R für die sozio-ökonomische Forschung

Version 0.3.0

Dr. Claudius Gräbner

2019-11-06

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).

Verhältnis zur Vorlesung

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
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: Wiederholung: Wahrscheinlichkeitstheorie	Wiederholung grundlegender Konzepte der Wahrscheinlichkeitstheorie und ihrer Implementierung in R	Optional; wird für die quantitativen VL vorausgesetzt
C: Wiederholung: Deskriptive Statistik	Wiederholung grundlegender Konzepte der deskriptiven Statistik und ihrer Implementierung in R	Optional; wird für die quantitativen VL vorausgesetzt
C: Wiederholung: Drei grundlegende Verfahren der schließenden Statistik	Wiederholung von Parameterschätzung, Hypothesentests und Konfidenzintervalle und deren Implementierung in R	Optional; wird für die quantitativen VL vorausgesetzt

Kapitel	Zentrale Inhalte	Verwandter Vorlesungstermin
E: 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
19.10.19	0.1.0	Erste Version veröffentlicht
03.11.19	0.2.0	Markdown-Anhang hinzugefügt
04.11.19	0.3.0	Anhänge zur Wiederholung grundlegender Statistik hinzugefügt
06.11.19	0.4.0	Kapitel zu linearen Modellen hinzugefügt

Lizenz

Chapter 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.

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!

[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.

Chapter 2

Einrichtung

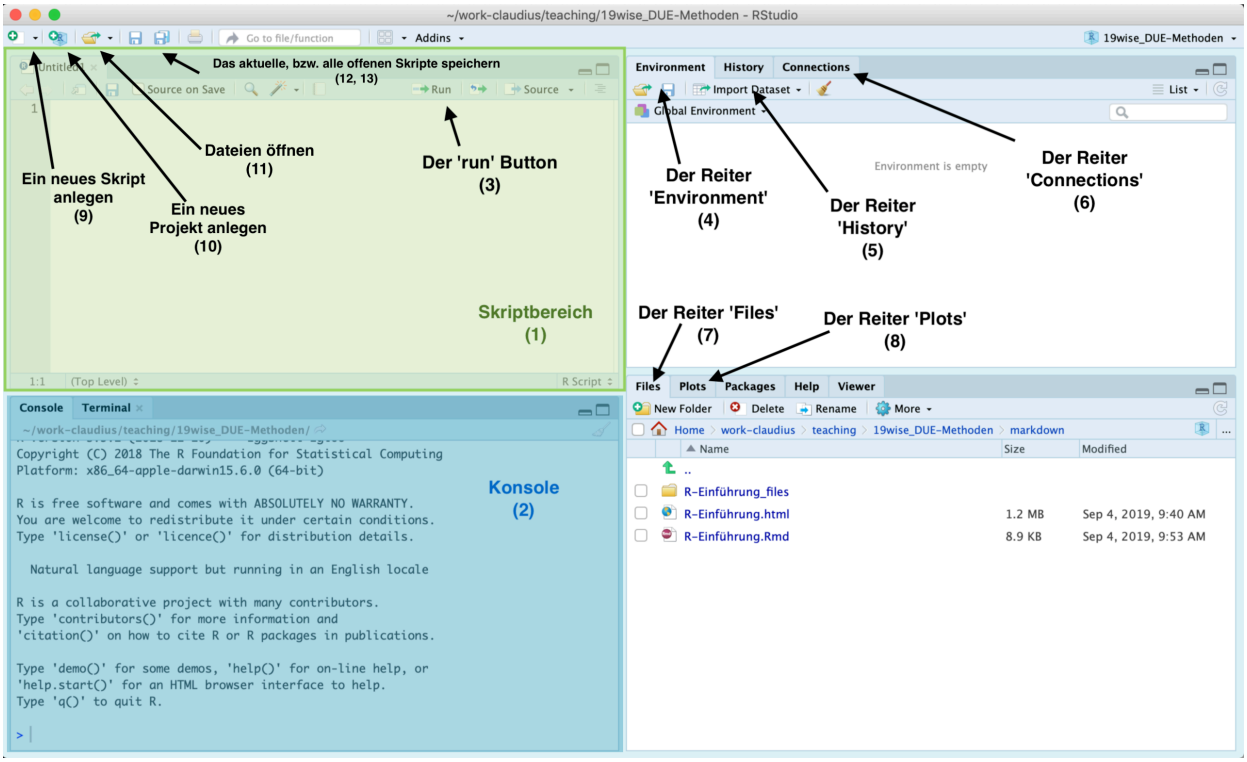
2.1 Installation von R und R-Studio

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 ar-

beiten 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

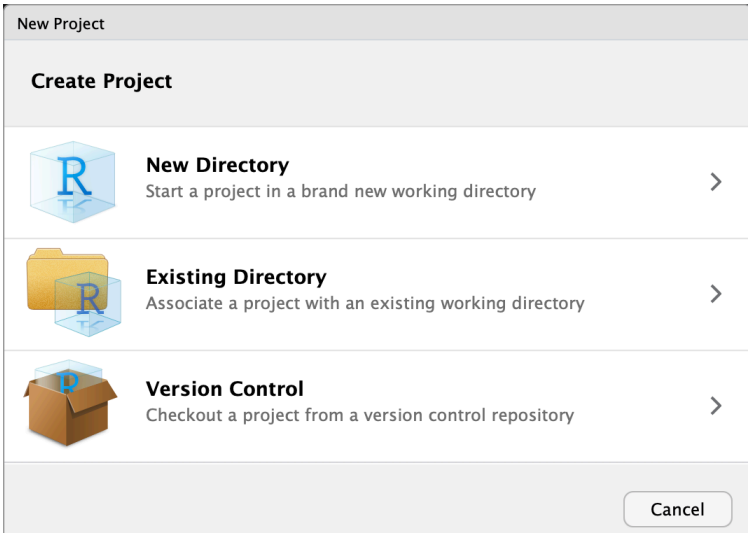
Zuerst müssen Sie sich für einen Ordner auf Ihrem Computer entscheiden, in dem alle Daten, die mit ihrem Projekt zu tun haben, also Daten, Skripte, Abbildungen, etc. gespeichert werden sollen und diesen Ordner gegebenenfalls neu erstellen. Es macht Sinn, einen solchen Ordner mit einem informativen Namen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

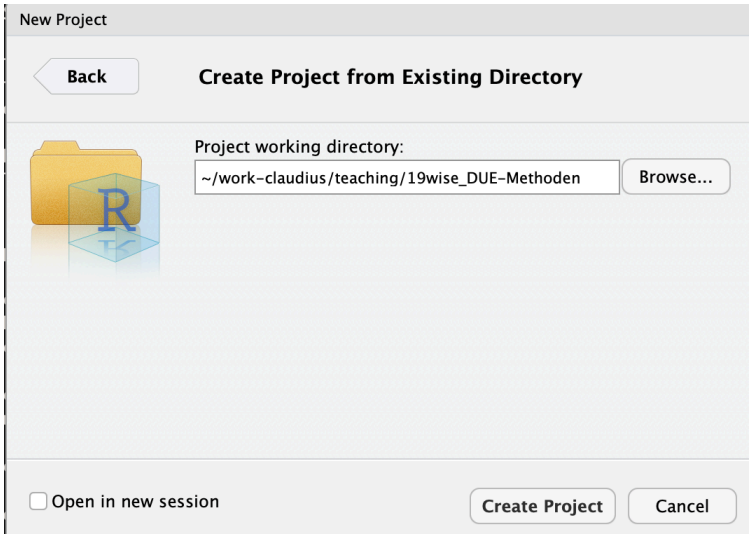
Wir möchten nun R Studio mitteilen den in Schritt 1 erstellten Ordner als R Projekt zu behandeln. Damit wird nicht nur dieses Ordner als Root-Verzeichnis festgelegt, man kann auch die Arbeitshistorie eines Projekts leich wiederherstellen und es ist einfacher, das Projekt auf verschiedenen Computern zu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

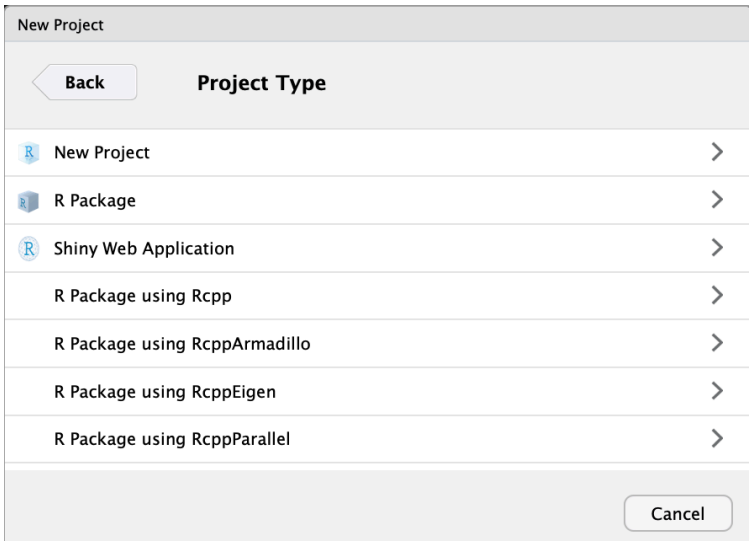


Falls Sie in Schritt 1 den Projektordner bereits erstellt haben wählen Sie hier `Existing Directory`, ansonsten erstellen Sie einen neuen Projektordner gleich mit dem Projektfile mit indem Sie `New Directory` auswählen.

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.

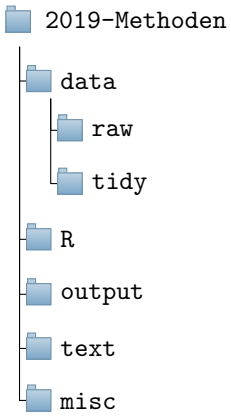
Insgesamt sollten die folgenden Ordner im Projektordner erstellt werden:

- 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.

Chapter 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]

[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').

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.

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
```

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:

```
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:

[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.

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

```r
#' 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:

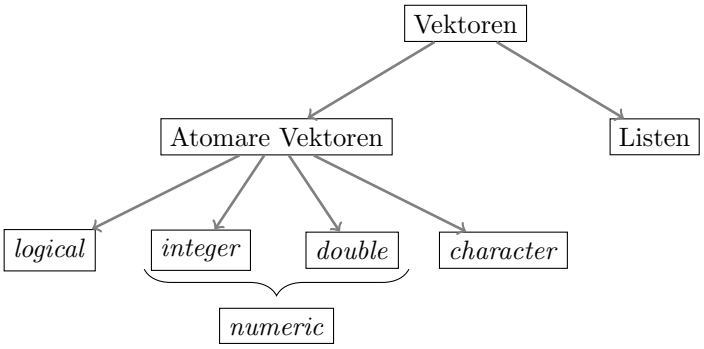
- `logical` (logische Werte): es gibt zwei logische Werte, `TRUE` und `FALSE`, welche auch mit `T` oder `F` abgekürzt werden können
- `integer` (ganze Zahlen): das sollte im Prinzip selbsterklärend sein, allerding muss den ganzen Zahlen in R immer der Buchstabe `L` folgen, damit die Zahl tatsächlich als ganze Zahl interpretiert wird.[5]

[5] Diese auf den ersten Blick merkwürdige Syntax hat historische Gründe: als der integer Typ in die R Programmiersprache

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

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 <- 22.0
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 <- 1.0
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"
```

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.

```
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 <- 2.0 # 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 "           "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"  "Versuch Nr. 3"  "Versuch Nr. 4"
#>  [5] "Versuch Nr. 5"  "Versuch Nr. 6"  "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"  "Tag: 5"  "Tag: 6"  "Tag: 7"
#>  [8] "Tag: 8"  "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"   "Tag 3: Mittwoch"
#> [4] "Tag 4: Donnerstag" "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.

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  5.0  5.5  6.0  6.5  7.0  7.5
#> [15]  8.0  8.5  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)
```

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

```
#> [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" "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(2.0)
```

```
#> [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 rechteckig angeordnete 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]

```
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]

```
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.

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:

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

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

[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`.

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

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

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

```
#> [1] "male"   "male"   "male"   "female" "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 Gerbrauch zu schreiben, z.B. um für bestimmte Routinen Funktionen zu programmieren, zu dokumentieren und in verschiedenen Projekten verfügbar zu machen.[11]

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.

