit bleibt komplett transparent.

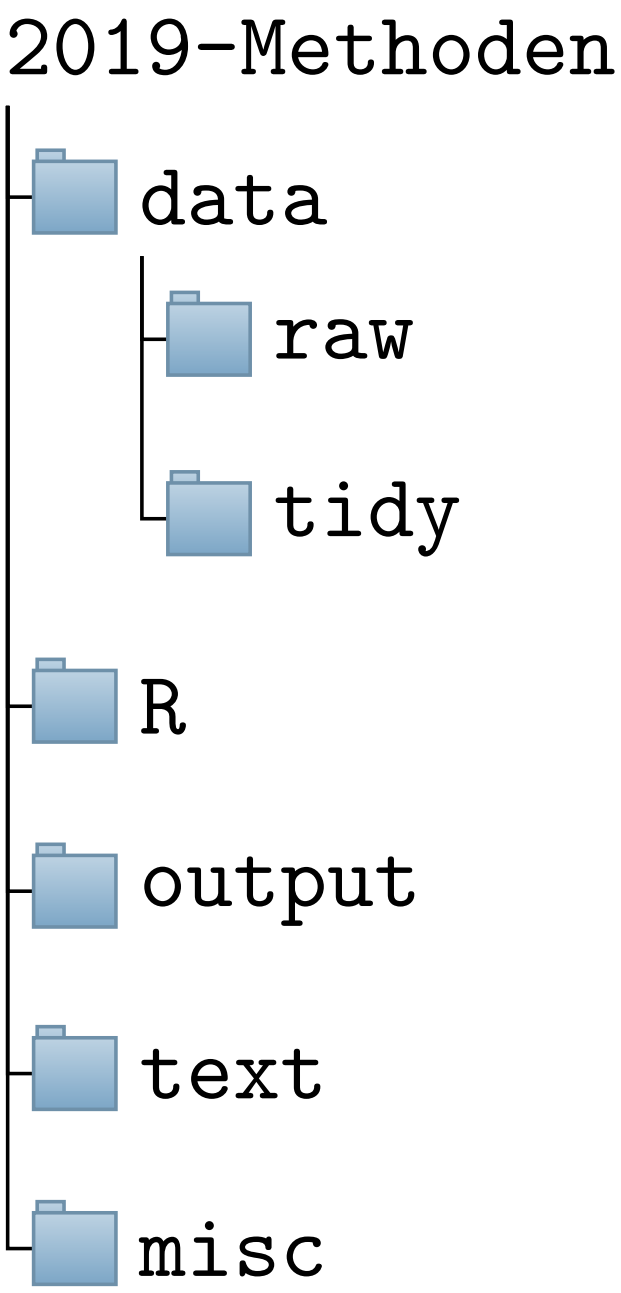
- Ein Ordner `R`, der alle R Skripte enthält, also alle Textdokumente, die R Code enthalten.

- Ein Ordner `output`, in dem der Output ihrer Berechnungen, z.B. Tabellen oder Plots gespeichert werden können. Der Inhalt dieses Ordners sollte sich komplett mit den Inhalten der Ordner `data` und `R` replizieren lassen.

- Ein Ordner `text`, in dem Sie Ihre Verschriftlichungen speichern, z.B. das eigentliche Forschungspapier, ihre Hausarbeit oder Ihre Vorlesungsmitschriften.

- Einen Ordner `misc` in den Sie alles packen, was in keinen der anderen Ordner passt. Ein solcher Ordner ist wichtig und Sie sollten nicht zuordbare Dateien nie in den Projektordner als solchen speichern.

Wenn wir annehmen unser Projektordner heißt `2019-Methoden` ergibt sich damit insgesamt folgende Ordner und Datenstruktur:



2.4 Abschließende Bemerkungen

Eine gute Ordnerstruktur ist nicht nur absolut essenziell um selbst einen Überblick über seine Forschungsprojekte zu behalten, sondern auch wenn man mit anderen Menschen kollaborieren möchte. In einem

solchen Fall sollte man auf jeden Fall eine Versionskontrolle wie Git und GitHub verwenden. Wir werden uns damit im nächsten Semester genauer beschäftigen, aber Sie werden merken, dass die Kollaboration durch eine gut durchdachte Ordnerstruktur massiv erleichtert wird.

3
Erste Schritte in R

Nach diesen (wichtigen) Vorbereitungsschritten wollen wir nun mit dem eigentlichen Programmieren anfangen. Zu diesem Zweck müssen wir uns mit der Syntax von R vertraut machen, also mit den Regeln, denen wir folgen müssen, wenn wir Code schreiben, damit der Computer versteht, was wir ihm eigentlich in R sagen wollen.

3.1 Befehle in R an den Computer übermitteln

Grundsätzlich können wir über R Studio auf zwei Arten mit dem Computer "kommunizieren": über die Konsole direkt, oder indem wir im Skriptbereich ein Skript schreiben und dies dann ausführen.

Als Beispiel für die erste Möglichkeit wollen wir mit Hilfe von R die Zahlen `2` und `5` miteinander addieren. Zu diesem Zweck können wir einfach `2 + 2` in die Konsole eingeben, und den Befehl mit 'Enter' an den Computer senden. Da es sich beim Ausdruck `2 + 3` um korrekten R Code handelt, 'versteht' der Computer was wir von ihm wollen und gibt uns das entsprechende Ergebnis aus:

```
2 + 3
```

```
#> [1] 5
```

Auf diese Art und Weise könne wir R als einfachen Taschenrechner verwenden, denn für alle einfachen mathematischen Operationen können wir bestimmte Symbole als Operatoren verwenden. An dieser Stelle sei noch darauf hingewiesen, dass das Symbol `#` in R einen Kommentar einleitet, das heißt alles was in einer Zeile nach `#` steht wird vom Computer ignoriert und man kann sich einfach Notizen in seinem Code machen.

```
2 + 2  # Addition
```

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

```
2/2  # Division
```

```
#> [1] 1
```

```
4 * 2  # Multiplikation
```

```
#> [1] 8
```

```
3^2  # Potenzierung
```

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

Alternativ können wir die Befehle in einem Skript aufschreiben, und dieses Skript dann ausführen. Während die Interaktion über die Konsole sinnvoll ist um die Effekte bestimmter Befehle auszuprobieren, bietet sich die Verwendung von Skripten an, wenn wir mit den Befehlen später weiter arbeiten wollen, oder sie anderen Menschen zugänglich zu machen. Den das Skript können wir als Datei auf unserem Computer speichern, vorzugsweise im Unterordner `R` unseres R-Projekts (siehe Abschnitt Relevante Unterordner erstellen), und dann später weiterverwenden.

Die Berechnungen, die wir bisland durchgeführt haben sind zugegebenermaßen nicht sonderlich spannend. Um fortgeschrittene Operationen in R durchführen und verstehen zu können müssen wir uns zunächst mit den Konzepten von **Objekten**, **Funktionen** und **Zuweisungen** beschäftigen.

3.2 Objekte, Funktionen und Zuweisungen

> To understand computations in R, two slogans are helpful: Everything that exists is an object. Everything that happens is a function call.
> —John Chambers

Mit der Aussage ‘Alles in R ist ein Objekt’ ist gemeint, dass jede Zahl, jede Funktion, oder jeder Buchstabe in R ein Objekt ist, das irgendwo auf dem Speicher Ihres Rechners abgespeichert ist.

In der Berechnung `2 + 3` ist die Zahl `2` genauso ein Objekt wie die Zahl `3` und die Additionsfunktion, die durch den Operator `+` aufgerufen wird.

Mit der Aussage ‘Alles was in R passiert ist ein Funktionsaufruf’ ist gemeint, dass wenn wir R eine Berechnung durchführen lassen, tun wir dies indem wir eine Funktion aufrufen.

Funktionen sind Algorithmen, die bestimmte Routinen auf einen *Input* anwenden und dabei einen *Output* produzieren. Die Additionsfunktion, die wir in der Berechnung `2 + 3` aufgerufen haben hat als Input die beiden Zahlen `2` und `3` aufgenommen, hat auf sie die Routine der Addition angewandt und als Output die Zahl `5` ausgegeben. Der

Output `5` ist dabei in R genauso ein Objekt wie die Inputs `2` und `3`, sowie die Funktion `+`.

Ein 'Problem' ist, dass R im vorliegenden Falle den Output der Berechnung zwar ausgibt, wir danach aber keinen Zugriff darauf mehr haben:

```
2 + 3
```

```
#> [1] 5
```

Falls wir den Output weiterverwenden wollen, macht es Sinn, dem Output Objekt einen Namen zu geben, damit wir später wieder darauf zugreifen können. Der Prozess einem Objekt einen Namen zu Geben wird **Zuweisung** oder **Assignment** genannt und durch die Funktion `assign` vorgenommen:

```
assign("zwischenergebnis", 2 + 3)
```

Wir können nun das Ergebnis der Berechnung `2 + 3` aufrufen, indem wir in R den Namen des Output Objekts eingeben:

```
zwischenergebnis
```

```
#> [1] 5
```

Da Zuweisungen so eine große Rolle spielen und sehr häufig vorkommen gibt es auch für die Funktion `assign` eine Kurzschreibweise, nämlich `<-`. Entsprechend sind die folgenden beiden Befehle äquivalent:

```
assign("zwischenergebnis", 2 + 3)
zwischenergebnis <- 2 + 3
```

Entsprechend werden wir Zuweisungen immer mit dem `<-` Operator durchführen. [1]

Wir können in R nicht beliebig Namen vergeben. Gültige (also: syntaktisch korrekte) Namen …

- enthalten nur Buchstaben, Zahlen und die Symbole `.` und `_`
- fangen nicht mit `.` oder einer Zahl an!

Zudem gibt es einige Wörter, die schlicht nicht als Name verwendet werden dürgen, z.B. `function`, `TRUE`, oder `if`. Die gesamte Liste verbotener Worte kann mit dem Befehl `?Reserved` ausgegeben werden.

Wenn man einen Namen vergeben möchte, der nicht mit den gerade formulierten Regeln kompatibel ist, gibt R eine Fehlermeldung aus:

```
TRUE <- 5
```

```
#> Error in TRUE <- 5: invalid (do_set) left-hand side to assignment
```

[1] Theoretisch kann `<-` auch andersherum verwendet werden: `2 + 3 -> zwischenergebnis`. Das mag zwar auf den ersten Blick intuitiver erscheinen, da das aus `2 + 3` resultierende Objekt den Namen `zwischenergebnis` bekommt, also immer erst das Objekt erstellt wird und dann der Name zugewiesen wird, es führt jedoch zu deutlich weniger lesbarem Code und sollte daher nie verwendet werden. Ebensoweinig sollten Zuweisungen durch den `=` Operatur vorgenommen werden, auch wenn es im Fall `zwischenergebnis = 2 + 3` funktionieren würde.

Zudem sollte man folgendes beachten:

- Namen sollten kurz und informativ sein; entsprechen ist `sample_mean` ein guter Name, `shit15_2` dagegen eher weniger
- Man sollte **nie Umlaute in Namen verwenden**
- R ist *case sensitive*, d.h. `mean_value` ist ein anderer Name als `Mean_Value`
- Auch wenn möglich, sollte man nie von R bereit gestellte Funktionen überschreiben. Eine Zuweisung wie `assign <- 2` ist zwar möglich, führt in der Regel aber zu großem Unglück, weil man nicht mehr ganz einfach auf die zugrundeliegende Funktion zurückgreifen kann.

  Hinweis: Alle aktuellen Namenszuweisungen sind im Bereich `Environment` in R Studio (Nr. 4 in der Abbildung oben) aufgelistet und können durch die Funktion `ls()` angezeigt werden.

  Hinweis: Ein Objekt kann mehrere Namen haben, aber kein Name kann zu mehreren Objekten zeigen, da im Zweifel eine neue Zuweisung die alte Zuweisung überschreibt:

```
x <- 2
y <- 2  # Das Objekt 2 hat nun zwei Namen
print(x)
```

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

```
print(y)
```

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

```
x <- 4  # Der Name 'x' zeigt nun zum Objekt '4', nicht mehr zu '2'
print(x)
```

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

  Hinweis: Wie Sie vielleicht bereits bemerkt haben wird nach einer Zuweisung kein Wert sichtbar ausgegeben:

```
2 + 2  # Keine Zuweisung, R gibt das Ergebnis in der Konsole aus
```

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

```
x <- 2 + 2  # Zuweisung, R gibt das Ergebnis in der Konsole nicht aus
```

3.3 Zusammenfassung

- Wir können Befehle in R Studio an den Computer übermitteln indem wir (a) den R Code in die Konsole schreiben und Enter drücken oder (b) den Code in ein Skript schreiben und dann ausführen

- Alles was in R *existiert* ist ein Objekt, alles was in R *passiert* ist ein Funktionsaufruf
- Wir können einem Objekt mit Hilfe von `<-` einen Namen geben und dann später wieder aufrufen. Den Prozess der Namensgebung nennen wir **Assignment** und wir können uns alle aktuell von uns vergebenen Namen mit der Funktion `ls()` anzeigen lassen.
- Eine Funktion ist ein Objekt, das auf einen Input eine bestimmte Routine anwendet und einen Output produziert

An dieser Stelle sei noch auf die Hilfefunktion `help()` hingewiesen. Falls Sie Informationen über ein Objekt bekommen wollen können Sie so weitere Informationen bekommen. Wenn Sie z.B. genauere Informationen über die Verwendung der Funktion `assign` erhalten wollen, können Sie Folgendes eingeben:

```
help(assign)
```

3.4 Grundlegende Objeke in R

Wir haben bereits gelernt, dass alles was in R existiert ein Objekt ist. Wir haben aber auch schon gelernt, dass es unterschiedliche Typen von Objekten gibt: Zahlen, wie `2` oder `3` und Funktionen wie `assign`.[2] Tatsächlich gibt es noch viel mehr Arten von Objekten. Ein gutes Verständnis der Objektarten ist Grundvoraussetzung später anspruchsvolle Programmieraufgaben zu lösen. Daher wollen wir uns im Folgenden mit den wichtigsten Objektarten in R auseinandersetzen.

[2] Wie wir unten lernen werden sind `2` und `3` in erster Linie keine Zahlen, sondern Vektoren der Länge 1, und gelten erst in nächster Instanz als 'Zahl' (genauer: 'double').

3.4.1 Funktionen

Wie oben bereits kurz erwähnt handelt es sich bei Funktionen um Algorithmen, die bestimmte Routinen auf einen *Input* anwenden und dabei einen *Output* produzieren.

Die Funktion `log()` zum Beispiel nimmt als Input eine Zahl und gibt als Output den Logarithmus dieser Zahl aus:

```
log(2)
```

```
#> [1] 0.6931472
```

Eine Funktion aufrufen

In R gibt es prinzipiell vier verschiedene Arten Funktionen aufzurufen. Nur zwei davon sind allerdings aktuell für uns relevant.

Die bei weitem wichtigste Variante ist die so genannte *Prefix-Form*. Dies ist die Form, die wir bei der überwältigenden Anzahl von Funktionen verwenden werden. Wir schreiben hier zunächst den Namen der Funktion (im Folgenden Beispiel `assign`), dann in Klammern und mit

Kommata getrennt die Argumente der Funktion (hier der Name `test` und die Zahl `2`):

```
assign("test", 2)
```

Ein hin und wieder auftretende Form ist die so genannte *Infix-Form*. Hier wird der Funktionsname zwischen die Argumente geschrieben. Dies ist, wie wir oben bereits bemerkt haben, bei vielen mathematischen Funktionen wie `+`, `-` oder `/` der Fall. Streng genommen ist die die Infix-Form aber nur eine *Abkürzung*, denn jeder Funktionsaufruf in Infix-Form kann auch in Prefix-Form geschrieben werden, wie folgendes Beispiel zeigt:

```
2 + 3
```

```
#> [1] 5
```

```
2 + 3
```

```
#> [1] 5
```

Die Argumente einer Funktion

Die Argumente einer Funktion stellen zum einen den *Input* für die in der Funktion implementierten Routine dar.

Die Funktion `sum` zum Beispiel nimmt als Argumente eine beliebige Anzahl an Zahlen (ihr 'Input') und berechnet die Summe dieser Zahlen:

```
sum(1, 2, 3, 4)
```

```
#> [1] 10
```

Darüber hinaus akzeptiert `sum()` noch ein *optionales Argument*, `na.rm`, welches entweder den Wert `TRUE` oder `FALSE` annehmen kann. Wenn wir das Argument nicht explizit spezifizieren nimmt es automatisch `FALSE` als den Standardwert an.

Dieses optionale Argument ist kein klassischer Input, sondern kontrolliert das genaue Verhalten der Funktion. Im Falle von `sum()` werden fehlende Werte, so genannte `NA` (siehe unten) ignoriert bevor die Summe der Inputs gebildet wird wenn `na.rm` den Wert `TRUE` hat:

```
sum(1, 2, 3, 4, NA)
```

```
#> [1] NA
```

```
sum(1, 2, 3, 4, NA, na.rm = TRUE)
```

```
#> [1] 10
```

Wenn wir wissen wollen, welche Argumente eine Funktion akzeptiert ist es immer eine gute Idee über die Funktion `help()` einen Blick in die Dokumentation zu werfen!

Im Falle von `sum()` sehen wir hier sofort, dass die Funktion neben den zu addierenden Zahlen ein optionales Argument `na.rm` akzeptiert, welches den Standardwert `FALSE` annimmt.

Eigene Funktionen definieren

Sehr häufig möchten wir selbst Funktionen definieren. Das können wir mit dem reservierten Keyword `function` machen. Als Beispiel wollen wir eine Funktion `pythagoras` definieren, die als Argumente die Seitenlängen der Katheten eines rechtwinkligen Dreiecks annimmt und über den Satz des Pythagoras die Länge der Hypothenuse bestimmt:

```
pythagoras <- function(kathete_1, kathete_2) {
    hypo_quadrat <- kathete_1^2 + kathete_2^2
    l_hypothenuse <- sqrt(hypo_quadrat)  # sqrt() zieht die Quadratwurzel
    return(l_hypothenuse)
}
```

Wir definieren eine Funktion durch die Funktion `function()`. In der Regel beginnen wir die Definition indem wir der zu erstellenden mit einem Namen assoziieren (hier: 'pythagoras') damit wir sie später auch verwenden können.

Die Argumente für `function` sind dann die Argumente, welche die zu definierende Funktion annehmen soll, in diesem Fall `kathete_1` und `kathete_2`. Danach beginnen wir den 'function body', also den Code für die Routine, welche die Funktion ausführen soll, mit einer geschweiften Klammer.

Innerhalb des *function bodies* wird dann die entsprechende Routine implementiert. Im vorliegenden Beispiel definieren wir zunächst die Summe der Werte von `kathete_1` und `kathete_2` als ein Zwischenergebnis, welches hier `hypo_quadrat` genannt wird. Dies ist der häufig unter $c^2 = a^2 + b^2$ bekannte Teil des Satz von Pythagoras. Da wir an der 'normalen' Länge der Hypothenuse interesssiert sind, ziehen wir mit der Funktion `sqrt()` noch die Wurzel von `hypo_quadrat`, und geben dem resultierenden Objekt den Namen `l_hypothenuse`, welches in der letzten Zeile mit Hilfe des Keywords `return` als der Wert definiert wird, den die Funktion als Output ausgibt.[3]

Am Ende der Routine kann man mit dem Keyword `return` explizit machen welchen Wert die Funktion als Output ausgeben soll. Wenn wir die Funktion nun aufrufen wird die oben definierte Routine ausgeführt:

```
pythagoras(2, 4)
```

```
#> [1] 4.472136
```

[3] Das ist strikt genommen nicht notwendig, aber der Übersichtlichkeit werden wir immer `return` verwenden. Eine interessante Debatte darüber ob man `return` verwenden sollte oder nicht findet sich hier.

Beachten Sie, dass alle Objet Namen, die innerhalb des *function bodies* verwendet werden gehen nach dem Funktionsaufruf verloren:[4] Deswegen kommt es im vorliegenden Falle zu einem Fehler, da `hypo_quadrat` nur innerhalb des *function bodies* existiert:

[4] Das liegt daran, dass Funktionen ihr eigenes environment haben.

```
pythagoras <- function(kathete_1, kathete_2) {
    hypo_quadrat <- kathete_1^2 + kathete_2^2
    l_hypothenuse <- sqrt(hypo_quadrat)  # sqrt() zieht die Quadratwurzel
    return(l_hypothenuse)
}
x <- pythagoras(2, 4)
hypo_quadrat
```

```
#> Error in eval(expr, envir, enclos): object 'hypo_quadrat' not found
```

Es ist immer eine gute Idee, die selbst definierten Funktionen zu dokumentieren - nicht nur wenn wir sie auch anderen zur Verfügung stellen wollen, sondern auch damit wir selbst nach einer möglichen Pause unseren Code noch gut verstehen können. Nichts ist frustrierender als nach einer mehrwöchigen Pause viele Stunden investieren zu müssen, den eigens programmierten Code zu entschlüsseln!

Die Dokumentation von Funktionen kann mit Hilfe von einfachen Kommentaren erfolgen, ich empfehle jedoch sofort sich die hier beschriebenen Konventionen anzugewöhnen. In diesem Falle würde eine Dokumentation unserer Funktion `pythagoras` folgendermaßen aussehen:

```
#' Berechne die Länge der Hypothenuse in einem rechtwinkligen Dreieck
#'
#' Diese Funktion nimmt als Argumente die Längen der beiden Katheten eines
#'  rechtwinkligen Dreiecks und berechnet daraus die Länge der Hypothenuse.
#' @param kathete_1 Die Länge der ersten Kathete
#' @param kathete_2 Die Länge der zweiten Kathete
#' @return Die Länge der Hypothenuse des durch a und b definierten
#'  rechtwinkligen Dreieckst
pythagoras <- function(kathete_1, kathete_2) {
    hypo_quadrat <- kathete_1^2 + kathete_2^2
    l_hypothenuse <- sqrt(hypo_quadrat)  # sqrt() zieht die Quadratwurzel
    return(l_hypothenuse)
}
```

Die Dokumentation wird also direkt vor die Definition der Funktion gesetzt. In der ersten Zeile gibt man der Funktion einen maximal einzeiligen Titel, der nicht länger als 80 Zeichen sein sollte und die Funktion prägnant beschreibt.

Dann, nach einer Lehrzeile wird genauer beschrieben was die Funktion macht. Danach werden die Argumente der Funktion beschrieben.

Für jedes Argument beginnen wir die Reihe mit `@param`, gefolgt von dem Namen des Arguments und dann einer kurzen Beschreibung.

Nach den Argumenten beschreiben wir noch kurz was der Output der Funktion ist. Diese Zeile wird mit `@return` begonnen.

Die Dokumentation einer Funktion sollte also zumindest die Parameter und die Art des Outputs erklären.

Gründe für die Verwendung eigener Funktionen

Eigene Funktionen zu definieren ist in der Praxis extrem hilfreich und es ist empfehlenswert Routinen, die mehrere Male verwendet werden grundsätzlich als Funktionen zu schreiben. Dafür gibt es mehrere Gründe:

1. **Der Code wird kürzer und transparenter.** Zwar ist kurzer Code nicht notwendigerweise leichter zu verstehen als langer, aber Funktionen können besonders gut dokumentiert werden (am besten indem man den hier beschriebenen Konventionen folgt).
2. **Funktionen bieten Struktur.** Funktionen fassen in der Regel Ihre Vorstellung davon zusammen, wie ein bestimmtes Problem zu lösen ist. Da man sich diese Gedanken nicht ständig neu machen möchte ist es sinnvoll sie einmalig in einer Funktion zusammen zu fassen.
3. **Funktionen erleichtern Korrekturen.** Wenn Sie merken, dass Sie in der Implementierung einer Routine einen Fehler gemacht haben müssen Sie im besten Falle nur einmal die Definition der Funktion korrigieren - im schlimmsten Falle müssen sie in ihrem Code nach der Routine suchen und sie in jedem einzelnen Anwendungsfall erneut korrigieren.

Es gibt noch viele weitere Gründe dafür, Funktionen häufig zu verwenden. Viele hängen mit dem Entwicklerprinzip DRY ("Don't Repeat Yourself") zusammen.

3.4.2 Vektoren

Vektoren sind einer der wichtigsten Objektypen in R. Quasi alle Daten mit denen wir in R arbeiten werden als Vektoren behandelt.

Was Vektoren angeht gibt es wiederum die wichtige **Unterscheidung von atomaren Vektoren und Listen**. Beide bestehen ihrerseits aus Objekten und sie unterscheiden sich dadurch, dass atomare Vektoren nur aus Objekten des gleichen Typs bestehen können, Listen dagegen auch Objekte unterschiedlichen Typs beinhalten können.

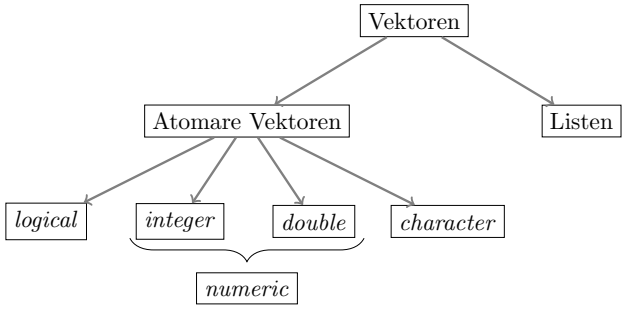
Entsprechend kann jeder atomare Vektor einem Typ zugeordnet werden, je nachdem welchen Typ seine Bestandteile haben. Hier sind insbesondere vier Typen relevant:

- `logical` (logische Werte): es gibt zwei logische Werte, `TRUE` und

`FALSE`, welche auch mit `T` oder `F` abgekürzt werden können
- `integer` (ganze Zahlen): das sollte im Prinzip selbsterklärend sein, allerding muss den ganzen Zahlen in R immer der Buchstabe `L` folgen, damit die Zahl tatsächlich als ganze Zahl interpretiert wird.[5] Beispiele sind `1L`, `400L` oder `10L`.
- `double` (Dezimalzahlen): auch das sollte selbsterklärend sein; Beispiele wären `1.5`, `0.0`, oder `-500.32`.
- Ganze Zahlen und Dezimalzahlen werden häufig unter der Kategorie `numeric` zusammengefasst. Dies ist in der Praxis aber quasi nie hilfreich und man sollte diese Kategorie möglichst nie verwenden.
- Wörter (`character`): sie sind dadurch gekennzeichnet, dass sie auch Buchstaben enthalten können und am Anfang und Ende ein `"` haben. Beispiele hier wären `"Hallo"`, `"500"` oder `"1_2_Drei"`.
- Es gibt noch zwei weitere besondere 'Typen', die strikt gesehen keine atomaren Vektoren darstellen, allerdings in diesem Kontext schon häufig auftauchen: `NULL`, was strikt genommen ein eigener Datentyp ist und immer die Länge 0 hat, sowie `NA`, das einen fehlenden Wert darstellt

[5] Diese auf den ersten Blick merkwürdige Syntax hat historische Gründe: als der integer Typ in die R Programmiersprache eingeführt wurde war er sehr stark an den Typ `long integer` in der Programmiersprache 'C' angelehnt. In C wurde ein solcher 'long integer' mit dem Suffix 'l' oder 'L' definiert, diese Regel wurde aus Kompatibilitätsgründen auch für R übernommen, jedoch nur mit 'L', da man Angst hatte, dass 'l' mit 'i' verwechselt wird, was in R für die imaginäre Komponente komplexer Zahlen verwendet wird.

Hieraus ergibt sich folgende Aufteilung für Vektoren:



Wir werden nun die einzelnen Typen genauer betrachten. Vorher wollen wir jedoch noch die Funktion `typeof` einführen. Sie hilft uns in der Praxis den Typ eines Objekts herauszufinden. Dafür rufen wir einfach die Funktion `typeof` mit dem zu untersuchenden Objekt oder dessen Namen auf:

```
typeof(2L)
```

```
#> [1] "integer"
```

```
x <- 22
typeof(x)
```

```
#> [1] "double"
```

Wir können auch explizit testen ob ein Objekt ein Objekt bestimmten Typs ist. Die generelle Syntax hierfür ist: `is.*()`, also z.B.:

```
x <- 1
is.integer(x)
```

```
#> [1] FALSE
```

```
is.double(x)
```

```
#> [1] TRUE
```

Diese Funktion gibt als Output also immer einen logischen Wert aus, je nachdem ob die Inputs des entsprechenden Typs sind oder nicht.

Bestimmte Objekte können in einen anderen Typ transformiert werden. Hier spricht man von `coercion` und die generelle Syntax hierfür ist: `as.*()`, also z.B.:

```
x <- "2"
print(typeof(x))
```

```
#> [1] "character"
```

```
x <- as.double(x)
print(typeof(x))
```

```
#> [1] "double"
```

Allerdings ist eine Transformation nicht immer möglicht:

```
as.double("Hallo")
```

```
#> Warning: NAs introduced by coercion
```

```
#> [1] NA
```

Da R nicht weiß wie man aus dem Wort 'Hallo' eine Dezimalzahl machen soll, transformiert er das Wort in einen 'Fehlenden Wert', der in R als `NA` bekannt ist und unten noch genauer diskutiert wird.

Für die Grundtypen ergibt sich folgende logische Hierachie an trivialen Transformationen: `logical` → `integer` → `double` → `character`, d.h. man kann eine Dezimalzahl ohne Probleme in ein Wort transformieren, aber nicht umgekehrt:

```
x <- 2
y <- as.character(x)
print(y)
```

```
#> [1] "2"
```

```
z <- as.double(y)  # Das funktioniert
print(z)
```

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

```
k <- as.double("Hallo")  # Das nicht
```

```
#> Warning: NAs introduced by coercion
```

```
print(k)
```

```
#> [1] NA
```

Da nicht immer ganz klar ist wann R bei Transformationen entgegen der gerade eingeführten Hierachie eine Warnung ausgibt und wann nicht sollte man hier immer besondere Vorsicht walten lassen!

Zudem ist bei jeder Transformation Vorsicht geboten, da sie häufig Eigenschaften der Objekte implizit verändert. So führt eine Transformation von einer Dezimalzahl hin zu einer ganzen Zahl teils zu unerwartetem Rundungsverhalten:

```
x <- 1.99
as.integer(x)
```

```
#> [1] 1
```

Auch führen Transformationen, die der eben genannten Hierachie zuwiderlaufen, nicht zwangsweise zu Fehlern, sondern 'lediglich' zu unerwarteten Änderungen, die in jedem Fall vermieden werden sollten:

```
z <- as.logical(99)
print(z)
```

```
#> [1] TRUE
```

Häufig transformieren Funktionen ihre Argumente automatisch, was meistens hilfreich ist, manchmal aber auch gefährlich sein kann:

```
x <- 1L  # Integer
y <- 2  # Double
z <- x + y
typeof(z)
```

```
#> [1] "double"
```

Interessanterweise werden logische Werte ebenfalls transformiert:

```
x <- TRUE
y <- FALSE
z <- x + y  # TRUE wird zu 1, FALSE zu 0
print(z)
```

```
#> [1] 1
```

Daher sollte man immer den Überblick behalten, mit welchen Objekttypen man gerade arbeitet.

Hier noch ein kurzer Überblick zu den Test- und Transformationsbefehlen:

Typ	Test	Transformation
logical	`is.logical`	`as.logical`
double	`is.double`	`as.double`
integer	`is.integer`	`as.integer`
character	`is.character`	`as.character`
function	`is.function`	`as.function`
NA	`is.na`	NA
NULL	`is.null`	`as.null`

Ein letzter Hinweis zu **Skalaren**. Unter Skalaren verstehen wir in der Regel 'einzelne Zahlen', z.B. `2`. Dieses Konzept gibt es in R nicht. `2` ist ein Vektor der Länge 1. Wir unterscheiden also vom Typ her nicht zwischen einem Vektor, der nur ein oder mehrere Elemente hat.

Hinweis: Um längere Vektoren zu erstellen, verwenden wir die Funktion `c()`:

```
x <- c(1, 2, 3)
x
```

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

Dabei können auch Vektoren miteinander verbunden werden:

```
x <- 1:3  # Shortcut für: x <- c(1, 2, 3)
y <- 4:6
z <- c(x, y)
z
```

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

Da atomare Vektoren immer nur Objekte des gleichen Typs enthalten können, könnte man erwarten, dass es zu einem Fehler kommt, wenn wir Objete unterschiedlichen Type kombinieren wollen:

```
x <- c(1, "Hallo")
```

Tatsächlich transformiert R die Objekte allerdings nach der oben beschriebenen Hierachie `logical` → `integer` → `double` → `character`. Da hier keine Warnung oder kein Fehler ausgegeben wird, sind derlei Transformationen eine gefährliche Fehlerquelle!

Hinweis: Die Länge eines Vektors kann mit der Funktion `length` bestimmt werden:

```
x = c(1, 2, 3)
len_x <- length(x)
len_x
```

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

3.4.3 Logische Werte (logical)

Die logischen Werte `TRUE` und `FALSE` sind häufig das Ergebnis von logischen Abfragen, z.B. 'Ist 2 größer als 1?'. Solche Abfragen kommen in der Forschungspraxis häufig vor und es macht Sinn, sich mit den häufigsten logischen Operatoren vertraut zu machen:

Operator	Funktion in R	Beispiel
größer	`>`	`2>1`
kleiner	`<`	`2<4`
gleich	`==`	`4==3`
größer gleich	`>=`	`8>=8`
kleiner gleich	`<=`	`5<=9`
nicht gleich	`!=`	`4!=5`
und	`&`	`x<90 & x>55`
oder	`|`	`x<90 | x>55`
entweder oder	`xor()`	`xor(2<1, 2>1)`
nicht	`!`	`!(x==2)`
ist wahr	`isTRUE()`	`isTRUE(1>2)`

Das Ergebnis eines solches Tests ist immer ein logischer Wert:

```
x <- 4
y <- x == 8
typeof(y)
```

```
#> [1] "logical"
```

Es können auch längere Vektoren getestet werden:

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

```
#> [1]  TRUE FALSE FALSE
```

Tests können beliebig miteinander verknüpft werden:

```
x <- 1L
x > 2 | x < 2 & (is.double(x) & x != 0)
```

```
#> [1] FALSE
```

Da für viele mathematischen Operationen `TRUE` als die Zahl `1` interpretiert wird, ist es einfach zu testen wie häufig eine bestimmte Bedingung erfüllt ist:

```
x <- 1:50
smaller_20 <- x < 20
print(sum(smaller_20)  # Wie viele Elemente sind kleiner als 20?
)
```

```
#> [1] 19
```

```
print(sum(smaller_20/length(x))  # Wie hoch ist der Anteil von diesen Elementen?
)
```

```
#> [1] 0.38
```

3.4.4 Wörter (character)

Wörter werden in R dadurch gebildet, dass an ihrem Anfang und Ende das Symbol `'` oder `""` steht:

```
x <- "Hallo"
typeof(x)
```

```
#> [1] "character"
```

```
y <- "Auf Wiedersehen"
typeof(y)
```

```
#> [1] "character"
```

Wie andere Vektoren können sie mit der Funktion `c()` verbunden werden:

```
z <- c(x, " und ", y)
z
```

```
#> [1] "Hallo"           " und "
#> [3] "Auf Wiedersehen"
```

Nützlich ist in diesem Zusammenhang die Funktion `paste()`, die Elemente von mehreren Vektoren in Wörter transformiert und verbindet:

```
x <- 1:10
y <- paste("Versuch Nr.", x)
y
```

```
#>  [1] "Versuch Nr. 1"  "Versuch Nr. 2"
#>  [3] "Versuch Nr. 3"  "Versuch Nr. 4"
#>  [5] "Versuch Nr. 5"  "Versuch Nr. 6"
#>  [7] "Versuch Nr. 7"  "Versuch Nr. 8"
#>  [9] "Versuch Nr. 9"  "Versuch Nr. 10"
```

`paste()` akzeptiert ein optionales Argument `sep`, mit dem wir den Wert angeben können, der zwischen die zu verbindenden Elemente gesetzt wird:

```
tag_nr <- 1:10
x_axis <- paste("Tag", tag_nr, sep = ": ")
x_axis
```

```
#>  [1] "Tag: 1"  "Tag: 2"  "Tag: 3"  "Tag: 4"
#>  [5] "Tag: 5"  "Tag: 6"  "Tag: 7"  "Tag: 8"
#>  [9] "Tag: 9"  "Tag: 10"
```

> Hinweis: Hier haben wir ein Beispiel für das so genannte 'Recycling' gesehen: da der Vektor `c("Tag")` kürzer war als der Vektor `tag_nr` wird `c("Tag")` einfach kopiert damit die Operation mit `paste()` Sinn ergibt. Recycling ist oft praktisch, aber manchmal auch schädlich, nämlich dann, wenn man eigentlich davon ausgeht eine Operation mit zwei gleich langen Vektoren durchzuführen, dies aber tatsächlich nicht tut. In einem solchen Fall führt Recycling dazu, dass keine Fehlermeldung ausgegeben wird. Ein Beispiel dafür gibt folgender Code, in dem die Intention klar die Verbindung aller Wochentage zu Zahlen ist und einfach ein Wochentag vergessen wurde:

```
tage <- paste("Tag ", 1:7, ":", sep = "")
tag_namen <- c("Montag", "Dienstag", "Mittwoch",
    "Donnerstag", "Freitag", "Samstag")
paste(tage, tag_namen)
```

```
#> [1] "Tag 1: Montag"     "Tag 2: Dienstag"
#> [3] "Tag 3: Mittwoch"   "Tag 4: Donnerstag"
#> [5] "Tag 5: Freitag"    "Tag 6: Samstag"
#> [7] "Tag 7: Montag"
```

3.4.5 Fehlende Werte und NULL

Fehlende Werte werden in R als `NA` kodiert. `NA` erfüllt gerade in statistischen Anwendungen eine wichtige Rolle, da ein bestimmter Platz in einem Vektor aktuell fehlend sein müsste, aber als Platz dennoch existieren muss.

> **Beispiel:** Der Vektor `x` enthält einen logischen Wert, der zeigt ob eine Person die Fragen auf einem Fragebogen richtig beantwortet hat.

> Wenn die Person die dritte Frage auf dem Fragebogen nicht beantwortet hat, sollte dies durch `NA` kenntlich gemacht werden. Einfach den Wert komplett wegzulassen macht es im Nachhinein unmöglich festzustellen *welche* Frage die Person nicht beantwortet hat.

Die meisten Operationen die `NA` als einen Input bekommen geben auch als Output `NA` aus, weil unklar ist wie die Operation mit unterschiedlichen Werten für den fehlenden Wert ausgehen würde:

```
5 + NA
```

```
#> [1] NA
```

Einzige Ausnahmen sind Operationen, die unabhängig vom fehlenden Wert einen bestimmten Wert annehmen:

```
NA | TRUE  # Gibt immer TRUE, unabhängig vom Wert für NA
```

```
#> [1] TRUE
```

Um zu testen ob ein Vektor `x` fehlende Werte enthält sollte die Funktion `is.na` verwendet werden, und nicht etwa der Ausdruck `x==NA`:

```
x <- c(NA, 5, NA, 10)
print(x == NA)  # Unklar, da man nicht weiß, ob alle NA für den gleichen Wert stehen
```

```
#> [1] NA NA NA NA
```

```
print(is.na(x))
```

```
#> [1]  TRUE FALSE  TRUE FALSE
```

Wenn eine Operation einen nicht zu definierenden Wert ausgibt, ist das Ergebnis nicht `NA` sondern `NaN` (*not a number*):

```
0/0
```

```
#> [1] NaN
```

Eine weitere Besonderheit ist `NULL`, welches in der Regel als Vektor der Länge 0 gilt, aber häufig zu besonderen Zwecken verwendet wird:

```
x <- NULL
length(x)
```

```
#> [1] 0
```

3.4.6 Indizierung und Ersetzung

Einzelne Elemente von atomare Vektoren können mit eckigen Klammern extrahiert werden:

```
x <- c(2, 4, 6)
x[1]
```

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

Auf diese Weise können auch bestimmte Elemente modifiziert werden:

```
x <- c(2, 4, 6)
x[2] <- 99
x
```

```
#> [1]  2 99  6
```

Es kann auch mehr als ein Element extrahiert werden:

```
x[1:2]
```

```
#> [1]  2 99
```

Negative Indizes sind auch möglich, diese eliminieren die entsprechenden Elemente:

```
x[-1]
```

```
#> [1] 99  6
```

Um das letzte Element eines Vektors zu bekommen verwendet man einen Umweg über die Funktion `length()`:

```
x[length(x)]
```

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

3.4.7 Nützliche Funktionen für atomare Vektoren

Hier sollen nur einige Funktionen erwähnt werden, die im Kontext von atomaren Vektoren besonders praktisch sind,[6] inbesondere wenn es darum geht solche Vektoren herzustellen, bzw. Rechenoperationen mit ihnen durchzuführen.

[6] Für viele typische Aufgaben gibt es in R bereits eine vordefinierte Funktion. Am einfachsten findet man diese durch googlen.

Herstellung von atomaren Vektoren:

Eine Sequenz ganzer Zahlen wird in der Regel sehr häufig gebraucht. Entsprechend gibt es den hilfreichen Shortcut `:`

```
x <- 1:10
x
```

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

```
y <- 10:1
y
```

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

Häufig möchten wir jedoch eine kompliziertere Sequenz bauen. In dem Fall hilft uns die allgemeinere Funktion `seq()`:

```
x <- seq(1, 10)
print(x)
```

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

In diesem Fall ist `seq()` äquivalent zu `:`. `seq` erlaubt aber mehrere optionale Argumente: so können wir mit `by` die Schrittlänge zwischen den einzelnen Zahlen definieren.

```
y <- seq(1, 10, by = 0.5)
print(y)
```

```
#>  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5
#>  [9]  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5
#> [17]  9.0  9.5 10.0
```

Wenn wir die Länge des resultierenden Vektors festlegen wollen und die Schrittlänge von R automatisch festgelegt werden soll, können wir dies mit dem Argument `length.out` machen:

```
z <- seq(2, 8, length.out = 4)
print(z)
```

```
#> [1] 2 4 6 8
```

Und wenn wir einen Vektor in der Länge eines anderen Vektors erstellen wollen, bietet sich das Argument `along.with` an. Dies wird häufig für das Erstellen von Indexvektoren verwendet. In einem solchen Fall müssen wir die Indexzahlen nicht direkt angeben:

```
z_index <- seq(along.with = z)
print(z_index)
```

```
#> [1] 1 2 3 4
```

Auch häufig möchten wir einen bestimmten Wert wiederholen. Das geht mit der Funktion `rep`:

```
x <- rep(NA, 5)
print(x)
```

```
#> [1] NA NA NA NA NA
```

Rechenoperationen

Es gibt eine Reihe von Operationen, die wir sehr häufig gemeinsam mit Vektoren anwenden. Häufig interessiert und die **Länge** eines Vektors. Dafür können wir die Funktion `length()` verwenden:

```
x <- c(1, 2, 3, 4)
length(x)
```

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

Wenn wir den **größten** oder **kleinsten Wert** eines Vektors erfahren möchten geht das mit den Funktionen `min()` und `max()`:

```
min(x)
```

```
#> [1] 1
```

```
max(x)
```

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

Beide Funktionen besitzen ein optionales Argument `na.rm`, das entweder `TRUE` oder `FALSE` sein kann. Im Fallse von `TRUE` werden alle `NA` Werte für die Rechenoperation entfernt:

```
y <- c(1, 2, 3, 4, NA)
min(y)
```

```
#> [1] NA
```

```
min(y, na.rm = TRUE)
```

```
#> [1] 1
```

Den **Mittelwert** bzw die **Varianz/Standardabweichung** der Elemente bekommen wir mit `mean()`, `var()`, bzw. `sd()`, wobei alle Funktionen auch das optionale Argument `na.rm` akzeptieren:

```
mean(x)
```

```
#> [1] 2.5
```

```
var(y)
```

```
#> [1] NA
```

```
var(y, na.rm = T)
```

```
#> [1] 1.666667
```

Ebenfalls häufig sind wir an der **Summe**, bzw, dem **Produkt** aller Elemente des Vektors interessiert. `sum()` und `prod()` helfen weiter und auch sie kennen das optionale Argument `na.rm`:

```
sum(x)
```

```
#> [1] 10
```

```
prod(y, na.rm = T)
```

```
#> [1] 24
```

3.4.8 Listen

Im Gegensatz zu atomaren Vektoren können Listen Objekte verschiedenen Typs enthalten. Sie werden mit der Funktion `list()` erstellt:

```
l_1 <- list("a", c(1, 2, 3), FALSE)
typeof(l_1)
```

```
#> [1] "list"
```

```
l_1
```

```
#> [[1]]
#> [1] "a"
#>
#> [[2]]
#> [1] 1 2 3
#>
#> [[3]]
#> [1] FALSE
```

Wir können Listen mit der Funktion `str()` inspizieren. In diesem Fall erhalten wir unmittelbar Informationen über die Art der Elemente:

```
str(l_1)
```

```
#> List of 3
#>  $ : chr "a"
#>  $ : num [1:3] 1 2 3
#>  $ : logi FALSE
```

Die einzelnen Elemente einer Liste können auch benannt werden:

```
l_2 <- list(erstes_element = "a", zweites_element = c(1,
    2, 3), drittes_element = FALSE)
```

Die Namen aller Elemente in der Liste erhalten wir mit der Funktion `names()`:

```
names(l_2)
```

```
#> [1] "erstes_element"  "zweites_element"
#> [3] "drittes_element"
```

Um einzelne Elemente einer Liste auszulesen müssen wir `[[` anstatt `[` verwemden. Wir können dann entweder Elemente nach ihrer Position oder ihren Namen auswählen:

```
l_2[[1]]
```

```
#> [1] "a"
```

```
l_2[["erstes_element"]]
```

```
#> [1] "a"
```

Im folgenden wollen wir uns noch mit zwei speziellen Typen beschäftigen, die weniger fundamental als die bislang diskutierten sind, jedoch häufig in der alltäglichen Arbeit vorkommen: Matrizen und Data Frames.

3.4.9 Matrizen

Bei Matrizen handelt es sich um zweidimensionale Objekte mit Zeilen und Spalten, bei denen es sich jeweils um atomare Vektoren handelt.

Erstellen von Matrizen

Matrizen werden mit der Funktion `matrix()`erstellt. Diese Funktion nimmt als erstes Argument die Elemente der Matrix und dann die Spezifikation der Anzahl von Zeilen (`nrow`) und/oder der Anzahl von Spalten (`ncol`):

```
m_1 <- matrix(11:20, nrow = 5)
m_1
```

```
#>      [,1] [,2]
#> [1,]   11   16
#> [2,]   12   17
#> [3,]   13   18
#> [4,]   14   19
#> [5,]   15   20
```

Wie können die Zeilen, Spalten und einzelne Werte folgendermaßen extrahieren und ggf. Ersetzungen vornehmen:

```
m_1[, 1]  # Erste Spalte
```

```
#> [1] 11 12 13 14 15
```

```
m_1[1, ]  # Erste Zeile
```

```
#> [1] 11 16
```

```
m_1[2, 2]  # Element [2,2]
```

```
#> [1] 17
```

Optionaler Hinweis: Matrizen sind weniger 'fundamantal' als atomare Vektoren. Entsprechend gibt uns `typeof()` für eine Matrix auch den Typ der enthaltenen atomaren Vektoren an:

```
typeof(m_1)
```

```
#> [1] "integer"
```

Um zu testen ob es sich bei einem Objekt um eine Matrix handelt verwenden wir entsprechend `is.matrix()`:

```
is.matrix(m_1)
```

```
#> [1] TRUE
```

```
is.matrix(2)
```

```
#> [1] FALSE
```

Matrizenalgebra

Matrizenalgebra spielt in vielen statistischen Anwendungen eine wichtige Rolle. In R ist es sehr einfach die typischen Rechenoperationen für Matrizen zu implementieren. Hier nur ein paar Beispiele, für die wir die folgenden Matrizen verwenden:

$$A = \begin{pmatrix} 1 & 6 \\ 5 & 3 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 2 \\ 4 & 8 \end{pmatrix}$$

```
matrix_a <- matrix(c(1, 5, 6, 3), ncol = 2)
matrix_b <- matrix(c(0, 4, 2, 8), ncol = 2)
```

Skalar-Addition:

$$4 + \boldsymbol{A} = \begin{pmatrix} 4 + a_{11} & 4 + a_{21} \\ 4 + a_{12} & 4 + a_{22} \end{pmatrix}$$

```
4 + matrix_a
```

```
#>      [,1] [,2]
#> [1,]    5   10
#> [2,]    9    7
```

Matrizen-Addition:

$$\boldsymbol{A} + \boldsymbol{B} = \begin{pmatrix} a_{11} + b_{11} & a_{21} + b_{21} \\ a_{12} + b_{12} & a_{22} + b_{22} \end{pmatrix}$$

```
matrix_a + matrix_b
```

```
#>      [,1] [,2]
#> [1,]    1    8
#> [2,]    9   11
```

Skalar-Multiplikation:

$$2 \cdot \boldsymbol{A} = \begin{pmatrix} 2 \cdot a_{11} & 2 \cdot a_{21} \\ 2 \cdot a_{12} & 2 \cdot a_{22} \end{pmatrix}$$

```
2 * matrix_a
```

```
#>      [,1] [,2]
#> [1,]    2   12
#> [2,]   10    6
```

Elementenweise Matrix Multiplikation (auch 'Hadamard-Produkt'):

$$\boldsymbol{A} \odot \boldsymbol{B} = \begin{pmatrix} a_{11} \cdot b_{11} & a_{21} \cdot b_{21} \\ a_{12} \cdot b_{12} & a_{22} \cdot b_{22} \end{pmatrix}$$

```
matrix_a * matrix_b
```

```
#>      [,1] [,2]
#> [1,]    0   12
#> [2,]   20   24
```

Matrizen-Multiplikation:

$$\boldsymbol{A} \cdot \boldsymbol{B} = \begin{pmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} & a_{11} \cdot b_{21} + a_{12} \cdot b_{22} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} \end{pmatrix}$$

```
matrix_a %*% matrix_b
```

```
#>      [,1] [,2]
#> [1,]   24   50
#> [2,]   12   34
```

Die Inverse einer Matrix $\boldsymbol{A}$, $\boldsymbol{A}^{-1}$, ist definiert sodass gilt

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}$$

Sie kann in R mit der Funktion `solve()` identifiziert werden:

```
solve(matrix_a)
```

```
#>             [,1]        [,2]
#> [1,] -0.1111111  0.22222222
#> [2,]  0.1851852 -0.03703704
```

```
matrix_a %*% solve(matrix_a)
```

```
#>      [,1]         [,2]
#> [1,]    1 2.775558e-17
#> [2,]    0 1.000000e+00
```

Die minimalen Abweichungen sind auf machinelle Rundungsfehler zurückzuführen und treten häufig auf.

Es gibt im Internet zahlreiche gute Überblicksartikel zum Thema Matrizenalgebra in R, z.B. hier oder in größerem Umfang hier.

3.4.10 Data Frames

Der `data.frame` ist eine besondere Art von Liste und ist ein in der Datenanalyse regelmäßig auftretender Datentyp. Gegensatz zu einer normalen Liste müssen bei einem `data.frame` alle Elemente die gleiche Länge aufweisen. Das heißt man kann sich einen `data.frame` als eine rechteckige Liste vorstellen.

Wegen der engen Verwandschaft können wir einen `data.frame` direkt aus einer Liste erstellen indem wir die Funktion `as.data.frame()` verwenden:

```
l_3 <- list(a = 1:3, b = 4:6, c = 7:9)
df_3 <- as.data.frame(l_3)
```

Wenn wir R nach dem Typ von `df_3` fragen, sehen wir, dass es sich weiterhin um eine Liste handelt:

```
typeof(df_3)
```

```
#> [1] "list"
```

Allerdings können wir testen ob `df_3` ein `data.frame` ist indem wir `is.data.frame` benutzen:

```
is.data.frame(df_3)
```

```
#> [1] TRUE
```

```
is.data.frame(l_3)
```

```
#> [1] FALSE
```

Wenn wir `df_3` ausgeben sehen wir unmittelbar den Unterschied zu klassischen Liste:[7]

[7] Gerade bei sehr großen Data Frames möchte man oft nur die ersten paar Elemente inspizieren. Das ist mit der Funktion `head()` möglich.

```
l_3
```

```
#> $a
#> [1] 1 2 3
#>
#> $b
#> [1] 4 5 6
#>
#> $c
#> [1] 7 8 9
```

```
df_3
```

```
#>   a b c
#> 1 1 4 7
#> 2 2 5 8
#> 3 3 6 9
```

Die andere Möglichkeit einen `data.frame` zu erstellen ist direkt über die Funktion `data.frame()`, wobei es hier in der Regel ratsam ist das optionale Argument `stringsAsFactors` auf `FALSE` zu setzen, da sonst Wörter in so genannte Faktoren umgewandelt werden:[8]

[8] Zur Geschichte dieses wirklich ärgerlichen Verhaltens siehe diesen Blog.

```
df_4 <- data.frame(gender = c(rep("male", 3),
    rep("female", 2)), height = c(89, 75, 80,
    66, 50), stringsAsFactors = FALSE)
df_4
```

```
#>   gender height
#> 1   male     89
#> 2   male     75
#> 3   male     80
#> 4 female     66
#> 5 female     50
```

Data Frames sind das klassische Objekt um eingelesene Daten zu repräsentieren. Wenn Sie sich z.B. Daten zum BIP in Deutschland aus dem Internet runterladen und diese Daten dann in R einlesen, werden diese Daten zunächst einmal als `data.frame` repräsentiert.[9] Diese Repräsentation erlaubt dann eine einfache Analyse und Manipulation der Daten.

[9] Das ist nicht ganz korrekt, weil es mittlerweilse Erweiterungen gibt, welche den `data.frame` mit effizienteren Objekten ersetzen, z.B. dem `tibble` oder dem `data.table`. Der Umgang mit diesen Objekten ist jedoch sehr ähnlich zum `data.frame`.

Zwar gibt es eine eigene Vorlesung zur Bearbeitung von Daten, wir wollen aber schon hier einige zentrale Befehle im Zusammenhang von Data Frames einführen.

An dieser Stelle sei jedoch schon angemerkt, dass um Zeilen, Spalten oder einzelne Elemente auszuwählen verwenden die gleichen Befehle wie bei Matrizen verwendet werdenkönnen:

```
df_4[, 1]  # erste Spalte
```

```
#> [1] "male"   "male"   "male"   "female"
#> [5] "female"
```

```
df_4[, 2]  # Werte der zweiten Spalte
```

```
#> [1] 89 75 80 66 50
```

Die Abfrage funktioniert nicht nur mit Indices, sondern auch mit Spaltennamen:[10]

[10] Anstelle von `[[` kann auch der Shortcut `$` verwendet werden. Das werden wir aufgrund der größeren Transparenz von `[[` hier jedoch nicht verwenden.

```
df_4[["gender"]]
```

```
#> [1] "male"   "male"   "male"   "female"
#> [5] "female"
```

Wenn wir `[` anstatt von `[[` verwenden erhalten wir als Output einen (reduzierten) Data Frame:

```
df_4["gender"]
```

```
#>   gender
#> 1   male
#> 2   male
#> 3   male
#> 4 female
#> 5 female
```

Es können auch mehrere Zeilen ausgewählt werden:

```
df_4[1:2, ]  # Die ersten beiden Zeilen
```

```
#>   gender height
#> 1   male     89
#> 2   male     75
```

Oder einzelne Werte:

```
df_4[2, 2]  # Zweiter Wert der zweiten Spalte
```

```
#> [1] 75
```

Dies können wir uns zu Nutze machen um den Typ der einzelnen Spalten herauszufinden:

```
typeof(df_4[["gender"]])
```

```
#> [1] "character"
```

3.5 Pakete

Bei Paketen handelt es sich um eine Kombination aus R Code, Daten, Dokumentationen und Tests. Sie sind der beste Weg, reproduzierbaren Code zu erstellen und frei zugänglich zu machen. Zwar werden Pakete häufig der Öffentlichkeit zugänglich gemacht, z.B. über GitHub oder CRAN. Es ist aber genauso hilfreich, Pakete für den privaten Gebrauch zu schreiben, z.B. um für bestimmte Routinen Funktionen zu programmieren, zu dokumentieren und in verschiedenen Projekten verfügbar zu machen.[11]

[11] Wickham and Bryan (2019) bietet eine exzellente Einführung in das Programmieren von R Paketen.

Die Tatsache, dass viele Menschen statistische Probleme lösen indem sie bestimmte Routinen entwickeln, diese dann generalisieren und über Pakete der ganzen R Community frei verfügbar machen, ist einer der Hauptgründe für den Erfolg und die breite Anwendbarkeit von R.

Wenn man R startet haben wir Zugriff auf eine gewisse Anzahl von Funktionen, vordefinierten Variablen und Datensätzen. Die Gesamtheit dieser Objekte wird in der Regel `base R` genannt, weil wir alle Funktionalitäten ohne Weiteres nutzen können.

Die Funktion `assign`, zum Beispiel, ist Teil von `base R`: wir starten R und können Sie ohne Weiteres verwenden.

Im Prinzip kann so gut wie jedwede statistische Prozedur in `base R` implementiert werden. Dies ist aber häufig zeitaufwendig und fehleranfällig: wie wir am Beispiel von Funktionen gelernt haben, sollten häufig verwendete Routinen im Rahmen von einer Funktion implementiert werden, die dann immer wieder angewendet werden kann. Das reduziert nicht nur Fehler, sondern macht den Code besser verständlich.

Pakete folgen dem gleichen Prinzip, nur tragen sie die Idee noch weiter: hier wollen wir die Funktionen auch über ein einzelnes R Projekt hinaus nutzbar machen, sodass sie nicht in jedem Projekt neu definiert werden müssen, sondern zentral nutzbar gemacht und dokumentiert werden.

Um ein Paket in R zu nutzen, muss es zunächst installiert werden. Für Pakete, die auf der zentralen R Pakete Plattform CRAN verfügbar sind, geht dies mit der Funktion `install.packages`. Wenn wir z.B. das Paket `data.table` installieren wollen geht das mit dem folgenden Befehl:

```
install.packages("data.table")
```

Das Paket `data.table` enthält viele Objekte, welche die Arbeit mit großen Datensätzen enorm erleichtern. Darunter ist eine verbesserte Version des `data.frame`, der `data.table`. Wir können einen `data.frame` mit Hilfe der Funktion `as.data.table()` in einen `data.table` umwandeln.

Allerdings haben wir selbst nach erfolgreicher Installation von `data.table` nicht direkt Zugriff auf diese Funktion:

```
x <- data.frame(a = 1:5, b = 21:25)
as.data.table(x)
```

```
#> Error in as.data.table(x): could not find function "as.data.table"
```

Wir haben zwei Möglichkeiten auf die Objekte im Paket `data.table` zuzugreifen: zum einen können wir mit dem Operator `::` arbeiten:

```
y <- data.table::as.data.table(x)
y
```

```
#>    a  b
#> 1: 1 21
#> 2: 2 22
#> 3: 3 23
#> 4: 4 24
#> 5: 5 25
```

Wir schreiben also den Namen des Pakets, direkt gefolgt von `::` und dann den Namen des Objets aus dem Paket, das wir vewendent wollen.

Zwar ist das der transparenteste und sauberste Weg auf Objekte aus anderen Paketen zuzugreifen, allerdings kann es auch nervig sein wenn man häufig oder sehr viele Objekte aus dem gleichen Paket verwendet. Wir können alle Objekte eines Paketes direkt zugänglich machen indem wir die Funktion `library()` verwenden.

```
library(data.table)
y <- as.data.table(x)
```

Der Übersicht halber sollte das für alle in einem Skript verwendeten Pakete ganz am Anfang des Skripts gemacht werden. So sieht man auch unmittelbar welche Pakete für das Skript installiert sein müssen.

Grundsätzlich sollte man in jedem Skript nur die Pakete mit `library()` einlesen, die auch tatsächlich verwendet werden. Ansonsten lädt man unnötigerweise viele Objekte und verliert den Überblick woher eine bestimmte Funktion eigentlich kommt. Außerdem ist es schwieriger für andere das Skript zu verwenden, weil unter Umständen viele Pakete unnötigerweise installiert werden müssen.

Da Pakete dezentral von verschiedensten Menschen hergestellt werden, besteht die Gefahr, dass Objekte in unterschiedlichen Paketen den gleichen Namen bekommen. Da in R ein Name nur zu einem Objekt gehören kann, werden beim Einladen mehrerer Pakete eventuell Namen überschrieben, oder ‘maskiert’. Dies wird am Anfang beim

Einlesen der Pakete mitgeteilt, gerät aber leicht in Vergessenheit und kann zu sehr kryptischen Fehlermeldungen führen.

Wir wollen das kurz anhand der beiden Pakete `dplyr` und `plm` illustrieren:

```
library(dplyr)
```

```
library(plm)
```

```
#>
#> Attaching package: 'plm'
```

```
#> The following objects are masked from 'package:dplyr':
#>
#>     between, lag, lead
```

```
#> The following object is masked from 'package:data.table':
#>
#>     between
```

In beiden Paketen gibt es Objekte mit den Namen `between`, `lag` und `lead`. Bei der Verwendung von `library` maskiert das später eingelesene Paket die Objekte des früheren. Wir können das illustrieren indem wir den Namen des Objekts eingeben:

```
lead
```

```
#> function (x, k = 1, ...)
#> {
#>     UseMethod("lead")
#> }
#> <bytecode: 0x7fe005e5c9a8>
#> <environment: namespace:plm>
```

Aus der letzten Zeile wird ersichtlich, dass `lead` hier aus dem Paket `plm` kommt.

Wenn wir die Funktion aus `dplyr` verwenden wollen, müssen wir `::` verwenden:

```
dplyr::lead
```

```
#> function (x, n = 1L, default = NA, order_by = NULL, ...)
#> {
#>     if (!is.null(order_by)) {
#>         return(with_order(order_by, lead, x, n = n, default = default))
#>     }
#>     if (length(n) != 1 || !is.numeric(n) || n < 0) {
#>         bad_args("n", "must be a nonnegative integer scalar, ",
```

```
#>             "not {friendly_type_of(n)} of length {length(n)}")
#>     }
#>     if (n == 0)
#>         return(x)
#>     xlen <- length(x)
#>     n <- pmin(n, xlen)
#>     out <- c(x[-seq_len(n)], rep(default, n))
#>     attributes(out) <- attributes(x)
#>     out
#> }
#> <bytecode: 0x7fe0059b7dc8>
#> <environment: namespace:dplyr>
```

Wenn es zu Maskierungen kommt ist es aber der Transparenz wegen besser in beiden Fällen `::` zu verwenden, also `plm::lead` und `dplyr::lead`.

Hinweis: Alle von Konflikten betroffenen Objekte können mit der Funktion `conflicts()` angezeigt werden.

Optionale Info: Um zu überprüfen in welcher Reihenfolge R nach Objekten sucht, kann die Funktion `search` verwendet werden. Wenn ein Objekt aufgerufen wird schaut R zuerst im ersten Element des Vektors nach, der globalen Umgebung. Wenn das Objekt dort nicht gefunden wird, schaut es im zweiten, etc. Wie man hier auch erkennen kann, werden einige Pakete standardmäßig eingelesen. Wenn ein Objekt nirgends gefunden wird gibt R einen Fehler aus. Im vorliegenden Falle zeigt uns die Funktion, dass er erst im Paket `plm` nach der Funktion `lead()` sucht, und nicht im Paket `dplyr`:

```
search()
```

```
#>  [1] ".GlobalEnv"
#>  [2] "package:plm"
#>  [3] "package:dplyr"
#>  [4] "package:data.table"
#>  [5] "package:stats"
#>  [6] "package:graphics"
#>  [7] "package:grDevices"
#>  [8] "package:utils"
#>  [9] "package:datasets"
#> [10] "package:methods"
#> [11] "Autoloads"
#> [12] "package:base"
```

Weiterführender Hinweis Um das Maskieren besser zu verstehen sollte man sich mit dem Konzept von *namespaces* und *environments* auseinandersetzen. Eine gute Erklärung bietet Wickham and Bryan (2019).

Weiterführender Hinweis Das Paket `conflicted` führt dazu, dass R immer einen fehler ausgibt wenn nicht eindeutige Objektnamen verwendet werden.

3.6 Kurzer Exkurs zum Einlesen und Schreiben von Daten

Zum Abschluss wollen wir noch kurz einige Befehle zum Einlesen von Daten einführen. Später werden wir uns ein ganzes Kapitel mit dem Einlesen und Schreiben von Daten beschäftigen, da dies in der Regel einen nicht unbeträchtlichen Teil der quantitativen Forschungsarbeit in Anspruch nimmt. An dieser Stelle wollen wir aber nur lernen, wie man einen Datensatz in R einliest.

R kann zahlreiche verschiedene Dateiformate einlesen, z.B. `csv`, `dta` oder `txt`, auch wenn für manche Formate bestimmte Pakete geladen sein müssen.

Das gerade für kleinere Datensätze mit Abstand beste Format ist in der Regel `csv`, da es von zahlreichen Programmen und auf allen Betriebssystemen gelesen und geschrieben werden kann.

Für die Beispiele hier nehmen wir folgende Ordnerstruktur an:

```
2019-Methoden
    2019-Methoden.Rproj

+---data

    +---raw
    |       Rohdaten.csv
|   |
|   +---tidy
```

Um die Daten einzulesen verwenden wir das Paket `tidyverse`, die wir später genauer kennen lernen werden. Sie enthält viele nützliche Funktionen zur Arbeit mit Datensätzen. Zudem verwende ich das Paket `here` um relative Pfade immer von meinem Arbeitsverzeichnis aus angeben zu können.[12]

[12] Das ist notwendig, da dieses Skript in R Markdown geschrieben ist und das Arbeitsverzeichnis automatisch auf den Ordner ändert, in dem das .Rmd file liegt. Mehr Information zum Schreiben von R Markdown finden Sie im Anhang. Dieser wird auch in der Vorlesung besprochen.

```
library(tidyverse)
library(here)
```

Nehmen wir an, die Datei `Rohdaten.csv` sähe folgendermaßen aus:

```
Auto,Verbrauch,Zylinder,PS
Ford Pantera L,15.8,8,264
Ferrari Dino,19.7,6,175
Maserati Bora,15,8,335
Volvo 142E,21.4,4,109
```

Wie in einer typischen csv Datei sind die Spalten hier mit einem Komma getrennt. Um diese Datei einzulesen verwenden wir die Funktion `read_csv` mit dem Dateipfad als erstes Argument:

```
auto_daten <- read_csv(here("data/raw/Rohdaten.csv"))
auto_daten
```

```
#> # A tibble: 4 x 4
#>   Auto            Verbrauch Zylinder    PS
#>   <chr>               <dbl>    <dbl> <dbl>
#> 1 Ford Pantera L       15.8        8   264
#> 2 Ferrari Dino         19.7        6   175
#> 3 Maserati Bora        15          8   335
#> 4 Volvo 142E           21.4        4   109
```

Wir haben nun einen Datensatz in R, mit dem wir dann weitere Analysen anstellen können. Nehmen wir einmal an, wir wollen eine weitere Spalte hinzufügen (Verbrauch/PS) und dann den Datensatz im Ordner `data/tidy` speichern. Ohne auf die Modifikation des Data Frames einzugehen können wir die Funktion `write_csv` verwenden um den Datensatz zu speichern. Hierzu geben wir den neuen Data Frame als erstes, und den Pfad als zweites Argument an:

```
auto_daten_neu <- mutate(auto_daten, Verbrauch_pro_PS = Verbrauch/PS)
write_csv(auto_daten_neu, here("data/tidy/NeueDaten.csv"))
```

Es wird ein späteres Kapitel (und einen späteren Vorlesungstermin) geben, in dem wir uns im Detail mit dem Lesen, Schreiben und Manipulieren von Datensätzen beschäftigen.

A
Bibliography

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Wickham, H. (2019). *Advanced R*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-0815384571.

Wickham, H. and Bryan, J. (2019). *Advanced R*. O'Reilly Media, Sebastopol, CA, 2nd edition. ISBN 978-1491910597.