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

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

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: 0x7fcbcf8a7968>
#> <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: 0x7fcbcbd0cc08>
#> <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"         "package:plm"        "package:dplyr"
#>  [4] "package:data.table" "package:stats"      "package:graphics"
#>  [7] "package:grDevices"  "package:utils"      "package:datasets"
#> [10] "package:methods"    "Autoloads"          "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.

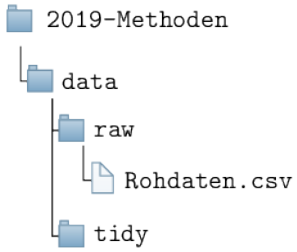
3.6 Kurzer Exkurs zum Einlesen und Schreiben von Daten

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:



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]

```
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
```

[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.

```
#> # 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.

Appendix A

Eine kurze Einführung in R Markdown

Hier gibt es eine kurze Einführung in `R Markdown`. Wir beschränken uns dabei auf die grundlegende Idee von Markdown, da die konkrete Syntax im Internet an zahlreichen Stellen wunderbar erläutert ist und man das konkrete Schreiben am besten in der Anwendung lernt.

A.1 Markdown vs. R-Markdown

Bei `Markdown` handelt es sich um eine sehr einfache Auszeichnungssprache, d.h. eine Programmiersprache, mit der schön formatierte Texte erstellt werden können und die gleichzeitig auch für Menschen sehr einfach lesbar ist. Dateien, die in Markdown geschrieben sind, sind gewöhnlicherweise an der Endung `.md` zu erkennen.

R-Markdown stellt man sich am besten als eine Kombination von Markdown und R vor: R-Markdown Dateien, die immer durch die Dateiendung `.Rmd` gekennzeichnet sind, bestehen sowohl aus Markdown-Code, als auch aus R-Code. Das bedeutet, dass man sein Forschungsprojekt gleichzeitig erklären und durchführen kann. Im Prinzip können ganze Forschungspapiere in R-Markdown verfasst werden und damit vollständig reproduzierbar gestaltet werden.

A.2 Installation von R-Markdown

Für den Fall, dass Sie mit R-Studio arbeiten brauchen Sie lediglich das Paket `rmarkdown` zu installieren:

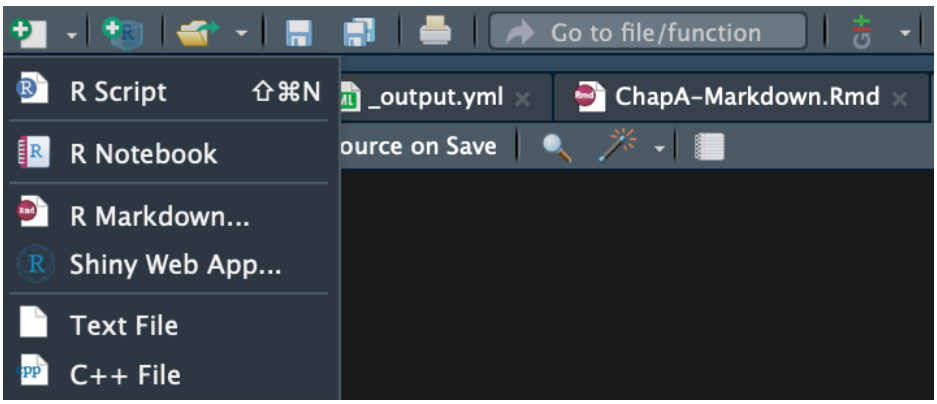
```
install.packages('rmarkdown')
```

Das Standardformat für R-Markdown Dokumente ist html. Wenn Sie aber auch PDF Dokumente erstellen wollen, müssen Sie auf Ihrem Computer LaTex installieren. Hierfür finden sich zahlreiche Anleitungen im Internet (z.B. hier oder hier).

A.3 Der R-Markdown Workflow

A.3.1 Ein neues R-Markdown Dokument erstellen

R-Studio macht es Ihnen sehr leicht R-Markdown Dokumente zu erstellen. Klicken Sie einfach auf den Button `Neu` und wählen dort dann `R Markdown` aus, wie auf folgendem Screenshot zu sehen ist:



Im folgenden Fenster können Sie getrost die Standardeinstellungen so wie vorgeschlagen belassen, da Sie alles später noch sehr leicht ändern können.

Sie sehen nun eine Datei, das bereits einigen Beispielcode enthält und damit schon einen Großteil der Syntax illustriert.

Ein R-Markdown Dokument besteht in der Regel aus zwei Teilen: dem Titelblock und dem darunter folgenden Dokumentenkörper:

A.3.2 Der Titelblock

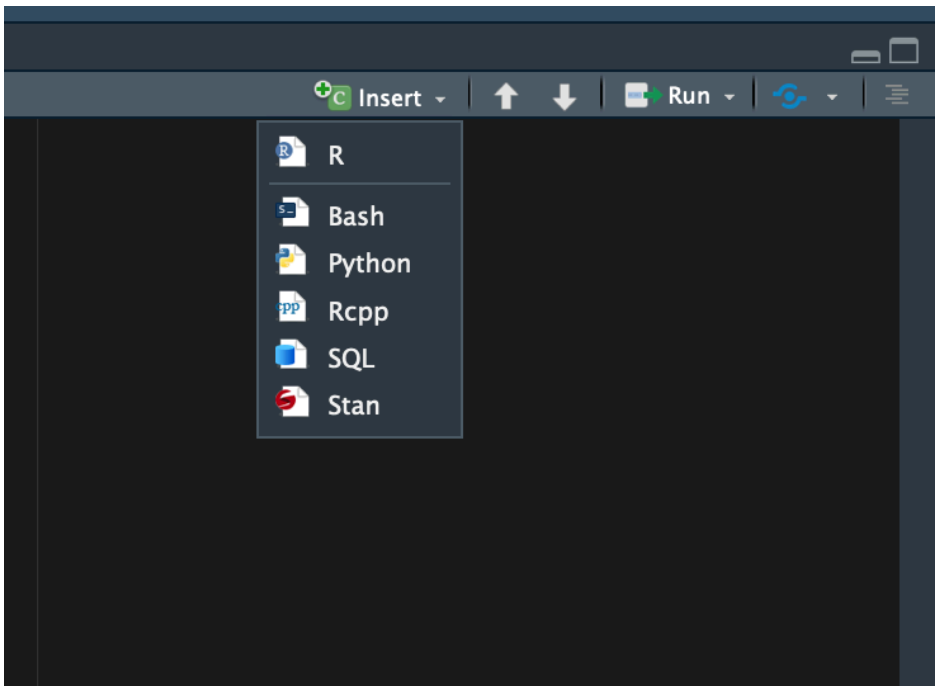
Der Titelblock ist immer durch zwei Zeilen mit dem Inhalt "---" oben und unten abgegrenzt. Die Syntax des Titelblocks folgt der Sprache YAML, aber das hat wenig praktische Relevanz. Im Titelblock werden alle globalen Einstellungen für das Dokument vorgenommen. Für einfache Dokumente muss nur wenig an den Standardeinstellungen geändert werden, aber im Laufe der Zeit werden Sie merken, dass Sie über den YAML-Block Ihr Dokument zu ganz großen Teilen individualisieren können. In der Regel finden Sie alle Antworten durch Googlen, daher werde ich hier nicht weiter auf den Header eingehen.

A.3.3 Der Textkörper

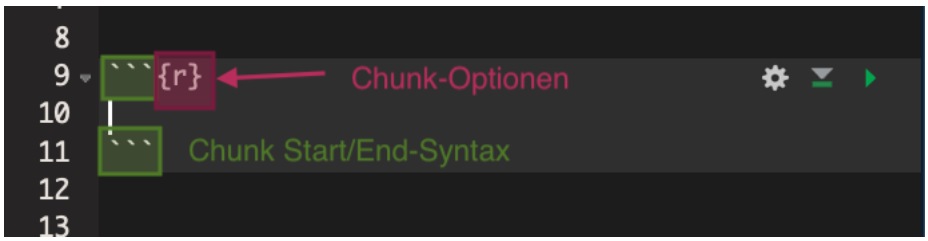
Der Textkörper besteht aus normalem Text, welcher in der Markdown Syntax geschrieben ist, und so genannten *Chunks*. Für die wirklich einfache Syntax für normalen Text gibt es zahlreiche gute Anleitungen im Internet, z.B. dieses eingängige Cheat Sheet.

Innerhalb der Chunks können Sie Code in einer beliebigen Programmiersprache schreiben, insbesondere auch in R. Die Syntax unterscheidet sich dabei überhaupt nicht von einem normalen R Skript.

Um einen Chunk zu Ihrem Dokument hinzuzufügen klicken Sie oben rechts im Skripbereich auf 'Insert' und wählen `R` aus:

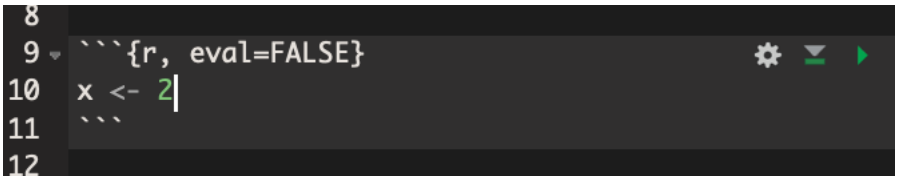


Daraufhin wird an der Stelle des Cursors ein Chunk in Ihr Dokument eingefügt. Dieser Chunk wird in der ersten und letzten Zeile durch ``` begrenzt. In der ersten Zeile wird zusätzlich innerhalb von geschweiften Klammern die Programmiersprache des Chunks definiert:



Darüber hinaus kann das Ausführverhalten für den Chunk durch weitere Argumente innerhalb der geschweiften Klammer weiter spezifiziert werden.

Häufig möchten Sie z.B., dass der Code im Chunk zwar im Dokument angezeigt, aber nicht ausgeführt werden soll. Dies können Sie durch die Option `eval=FALSE` erreichen. In diesem Fall sähe Ihr Chunk so aus:



In diesem Beispiel wird die Zuweisung `x <- 4` bei der Kompillierung des Dokuments nicht ausgeführt.

Eine gute Übersicht über die Optionen, die Ihnen offen stehen, finden Sie hier oder durch Googlen.

Sie können einzelne Chunks auch schon vor dem Kompillieren des Dokuments ausführen indem Sie auf das Play-Zeichen oben links beim Chunk drücken. Damit erhalten Sie eine Vorschau auf das Ergebnis.

A.3.4 Kompillieren von Dokumenten

Der Prozess, der aus dem Quellcode ihres Dokuments (also allem was in der `.Rmd` Datei geschrieben ist) das fertige Dokument erstellt, wird *Kompillieren* genannt. Dabei wird aus dem `.Rmd` Dokument ein gut lesbares `.html` oder `.pdf` Dokument erstellt, wobei alle Chunks normal ausgeführt werden (es sei denn dies wird durch die Option `eval=FALSE` verhindert).

Grundsätzlich gibt es zwei Möglichkeiten ein Dokument zu kompillieren: über die entsprechende R-Funktion, oder über den Knit-Button in R-Studio.

Die klassische Variante verwendet die Funktion `render()` aus dem Paket `rmarkdown`. Die wichtigsten Argumente sind dabei die folgenden: `input` spezifiziert die zu kompillierende `.Rmd`-Datei, `output_format` das für den Output gewünschte Format[1] und `output_file` den Pfad und den Namen der zu erstellenden Outputdatei.

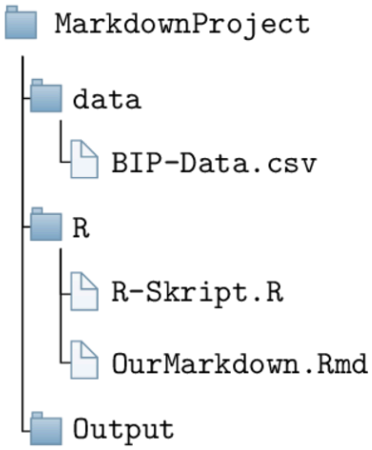
Wenn Sie also das Dokument `FirstMarkdown.Rmd` kompillieren wollen und den Output unter `Output/OurMarkdown.html` als html-Datei speichern wollen, dann können Sie das mit folgendem Code, vorausgesetzt die Datei `FirstMarkdown.Rmd` liegt im Unterordner `R`:

```
render(input = "R/FirstMarkdown.Rmd",
       output_format = "html",
       output_file = "output/FirstMarkdown.html")
```

Weitere Informationen zu den Parametern finden Sie wie immer über die `help()` Funktion. Alternativ können Sie auch den Button `Knit` in der R-Studio Oberfläche verwenden. Das ist in der Regel bequemer, lässt aber weniger Individualisierung zu.

A.4 Relative Pfade in Markdown-Dokumenten

Der problematischste Teil beim Arbeiten mit R-Markdown ist der Umgang mit relativen Pfaden. Um das Problem zu illustrieren nehmen wir einmal folgende Ordnerstruktur an, wobei der Ordner `MarkdownProject` unser Arbeitsverzeichnis ist:



Das Problem ist nun, dass wenn Sie eine R-Markdown Datei kompillieren, diese Datei alle Pfade **nicht** ausgehend von Ihrem Arbeitsverzeichnis interpretiert, sondern vom *Speicherort* der `.Rmd`-Datei. Das ist natürlich hochproblematisch, denn stellen Sie sich vor, Sie möchten in Ihrem R-Markdown-Skript die Datei `data/BIP-Data.csv` einlesen. Normalerweise würden Sie dafür den folgenden Code verwenden:

```
bip_data <- fread("data/BIP-Data.csv")
```

Zwar würde der Code in einem R-Skript, z.B. in `R/R-Skript.R` perfekt funktionieren. In einem R-Markdown Dokument, das nicht im Arbeitsverzeichnis direkt gespeichert ist, jedoch nicht. Da in R-Markdown-Dokumenten alle Pfade relativ des Speicherorts des Dokuments interpretiert werden, müssten wir hier schreiben:[2]

```
bip_data <- fread("../data/BIP-Data.csv")
```

Das wäre allerdings unschön, weil wir dann unterschiedlichen Codes in Skripten und in R-Markdown-Dokumenten verwenden müssten und das Ganze dadurch deutlich verwirrender werden würde.

Es wäre also schön, wenn R automatisch wüsste, was das Arbeitsverzeichnis des aktuellen Projekts ist und dieses automatisch berücksichtigt, unabhängig davon ob wir mit einem `.R` oder `.Rmd` Dokument arbeiten und wo dieses Dokument innerhalb unserer Projekt-Struktur gespeichert ist.

[1] R-Markdown Dateien können in sehr viele verschiedene Formate kompilliert werden, das am häufigsten verwendete Format ist jedoch `html`. Eine Übersicht finden Sie hier.

[2] Mit `../` bewegt man sich bei einem relativen Pfad einen Ordner nach oben.

Zum Glück können wir genau das mit Hilfe des Pakets here erreichen.[3] Das Paket enthält eine Funktion `here()` die als Argument einen Dateinamen oder einen relativen Pfad akzeptiert, und daraus einen absoluten Pfad auf dem Computer, auf dem der Code gerade ausgeführt wird, konstruiert.

Wir können also unseren Code von oben einfach folgendermaßen umschreiben:

```
bip_data <- fread(here("data/BIP-Data.csv"))
```

In dieser Form funktioniert er sowohl in `.R` als auch `.Rmd` Dateien ohne Probleme.

> **Hinweis I:** Die Funktion `here()` verwendet verschiedene Heuristiken um das Arbeitsverzeichnis des aktuellen Projekt herauszufinden. Darunter fällt auch das Suchen nach einer `.Rproj` Datei. Überhaupt funktionieren die Heuristiken in der Regel wunderbar und können für Ihren konkreten Fall über die Funktion `dr_here()` angezeigt werden. Um ganz sicherzugehen sollte man aber immer in das Arbeitsverzeichnis eine Datei `.here` ablegen. Diese kann manuell, oder über die Funktion `set_here()`, erstellt werden.
>
> **Hinweis II:** Die Verwendung von `here()` ist essenziell, wenn Ihre R-Markdown Dokumente auf mehreren Computern funktionieren sollen. Daher ist die Verwendung in den Arbeitsblättern **verpflichtend**.

A.5 Weitere Quellen

Eine gute Übersicht über die häufigsten Befehle enthält dieses Cheat Sheet. Eine sehr umfangreiche Einführung bietet das Online-Buch R Markdown: The Definitive Guide. Aber auch darüber hinaus finden sich im Internet zahlreiche Beispiele für die R-Markdown-Syntax. Dieses Skript wurde übrigens in R Bookdown, einer Erweiterung von R-Markdown für Bücher, geschrieben.

[3] Tatsächlich ist `here` dermaßen praktisch, dass ich empfehle grundsätzlich alle Pfade in jedem Projekt - ob R-Markdown oder nicht - mit Hilfe von `here` anzugeben.

Appendix B

Wiederholung: Wahrscheinlichkeitstheorie

In diesem Kapitel werden Grundlagen der Wahrscheinlichkeitstheorie wiederholt. Die zentralen Themen sind dabei:

- Der Zusammenhang zwischen Wahrscheinlichkeitstheorie und Statistik
- Grundbegriffe der Wahrscheinlichkeitstheorie und Statistik
- Zufallsvariablen
- Diskrete und stetige Verteilungen

Grundkonzepte der deskriptiven und schließenden Statistik (insb. Parameterschätzung, Hypothesentests und die Berechnung von Konfidenzintervallen) werden in den beiden Anhängen zur deskriptiven und schließenden Statistik wiederholt.

Für den Code in diesem Kapitel werden die folgenden Pakete verwendet:

```
library(here)
library(tidyverse)
library(ggpubr)
library(latex2exp)
library(icaeDesign)
library(data.table)
```

B.1 Einleitung: Wahrscheinlichkeitstheorie und Statistik

Statistik und Wahrscheinlichkeitstheorie sind untrennbar miteinander verbunden. In der Wahrscheinlichkeitstheorie beschäftigt man sich mit Modellen von Zufallsprozessen, also Prozessen, deren Ausgang nicht exakt vorhersehbar ist. Häufig spricht man von *Zufallsexperimenten*.

Die Wahrscheinlichkeitstheorie entwickelt dabei Modelle, welche diese Zufallsexperimenten und deren mögliche Ausgänge beschreiben und dabei den möglichen Ausgängen Wahrscheinlichkeiten zuordnern. Diese Modelle werden *Wahrscheinlichkeitsmodelle* genannt.

In der Statistik versuchen wir anhand von beobachteten Daten herauszufinden, welches Wahrscheinlichkeitsmodell gut geeignet ist, den die Daten generierenden Prozess (*data generating process* - DGP) zu beschreiben. Das ist der Grund warum man für Statistik auch immer Kenntnisse der Wahrscheinlichkeitstheorie braucht.

> Kurz gesagt: in der Wahrscheinlichkeitstheorie wollen wir mit Hilfe von Wahrscheinlichkeitsmodellen Daten vorhersagen, in der Statistik mit Hilfe bekannter Daten Rückschlüsse auf die zugrundeliegenden Wahrscheinlichkeitsmodelle ziehen.

B.2 Grundbegriffe der Wahrscheinlichkeitstheorie

Ein wahrscheinlichkeitstheoretisches Modell besteht *immer* aus den folgenden drei Komponenten:

Ergebnisraum: diese Menge Ω enthält alle möglichen Ergebnisse des modellierten Zufallsexperiments. Das einzelne Ergebnis bezeichnen wir mit ω.

> **Beispiel:** Handelt es sich bei dem Zufallsexperiment um das Werfen eines normalen sechseitigen Würfels gilt $\Omega = \{1, 2, 3, 4, 5, 6\}$. Wenn der Würfen gefallen ist, bezeichnen wir die oben liegende Zahl als das Ergebnis ω des Würfelwurfs, wobei hier gilt ω_1 = "Der Würfel zeigt 1", u.s.w.

Ereignisse: unter Ereignissen $A, B, C, ...$ verstehen wir die Teilmengen des Ergebnisraums. Ein Ereignis enthält ein oder mehrere Elemente des Ergebnisraums. Enthält ein Ereignis genau ein Element, sprechen wir von einem *Elementarereignis*.

> **Beispiel:** "Es wird eine gerade Zahl gewürfelt" ist ein mögliches Ereignis im oben beschriebenen Zufallsexperiment. Das Ereignis - nennen wir es hier A - tritt ein, wenn ein Würfelwurf mit dem Ergebnis "2", "4" oder "6" endet. Also: $A = \{\omega_2, \omega_4, \omega_6\}$ Das Ereignis B "Es wird eine 2 gewürfelt" tritt nur ein, wenn das Ergebnis des Würfelwurfs eine 2 ist: $B = \{\omega_2\}$. Entsprechend nennen wir es ein *Elementarereignis*.

Da es sich bei Ereignissen um Mengen handelt können wir die typischen mengentheoretischen Konzepte wie 'Vereinigung', 'Differenz' oder 'Komplement' zu ihrer Beschreibung verwenden:

Konzept	Symbol	Übersetzung
Schnittmenge	$A \cap B$	A und B
Vereinigung	$A \cup B$	A und/oder B
Komplement	A^c	Nicht A
Differenz	$A \setminus B = A \cap B^c$	A ohne B

Wahrscheinlichkeiten: jedem *Ereignis* A wird eine Wahrscheinlichkeit $\mathbb{P}(A)$ zugeordnet. Wahrscheinlichkeiten können aber nicht beliebige Zahlen sein. Vielmehr müssen sie im Einklang mit den drei *Axiomen von Kolmogorow* stehen:

1. Für jedes Ereignis A gilt: $0 \leq \mathbb{P}(A) \leq 1$
2. Das sichere Ereignis Ω umfasst den ganzen Ergebnisraum und es gilt entsprechend $\mathbb{P}(\Omega) = 1$.
3. Es gilt: $\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B)$ falls $A \cap B = \emptyset$, also wenn sich A und B gegenseitig ausschließen.

Aus diesen Axiomen lassen sich eine ganze Menge Sätze heraus ableiten, auf die wir im folgenden aber nicht besonders eingehen wollen. Die Grundidee ist aber, bestimmten Ereignissen von Anfang an bestimmte Wahrscheinlichkeiten zuzuordnen, und die Wahrscheinlichkeiten für andere Ereignisse dann aus den eben beschriebenen Regeln abzuleiten.

Je nach Art des Ergebnisraums Ω unterscheiden wir zwei grundsätzlich verschiedene Arten von Wahrscheinlichkeitsmodellen: ist Ω **abzählbar** handelt es sich um ein **diskretes Wahrscheinlichkeitsmodell**. Der Würfelwurf oder ein Münzwurf sind hierfür Beispiele: die Menge der möglichen Ergebnisse ist hier klar abzählbar.[1]

Ist Ω **nicht abzählbar** handelt es sich dagegen um ein **stetiges Wahrscheinlichkeitsmodell**. Ein Beispiel hierfür wäre das Fallenlassen von Steinen und die Messung der Falldauer. Die einzelnen Ereignisse wären dann die Falldauer und es würde gelten, dass $\Omega = \mathbb{R}^+$ und $\mathbb{R}^+$ ist nicht abzählbar.

Welches Modell für den konkreten Anwendungsfall vorzuziehen ist, muss auf Basis von theoretischen Überlegungen entschieden werden.

B.3 Diskrete Wahrscheinlichkeitsmodelle

Wenn wir die Wahrscheinlichkeit für das Eintreten eines Ereignisses A erfahren möchten können wir im Falle eines diskreten Ergebnisraums einfach die Eintrittswahrscheinlichkeiten für alle Ergebnisse, die zu A

[1] Wir nennen eine Menge abzählbar wenn sie mit Hilfe der ganzen Zahlen $\mathbb{N}$ indiziert werden kann. Das bedeutet, dass auch unendlich große Mengen als abzählbar gelten können.

gehören, aufsummieren:

$$\mathbb{P}(A) = \sum_{\omega \in A} \mathbb{P}(\{\omega\})$$

Beispiel: Beim Werfen eines sechseitigen Würfels ist die Wahrscheinlichkeit für das Ereignst "Es wird eine gerade Zahl gewürfelt": $\mathbb{P}(2) + \mathbb{P}(4) + \mathbb{P}(6) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$.

Von Interesse ist häufig aus den Wahrscheinlichkeiten für zwei Ereignisse, A und B, die Wahrscheinlichkeit für $A \cap B$, also die Wahrscheinlichkeit, dass beide Ereignisse auftreten, zu berechnen. Leider ist das nur im Spezialfall der **stochastischen Unabhängigkeit** möglich. Stochastische Unabhängigkeit kann immer dann sinnvollerweise angenommen werden, wenn zwischen den beteiligten Ereignissen kein kausaler Zusammenhang besteht. In diesem Fall gilt dann:

$$\mathbb{P}(A \cap B) = \mathbb{P}(A) \cdot \mathbb{P}(B)$$

Beispiel für stochastische Unabhängigkeit: Es ist plausibel anzunehmen, dass es keinen kausalen Zusammenhang zwischen zwei aufeinanderfolgenden Münzwürfen gibt. Entsprechend sind die Ereignisse A: "Zahl im ersten Wurf" und B: "Kopf im zweiten Wurf" stochastisch unabhängig und $\mathbb{P}(A \cap B) = \mathbb{P}(A) \cdot \mathbb{P}(B) = \frac{1}{4}$.

Beispiel für stochastische Abhängigkeit: Ein anderer Fall liegt vor, wenn wir die Ereignisse C: "Die Summe beider Würfe ist 6" und D: "Der erste Wurf zeigt eine 2." betrachten. Hier ist offensichtlich, dass ein kausaler Zusammenhang zwischen den beiden Würfen und den Ereignissen besteht. Es gilt: $\mathbb{P}(C \cap D) = \mathbb{P}(\{2, 4\}) = \frac{1}{36}$. Würden wir die Wahrscheinlichkeiten einfach multiplizieren erhielten wir allerdings $\mathbb{P}(C) \cdot \mathbb{P}(D) = \frac{5}{36} \cdot \frac{1}{6} = \frac{5}{216}$, wobei $\mathbb{P}(C) = \frac{5}{36}$.

Ein weiteres wichtiges Konzept ist das der **bedingten Wahrscheinlichkeit**: die bedingten Wahrscheinlichkeit von A gegeben B, $\mathbb{P}(A|B)$, bezeichnet die Wahrscheindlichkeit für A, wenn wir wissen, dass B bereits eingetreten ist.

Es gilt dabei:[2]

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}$$

Beispiel: Sei A: "Der Würfel zeigt eine 6" und B: "Der Würfelwurf zeigt eine gerade Zahl". Wenn wir bereits wissen, dass B eingetreten ist, ist $\mathbb{P}(A)$ nicht mehr $\frac{1}{6}$, weil wir ja wissen, dass 1, 3 und 5 nicht auftreten können. Vielmehr gilt $\mathbb{P}(A|B) = \frac{1/6}{1/2} = \frac{1}{3}$.

B.3.1 Bayes Theorem und Gesetz der total Wahrscheinlichkeiten

Ganz wichtig: es gilt *nicht notwendigerweise* $\mathbb{P}(A|B) = \mathbb{P}(B|A)$. Vielmehr gilt nach dem **Satz von Bayes**:

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)} = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}$$

Ein in Beweisen sehr häufig verwendeter Zusammenhang ist das **Gesetz der totalen Wahrscheinlichkeit**: seien $A_1, ..., A_k$ Ergeignisse, die sich nicht überschneiden und gemeinsam den kompletten Ereignisraum Ω abdecken, dann gilt:

$$\mathbb{P}(B) = \sum_{i=1}^{k} \mathbb{P}(B|A_k)\mathbb{P}(A_k)$$

Auch wenn das erst einmal sperrig aussieht, ist der Zusammenhang sehr praktisch und wird häufig in Beweisen in der Stochastik verwendet.

[2] An der Formel wird noch einmal deutlich, dass wenn A und B stochastisch unabhängig sind wir nichts von B über A und umgekehrt lernen können, also gilt: $\mathbb{P}(A|B) = \mathbb{P}(A)$ und $\mathbb{P}(B|A) = \mathbb{P}(B)$.

B.3.2 Diskrete Zufallsvariablen

Bei Zufallsvariablen (ZV) handelt es sich um besondere *Funktionen*. Die Definitionsmenge einer Zufallsvariable ist immer der zurgundeliegende Ergebnisraum Ω, die Zielmenge ist i.d.R. $\mathbb{R}$, sodass gilt:

$$X : \Omega \to \mathbb{R}, \omega \mapsto X(\omega)$$

Im Kontext von ZV sprechen wir häufig nicht von dem zugrundeliegenden Ergebnisraum Ω, sondern - inhaltlich äquivalent - vom *Wertebereich von X*, bezeichnet als W_X.

In der Regel bezeichnen wir Zufallsvariablen (ZV) mit Großbuchstaben und die konkrete Realisation einer ZV mit einem Kleinbuchstaben, sodass $\mathbb{P}(X = x)$ die Wahrscheinlichkeit angibt, dass die ZV X den konkreten Wert x annimmt. Bei x sprechen wir von einer *Realisierung* der ZV X. Wir nehmen für die weitere Notation an, dass $W_X = \{x_1, x_2, ..., x_K\}$ und bezeichnen das einzelne Element mit x_k mit $1 \leq k \leq K$.

Dies bedeutet streng genommen, dass die ZV selbst nicht als zufällig definiert wird. Zufällig ist nur der Input ω der entsprechenden Funktion $X : \Omega \to X(\omega)$, also z.B. ein Würfelwurf. Der funktionale Zusammenhang zwischen Funktionswert $X(\omega)$ und dem Input ω ist hingegen eindeutig.

Das bedeutet streng genommen, dass die ZV nicht *selbst* zufällig ist, sondern ihr Input ω. Das impliziert, dass wenn ein Zufallsexperiment zweimal das gleiche Ergebnis ω hat, ist auch der Wert $X(\omega)$ der gleiche.

Das mag im Moment ein wenig nach 'Pfennigfuchserei' aussehen, die Unterscheidung zwischen dem nicht-zufälligem funtionalen Zusammenhangs, aber einem zufälligen Input bei ZV ist wichtig, um den Sinn in vielen fortgeschrittenen Beiträgen im Bereich der Ökonometrie zu sehen.

Den unterschiedlichen Realisierungen von einer ZV haben jeweils Wahrscheinlichkeiten, die von den Wahrscheinlichkeiten der zugrundeliegenden Ergebnisse des modellierten Zufallsexperiments abhängen.

Produkte und Summen von ZV sind selbst wieder Zufallsvariables. Man addiert bzw. multipliziert ZV indem man ihre Werte addiert bzw. mutlipliziert.

Im Falle von diskreten ZV können wir eine Liste erstellen, die für alle möglichen Werte $x_k \in W_X$ die jeweilige Wahrscheinlichkeit $\mathbb{P}(X = x_k)$ angibt.[3] Diese Liste nennen wir **Wahrscheinlichkeitsverteilung** (*Probability Mass Function*, PMF) von X und sie werden häufig visuell dargestellen. Um diese Liste zu erstellen verwenden wir die zu X gehörende **Wahrscheinlichkeitsfunktion**, $(p(x_k))$,die uns für jedes Ergebnis die zugehörige Wahrscheinlichkeit gibt:[4]

$$p(x_k) = \mathbb{P}(X = x_k)$$

Wenn wir eine ZV analysieren tun wir dies in der Regel durch eine Analyse ihrer Wahrscheinlichkeitsverteilung. Zur genaueren Beschreibung einer ZV wird entsprechend häufig einfach die Wahrscheinlichkeitsfunktion angegeben.

Im folgenden wollen wir einige häufig auftretende Wahrscheinlichkeitsverteilungen kurz besprechen. Am Ende des Abschnitts findet sich dann ein tabellarischer Überblick. Doch vorher wollen wir uns noch mit den wichtigsten **Kennzahlen einer Verteilung** vertraut machen. Denn wie Sie sich vorstellen können sind Wahrscheinlichkeitsverteilungen als Listen, die alle möglichen Realisierungen einer ZV enthalten ziemlich umständlich zu handhaben. Daher beschreiben wir Wahrscheinlichkeitsverteilungen nicht indem wir eine Liste beschreiben, sondern indem wir bestimmte Kennzahlen zu ihrer Beschreibung verwenden. Die wichtigsten Kennzahlen einer ZV X sind der **Erwartungswert** $\mathbb{E}(x)$ als *Lageparameter* und die **Standardabweichung** $\sigma(X)$ als *Streuungsmaß*.

Der Erwartungswert ist definiterter als die nach ihrer Wahrscheinlichkeit gewichtete Summe aller Elemente im Wertebereich von X und gibt damit die mittlere Lage der Wahrscheinlichkeitsverteilung an. Wenn W_X der Wertebereich von X ist, dann gilt:

$$\mathbb{E}(x) = \mu_X = \sum_{x_k \in W_X} p(x_k) x_k$$

[3] Aus den *Kolmogorow Axiomen* oben ergibt sich, dass die Summe all dieser Wahrscheinlichkeiten 1 ergeben muss: $\sum_{k \geq 1} \mathbb{P}(X = x_k) = 1$.

[4] Zu jeder Wahrscheinlichkeitsverteilung gibt es eine eindeutige Wahrscheinlichkeitsfunktion und jede Wahrscheinlichkeitsfunktion definiert umgekehrt eine eindeutig bestimmte diskrete Wahrscheinlichkeitsverteilung.

Beispiel: Der Erwartungswert einer ZV X, die das Werfen eines fairen Würfels beschreibt ist: $\mathbb{E}(X) = \sum_{k=1}^{6} k \cdot \frac{1}{6} = 3.5$.

Wie wir später sehen werden, wird der Erwartungswert in der empirischen Praxis häufig über den Mittelwert einer Stichprobe identifiziert.

Ein gängiges Maß für die Streuung einer Verteilung X ist die Varianz $Var(X)$ oder ihre Quadratwurzel, die Standardabweichung, $\sigma(X) = \sqrt{Var(X)}$. Letztere wird häufiger verwendet, weil sie die gleiche Einheit hat wie X:

$$Var(X) = \sum_{x_k \in W_X} [x_k - \mathbb{E}(X)]^2 \, p(x_k)$$

Beispiel: Die Standardabweichung einer ZV X, die das Werfen eines fairen Würfels beschreibt ist: $\sigma_X = \sqrt{\sum_k^6 [x_k - \mathbb{E}(X)]^2 \, p(x_k)} = \sqrt{5.83} \approx 2.414$.

Im folgenden wollen wir uns einige der am häufigsten verwendeten ZV und ihre Verteilungen genauer ansehen. Am Ende der Beschreibung jeder Funktion folgt ein Beispiel für eine Anwendung. Wenn Ihnen die theoretischen Ausführungen am Anfang etwas kryptisch erscheinen, empfiehlt es sich vielleicht erst einmal das Anwendungsbeispiel anzusehen.

B.3.3 Beispiel: die Binomial-Verteilung

Die vielleicht bekannteste diskrete Wahrscheinlichkeitsverteilung ist die Binomialverteilung $\mathcal{B}(n, p)$. Mit ihr modelliert man Zufallsexperimente, die aus einer Reihe von Aktionen bestehen, die entweder zum 'Erfolg' oder 'Misserfolg' führen.

Die Binomialverteilung ist eine Verteilung mit zwei **Parametern**. Parameter sind Werte, welche die Struktur der Verteilung bestimmen. In der Statistik sind wir häufig daran interessiert, die Paramter einer Verteilung zu bestimmen. Im Falle der Binomialverteilung gibt es die folgenden zwei Parameter: p gibt die Erfolgswahrscheinlichkeit einer einzelnen Aktion an (und es muss daher gelten $p \in [0, 1]$) und n gibt die Anzahl der Aktionen an. Daher auch die Kurzschreibweise $\mathcal{B}(n, p)$.

Beispiel: Wenn wir eine faire Münze zehn Mal werfen, können wir das mit einer Binomialverteilung mit $p = 0.5$ und $n = 10$ modellieren.

Die *Wahrscheinlichkeitsfunktion* $p(x)$ der Binomialverteilung ist die folgende, wobei x die Anzahl der Erfolge darstellt:

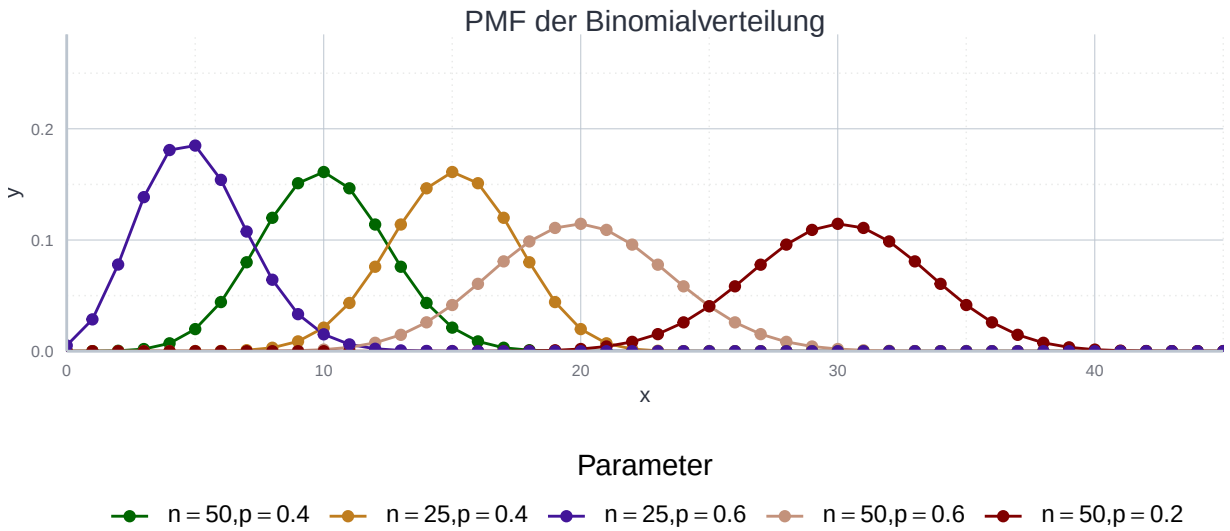
$$\mathbb{P}(X = x) = p(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

Dies ergibt sich aus den grundlegenden Wahrscheinlichkeitsgesetzen: $\binom{n}{x}$ ist der Binomialkoeffizient und gibt uns die Anzahl der Möglichkeiten wie man bei n Versuchen x Erfolge erziehlen kann. Dies multiplizieren wir mit der Wahrscheinlichkeit x-mal einen Erfolg zu erziehlen und $n - x$-mal einen Misserfolg zu erziehlen.

Wenn die ZV X einer Binomialverteilung mit bestimmten Parametern p und n folgt, dann schreiben wir $P \propto \mathcal{B}(n, p)$ und es gilt, dass $\mathbb{E}(X) = np$ und $\sigma(X) = \sqrt{np(1-p)}$.[5]

Im folgenden sehen wir eine Darstellung der Wahrscheinlichkeitsverteilung der Binomialverteilung für verschiedene Parameterwerte:

[5] Die Herleitung finden Sie im Statistikbuch Ihres Vertrauens oder auf Wikipedia.



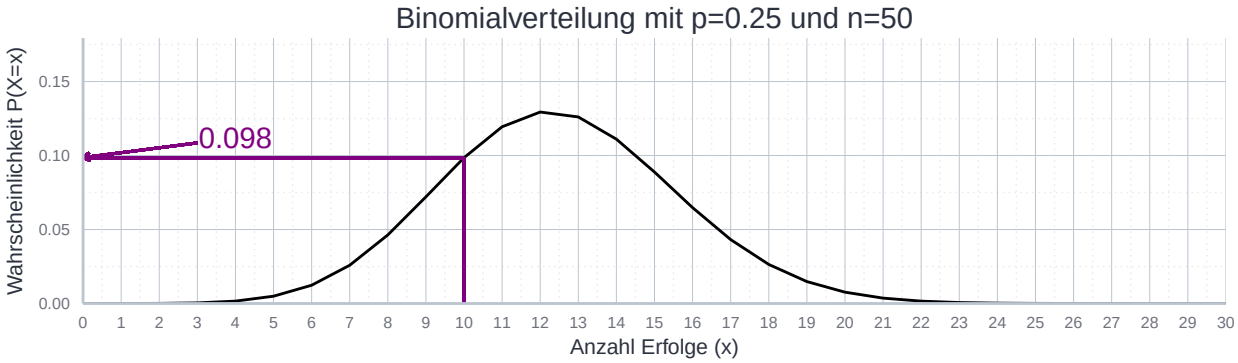
R stellt uns einige nützliche Funktionen bereit, mit denen wir typische Rechenaufgaben einfach lösen können:

Möchten wir die Wahrscheinlichkeit berechnen, genau x Erfolge zu beobachten, also $\mathbb{P}(X = x)$ geht das mit der Funktion `dbinom()`. Die notwendigen Argumente sind `x` für den interessierenden x-Wert, `size` für den Parameter n und `prob` für den Parameter p:

```
dbinom(x = 10, size = 50, prob = 0.25)
```

```
## [1] 0.09851841
```

Das bedeutet, wenn $X \propto B(50, 0.25)$, dann: $\mathbb{P}(X = 10) = 0.09852$. Die folgende Abbildung illustriert dies:



Natürlich können wir an die Funktion auch einen atomaren Vektor als erstes Argument übergeben:

```
dbinom(x = 5:10, size = 50, prob = 0.25)
```

```
## [1] 0.004937859 0.012344647 0.025864974 0.046341412 0.072086641 0.098518410
```

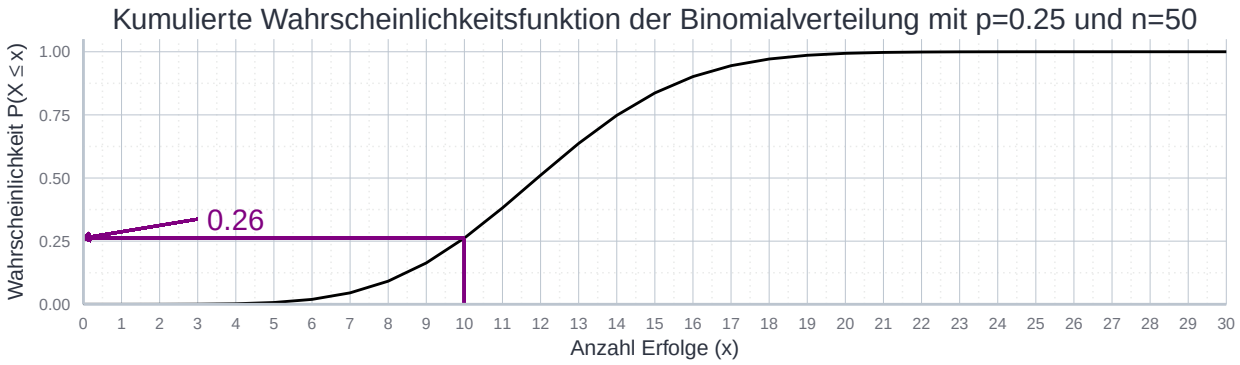
Häufig sind wir auch an der **kumulierten Wahrscheinlichkeitsfunktion** interessiert. Während uns die Wahrscheinlichkeitsfunktion die Wahrscheinlichkeit für genau x Erfolge angibt, also $\mathbb{P}(X = x)$, gibt uns die *kumulierte* Wahrscheinlichkeitsfunktion die Wahrscheinlichkeit für x oder weniger Erfolge, also $\mathbb{P}(X \leq x)$.

Die entsprechenden Werte für die kumulierten Wahrscheinlichkeitsfunktion erhalten wir mit der Funktion `pbinom()`, welche quasi die gleichen Argumente benötigt wie `dbinom()`. Nur gibt es anstatt des Parameters `x` jetzt einen Parameter `q`:

```
pbinom(q = 10, size = 50, prob = 0.25)
```

```
## [1] 0.2622023
```

Die Wahrscheinlichkeit 5 oder weniger Erfolge bei 5 Versuchen und einer Erfolgswahrscheinlichkeit von 25% zu erzielen beträgt also 25.2%:



Schlussendlich haben wir die Funktion `qbinom()`, welche als ersten Input eine Wahrscheinlichkeit `p` akzeptiert und dann den kleinsten Wert x findet, für den gilt, dass $\mathbb{P}(X = x) \geq p$.

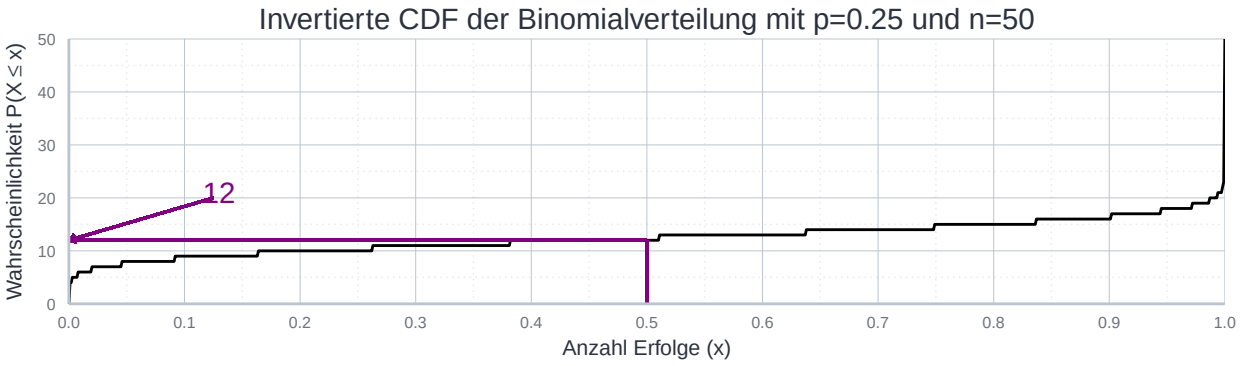
Wenn wir also wissen möchten wie viele Erfolge mit einer Wahrscheinlichkeit von 50% mindestens zu erwarten sind, dann schreiben wir:

```
qbinom(p = 0.5, size = 50, prob = 0.25)
```

```
## [1] 12
```

Es gilt also: $\mathbb{P}(X = 12) \geq p$.

Wir können dies grafisch verdeutlichen:

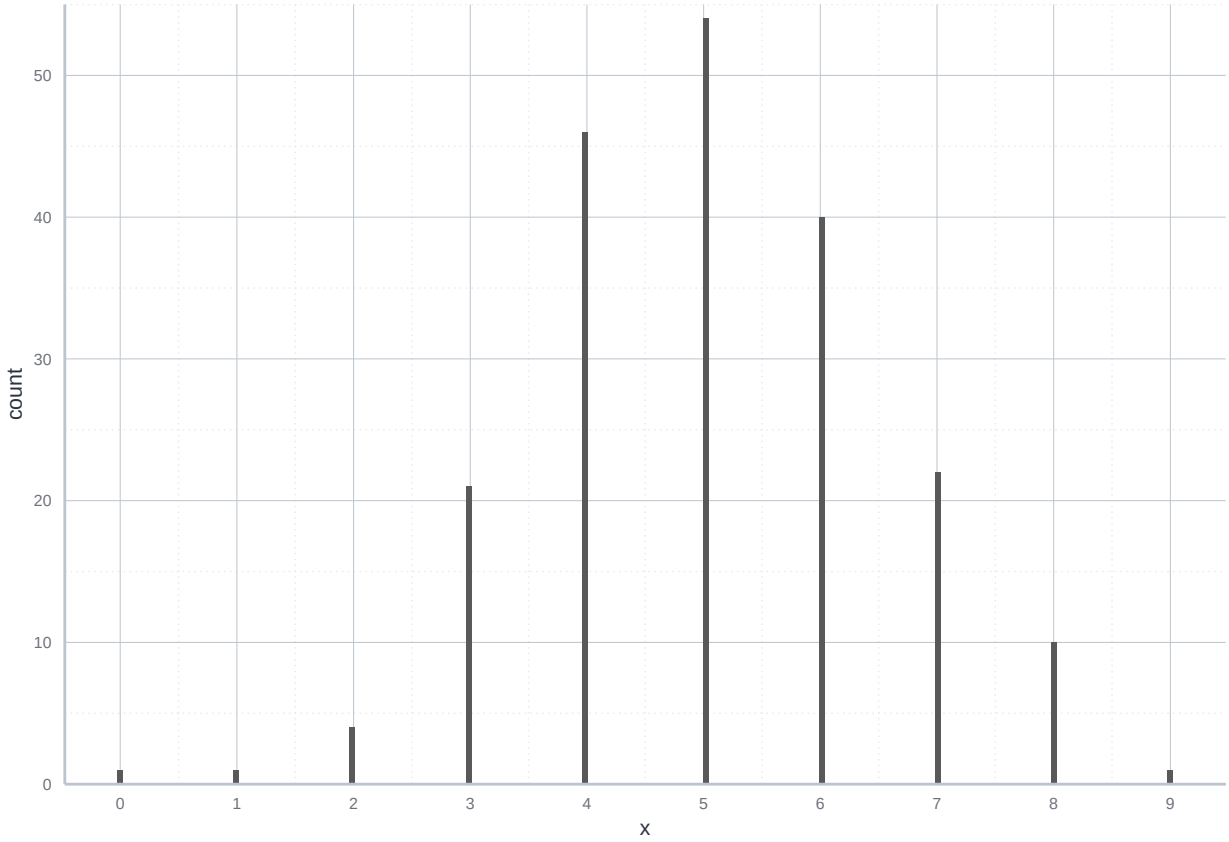


Möchten wir schließlich eine bestimmte Menge an **Realisierungen** aus einer Binomialverteilung ziehen geht das mit `rbinom()`, welches drei Argumente verlangt: `n` für die Anzahl der zu ziehenden Realisierungen, sowie `size` und `prob` als da Paramter n und p der Binomialverteilung:

```
sample_binom <- rbinom(n = 5, size = 10, prob = 0.4)
sample_binom
```

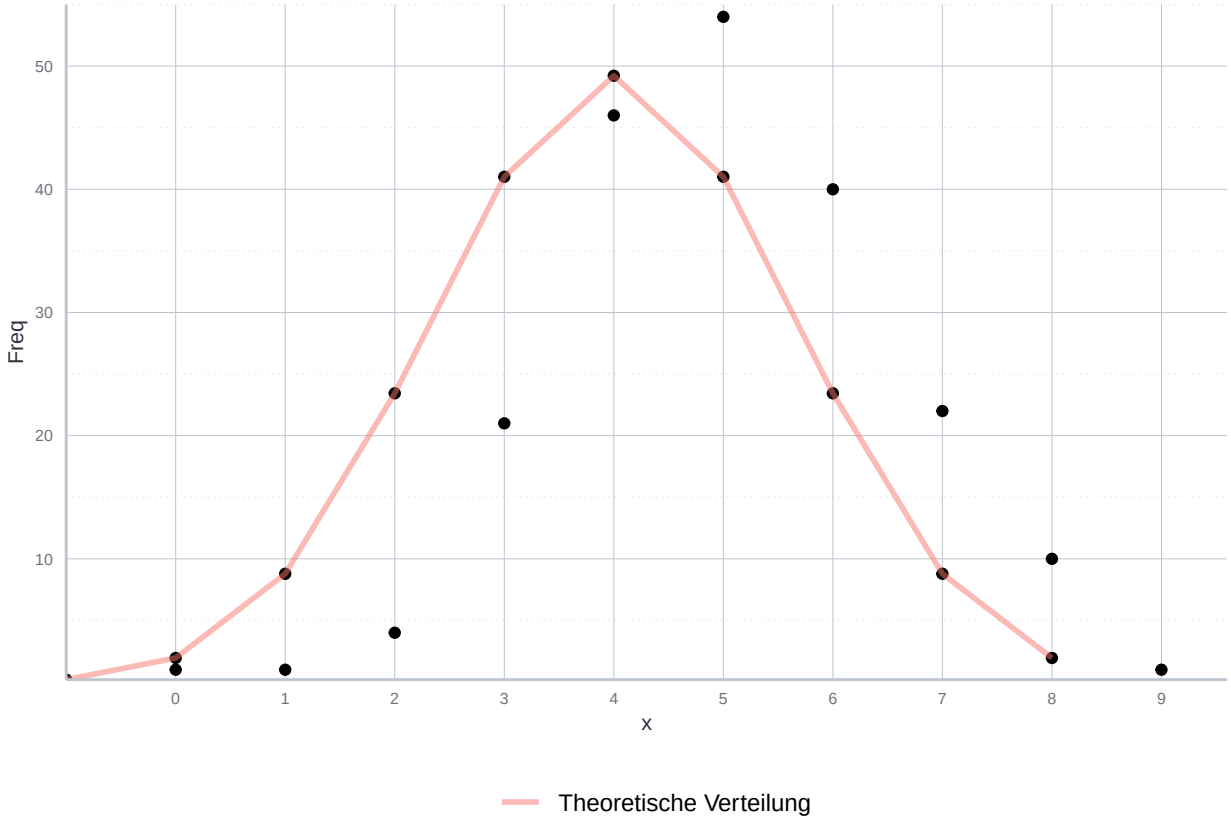
```
## [1] 7 1 4 3 4
```

> **Anwendungsbeispiel Binomialverteilung:** Unser Zufallsexperiment besteht aus dem zehnmaligen Werfen einer fairen Münze. Unter 'Erfolg' verstehen wir das Werfen von 'Zahl'. Nehmen wir an, wir führen das Zufallsexperiment 100 Mal durch, werfen also insgesamt 10 Mal die Münze und schreiben jeweils auf, wie häufig wir dabei einen Erfolg verbuchen konnten. Wenn wir unsere Ergebnisse aufmalen, indem wir auf der x-Achse die Anzahl der Erfolge, und auf der y-Achse die Anzahl der Experimente mit genau dieser Anzahl an Erfolgen aufmalen erhalten wir ein Histogram, das ungefähr so aussieht:



> Aus der Logik der Konstruktion des Zufallsexperiments und der Inspektion unserer Daten können wir schließen, dass die Binomialverteilung eine sinnvolle Beschreibung des Zufallsexperiments und der daraus entstandenen Stichprobe von 100 Münzwurfergebnissen ist. Da wir eine faire M+nze geworfen haben macht es Sinn für die Binomialverteilung $p = 0.5$ anzunehmen, und da wir in jedem einzelnen Experiment die Münze 10 Mal geworfen haben für $n = 10$. Wenn wir die mit $= 10$ und $p = 0.5$ parametrisierte theoretische Binomialverteilung nehmen und ihre theoretische Verteilungsfunktion über die Aufzeichnungen unserer Ergebnisse legen, können wir uns in dieser Vermutung bestärkt führen:

```
ggplot(data.frame(x=munzwurfe), aes(x=x)) +
  #geom_histogram(bins = wurzanzahl) +
  geom_point(data=data.frame(table(munzwurfe)),
             aes(x=munzwurfe, y=Freq)) +
  geom_point(data = data.frame(x=seq(0, max(munzwurfe), 1),
                               y=dbinom(seq(0, max(munzwurfe), 1), prob = p_zahl,
                                        size = wurfe_pro_experiment)*wurzanzahl),
             aes(x=x, y=y)
             ) +
    geom_line(data = data.frame(x=seq(0, max(munzwurfe), 1),
                               y=dbinom(seq(0, max(munzwurfe), 1), prob = p_zahl,
                                        size = wurfe_pro_experiment)*wurzanzahl),
             aes(x=x, y=y, color="Theoretische Verteilung"), alpha=0.5, lwd=1
             ) +
  scale_y_continuous(expand = expand_scale(c(0,0), c(0,1))) +
  #scale_color_manual(values=c("blue", "red"), name=c("Theoretische Verteilung", "Empirische Verteilung
  theme_icae() + theme(legend.position = "bottom")
```



B.3.4 Beispiel: die Poisson-Verteilung

Bei der Poisson-Verteilung handelt es sich um die Standardverteilung für unbeschränkte Zähldaten, also diskrete Daten, die kein natürliches Maximum haben.

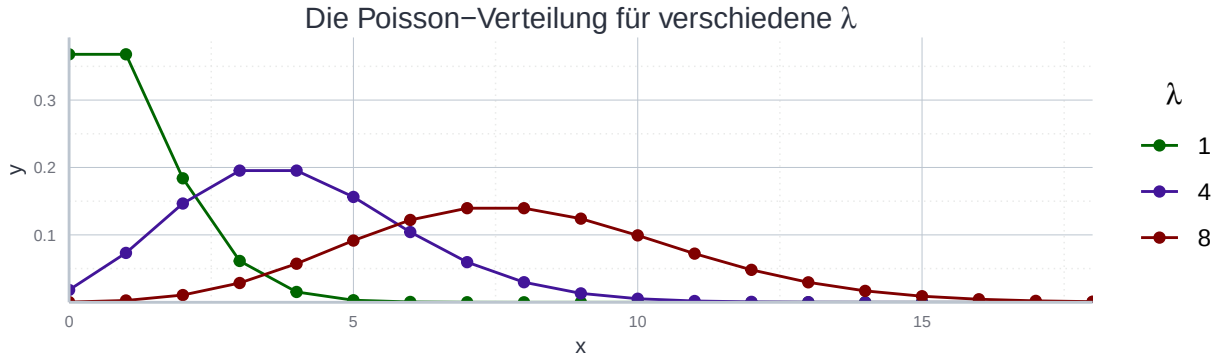
Bei der Poisson-Verteilung handelt es sich um eine **ein-parametrische** Funktion, deren einziger Parameter $\lambda > 0$ ist. λ wird häufig als die mittlere Ereignishäufigkeit interpretiert und ist **zugleich Erwartungswert als auch Varianz** der Verteilung: $\mathbb{E}(P_\lambda) = Var(P_\lambda) = \lambda$.

Ihre Definitionsmenge ist $\mathbb{N}$, also alle natürlichen Zahlen - daher ist sie im Gegensatz zur Binomialverteilung geeignet, wenn die Definitionsmenge der Verteilung keine natürliche Grenze hat.

Die **Wahrscheinlichkeitsfunktion** der Poisson-Verteilung hat die folgende Form:

$$P_\lambda(x) = \frac{\lambda^x}{x!} e^{-\lambda}$$

Die folgende Abbildung zeigt wie sich die Wahrscheinlichkeitsfunktion für unterschiedliche Werte von λ manifestiert:

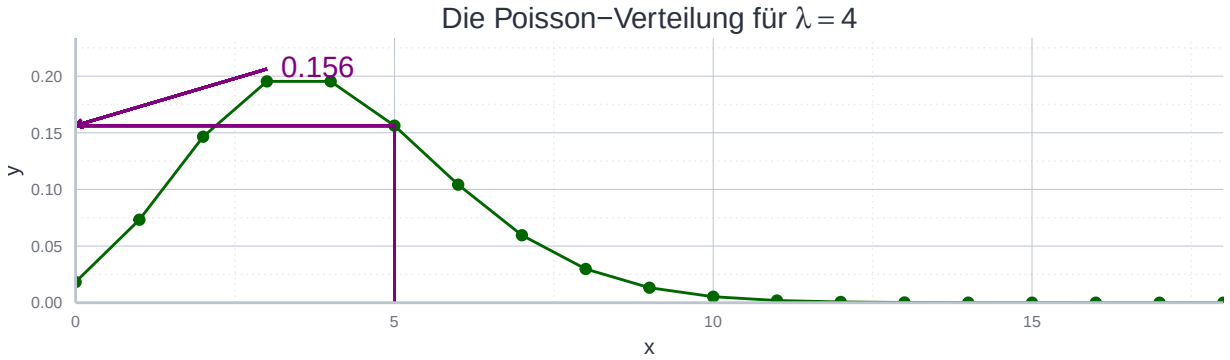


Wir können die Verteilung mit sehr ähnlichen Funktionen wie bei der Binomialverteilung analysieren. Nur die Parameter müssen entsprechend angepasst werden, da es bei der Poisson-Verteilung jetzt nur noch einen Paramter (`lambda`) gibt.

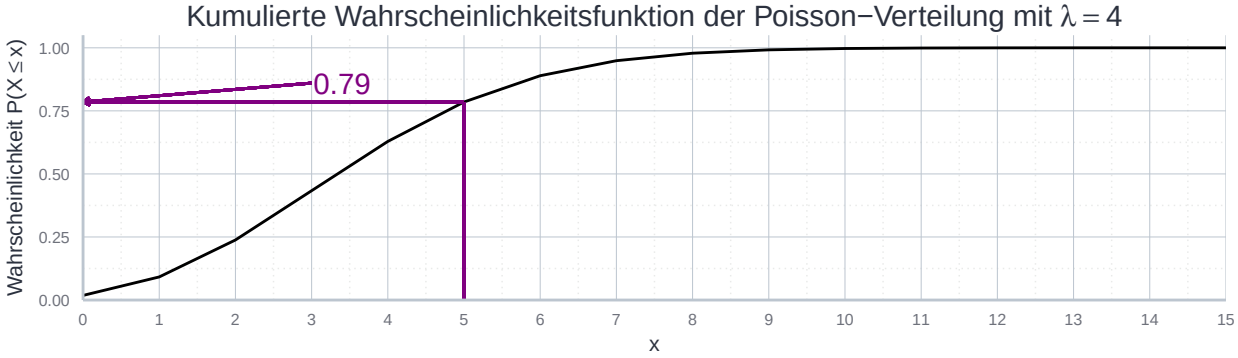
Möchten wir die Wahrscheinlichkeit bereichnen, genau x Erfolge zu beobachten, also $\mathbb{P}(X = x)$ geht das mit der Funktion `dpois()`. Das einzige notwendige Argument ist `lambda`:

```
dpois(5, lambda = 4)
```

```
## [1] 0.1562935
```



Informationen über die CDF erhalten wir über die Funktion `ppois()`, die zwei Argumente, `q` und `lambda`, annimmt.



Mit der Funktion `qpois()` finden wir für eine Wahrscheinlichkeit `p` den kleinsten Wert x, für den gilt, dass $\mathbb{P}(X = x) \geq p$.

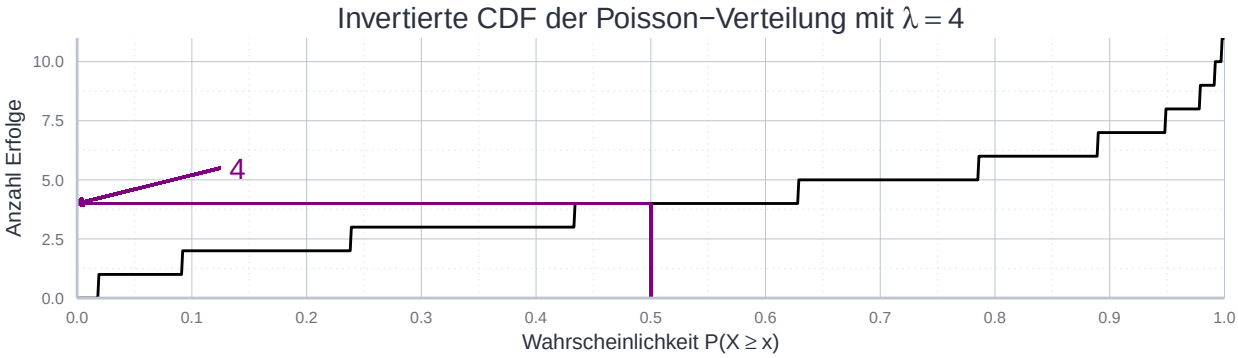
Wenn wir also wissen möchten wie viele Erfolge mit einer Wahrscheinlichkeit von 50% mindestens zu erwarten sind, dann schreiben wir:

```
qpois(p = 0.5, lambda = 4)
```

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

Es gilt also: $\mathbb{P}(X = 4) \geq 0.5$.

Wir können dies grafisch verdeutlichen:



Möchten wir schließlich eine bestimmte Menge an **Realisierungen** der ZV aus einer Poisson-Verteilung ziehen geht das mit `rpois()`, welches zwei notwendige Argumente annimmt: `n` für die Anzahl der Realisierungen und `lambda` für den Parameter λ:

```
pois_sample <- rpois(n = 5, lambda = 4)
pois_sample
```

```
## [1] 3 8 4 4 3
```

B.3.5 Hinweise zu diskreten Wahrscheinlichkeitsverteilungen

Wie Sie vielleicht bereits bemerkt haben sind die R Befehle für verschiedene Verteilungen alle gleich aufgebaut. Wenn `*` für die Abkürzung einer bestimmten Verteilung steht, können wir mit der Funktion `d*()` die Werte der Wahrscheinlichkeitsverteilung, mit `p*()` die Werte der kumulierten Wahrscheinlichkeitsverteilung und mit `q*()` die der Quantilsfunktion berechnen Mit `r*()` werden Realisierungen von Zufallszahlen realisiert. Für das Beispiel der Binomialverteilung, welcher die Abkürzung `binom` zugewiesen wurde, heißen die Funktionen entsprechend `dbinom()`, `pbinom()`, `qbinom()` und `rbinom()`.

Die folgende Tabelle gibt einen Überblick über gängige Abkürzungen und die Parameter der oben besprochenen diskreten Verteilungen.

Verteilung	Abkürzung	Parameter
Binomialverteilung	`binom`	`size`, `prob`
Poisson-Verteilung	`pois`	`lambda`

B.4 Stetige Wahrscheinlichkeitsmodelle

B.4.1 Stetige ZV

In vorangegangen Abschnitt haben wir uns mit diskreten Wahrscheinlichkeitsmodellen beschäftigt. Die diesen Modellen zugrundeliegenden ZV hatten einen abzählbaren Wertebereich. Häufig interessieren wir uns aber für ZV mit einem nicht abzählbaren Wertebereich, z.B. $\mathbb{R}$ oder $[0, 1]$.[6]

Bei stetigen Wahrscheinlichkeitsmodellen liegen zwischen zwei Punkten unendlich viele Punkte. Das hat bedeutende Implikationen für die Angabe von Wahrscheinlichkeiten. Im Gegensatz zu diskreten Wahrscheinlichkeitsmodellen hat demnach jeder einzelne Punkt im Wertebereich der ZV die Wahrscheinlichkeit 0:

$$\mathbb{P}(X = x_k) = 0 \quad \forall x_k \in W_X$$

wobei W_X für den Wertebereich von ZV X steht

Als Lösung werden Wahrscheinlichkeiten bei stetigen ZV nicht als Punktwahrscheinlichkeiten, sondern als Intervallwahrscheinlichkeiten angeben. Aus $\mathbb{P}(X = x)$ im diskreten Fall wird im stetigen Fall also:

$$\mathbb{P}(a < X \leq b), \quad a < b$$

Bei dieser Funktion sprechen wir von einer *kumulative Verteilungsfunktion* $F(x) = \mathbb{P}(X \leq x)$, wobei immer gilt:

$$\mathbb{P}(a < X \leq b) = F(b) - F(a)$$

Wann immer wir im diskreten Fall eine Wahrscheinlichkeitsfunktion verwendet haben um eine ZV zu beschreiben, verwenden wir im stetigen Fall die **Dichtefunktion** (*probability densitity function* - PDF) einer ZV. Hierbei handelt es sich um eine integrierbare und nicht-negative Funktion $f(x) \geq 0 \forall x \in \mathbb{R}$ mit $\int_{-\infty}^{\infty} f(x)dx = 1$ für die gilt:

$$\mathbb{P}([a, b]) = \int_a^b f(x)dx$$

Dementsprechend können wir den Ausdruck für die kumulative Verteilungsfunktion von oben ergänzen:

[6] Die Intervallschreibweise $[0, 1]$ ist potenziell verwirrent. Es gilt: $[a, b] = \{x \in \mathbb{R} | a \leq x \leq b\}$ (geschlossenes Intervall), $(a, b) = \{x \in \mathbb{R} | a < x < b\}$ (offenes Intervall), $(a, b) = \{x \in \mathbb{R} | a < x \leq b\}$(linksoffenes Intervall) und $(a, b) = \{x \in \mathbb{R} | a \leq x < b\}$(rechtsoffenes Intervall).

$$\mathbb{P}(a < X \leq b) = F(b) - F(a) = \int_a^b f(x)dx$$

Man sieht hier, dass die Dichtefunktion einer ZV die Ableitung ihrer kumulative Verteilungsfunktion ist. Wie oben beschrieben können wir die Werte an einzlnen Punkten nicht als *absolute* Wahrscheinlichkeiten interpretieren, da die Wahrscheinlichkeit für einzelne Punkte immer gleich 0 ist. Wir können aber die Werte der PDF an zwei oder mehr Punkten vergleichen um die *relative* Wahrscheinlichkeit der einzelnen Punkte zu bekommen.

Wie bei den diskreten ZV beschreiben wir eine ZV mit Hilfe von bestimmten Kennzahlen, wie dem **Erwartungswert**, der **Varianz** und den **Quantilen**. Diese sind quasi äquivalent zum diskreten Fall definiert, nur eben über Integrale (wir vergleichen alle folgenden Definitionen mit ihrem diskreten Pendant am Ende des Abschnitts). Für den Erwartungswert der ZV X gilt somit:

$$\mathbb{E}(X) = \int_{-\infty}^{\infty} xf(x)dx$$

Für die Varianz und die Standardabweichung entsprechend:

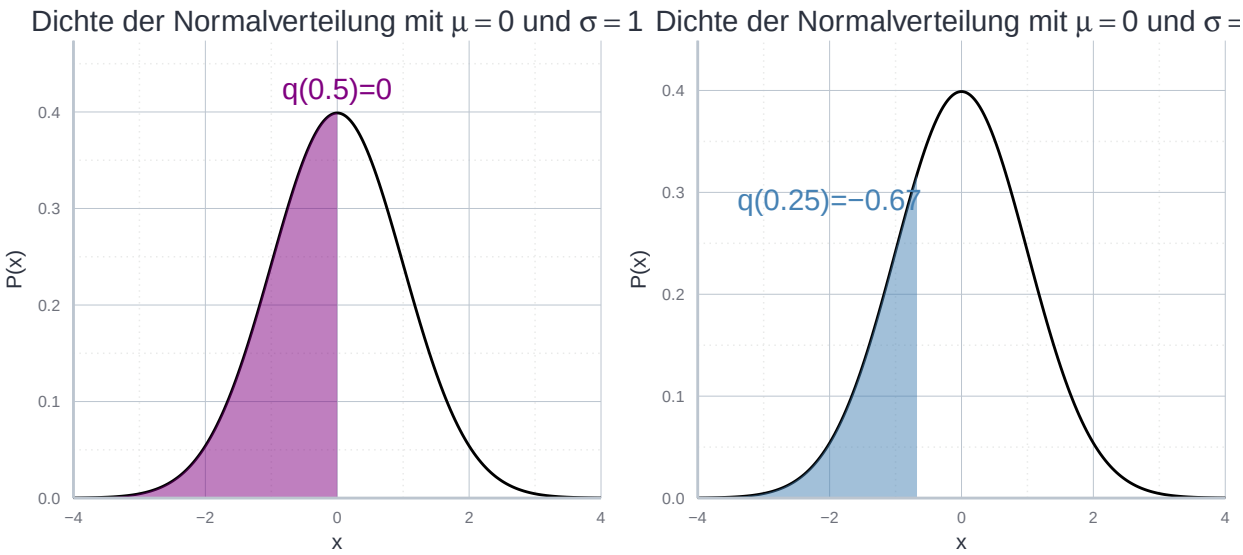
$$Var(X) = \mathbb{E}(X - \mathbb{E}(X))^2 = \int_{-\infty}^{\infty} (x - \mathbb{E}(X))^2 f(x)dx$$

$$\sigma_X = \sqrt{Var(X)}$$

Und, schlussendlich, gilt für das α-Quantil $q(\alpha)$:

$$\mathbb{P}(X \leq q(\alpha)) = \alpha$$

Im folgenden werden das 0.25 und 0.5-Quantil visuell dargestellt:



Abschließend wollen wir nun noch einmal die Definitionen der Kennzahlen und charakteristischer Verteilungen für den stetigen und diskreten Fall vergleichen:

Bezeichnung	Diskreter Fall	Stetiger Fall
Erwartungswert	$\mathbb{E}(x) = \sum_{x \in W_X} \mathbb{P}(X = x)x$	$\mathbb{E}(X) = \int_{-\infty}^{\infty} xf(x)dx$
Varianz	$Var(X) = \sum_{x \in W_X} [x - \mathbb{E}(X)]^2 \mathbb{P}(X = x)x$	$Var(X) = \mathbb{E}(X - \mathbb{E}(X))^2$
Standard-abweichung	$\sqrt{Var(X)}$	$\sqrt{Var(X)}$
α-Quantil	$\mathbb{P}(X \leq q(\alpha)) = \alpha$	$\mathbb{P}(X \leq q(\alpha)) = \alpha$

Bezeichnung	Diskreter Fall	Stetiger Fall
Dichtefunktion (PDF)	NA	$\mathbb{P}([a,b]) = \int_a^b f(x)dx$
Wahrsch's-funktion (PMF)	$p(x_k) = \mathbb{P}(X = x_k)$	NA
Kumulierte Verteilungsfunktion (CDF)	$\mathbb{P}(X \leq x)$	$F(x) = \mathbb{P}(X \leq x)$

Analog zum diskreten Fall wollen wir uns nun die am häufigsten vorkommenden stetigen Verteilungen noch einmal genauer anschauen.

B.4.2 Beispiel: die Uniformverteilung

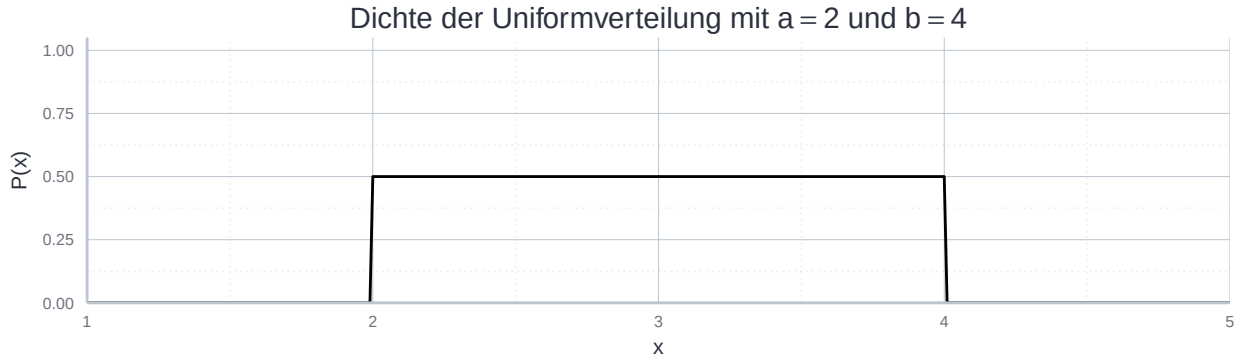
Die Uniformverteilung kann auch einem beliebigen Intervall $[a, b]$ mit $a < b$ definiert werden und ist dadurch gekennzeichnet, dass die Dichte über $[a, b]$ vollkommen konstant ist. Ihre einzigen Parameter sind die Grenzen des Intervalls, a und b.

Da bei stetigen Verteilungen die Dichte für aller Werte außerhalb des Wertebereichs per definitionem gleich Null ist, haben wir folgenden Ausdruck für die Dichte der Uniformverteilung:

$$f(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & \text{sonst } (x \notin W_X) \end{cases}$$

Auch der Erwartungswert ist dann intuitiv definiert, er liegt nämlich genau in der Mitte des Intervalls $[a, b]$. Er ist definiert als $\mathbb{E}(X) = \frac{a+b}{2}$ und ihre Varianz mit $Var(X) = \frac{(b-a)^2}{12}$ gegeben.

Ihre Dichtefunktion für $[a, b] = [2, 4]$ ist im folgenden dargestellt:



Die Abkürung in R für die Uniformverteilung ist `unif`. Endsprechend berechnen wir Werte für die Dichte mit `dunif()`, welches lediglich die Argumente `a` und `b` für die Grenzen des Intervalls benötigt:

```
dunif(seq(2, 3, 0.1), min = 0, max = 4)
```

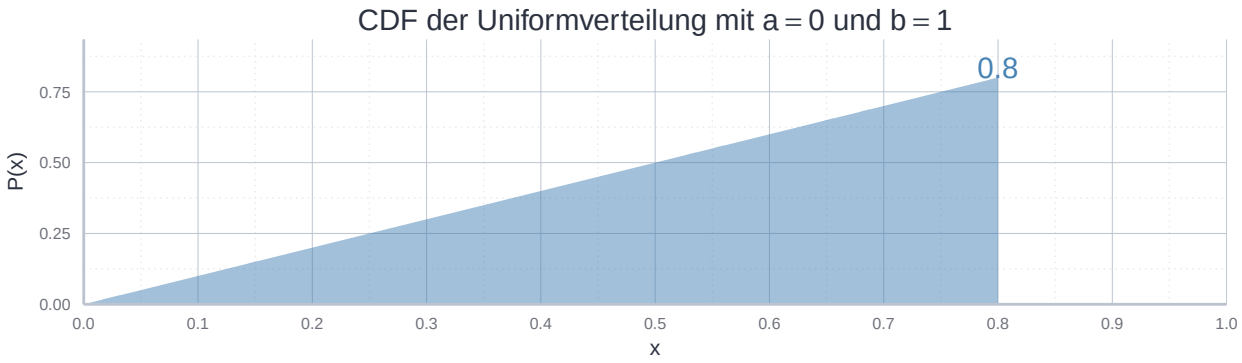
```
##  [1] 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25
```

Wie wir sehen erhalten wir hier immer den gleichen Wert $\frac{1}{b-a}$, was die zentrale Eigenschaft der Uniformverteilung ist. Hier wird auch deutlich, dass dieser Wert die *relative* Wahrscheinlichkeit angibt, da die absolute Wahrscheinlichkeit für jeden einzelnen Wert wie oben beschrieben bei stetigen ZV 0 ist.

Die CDF berechnen wir entsprechend mit `punif()`. Wenn $X \propto U(0, 4)$ erhalten wir $\mathbb{P}(X \leq 3)$ entprechend mit:

```
punif(0.8, min = 0, max = 4)
```

```
## [1] 0.2
```



Auch ansonsten können wir die Syntax der diskreten Verteilungen mehr oder weniger übernehmen: `qunif()` akzeptiert die gleichen Parameter wie `punif()` und gibt uns Werte der inversen CDF. `runif()` kann verwendet werden um Realisierungen einer uniform verteilten ZV zu generieren:

```
uniform_sample <- runif(5, min = 0, max = 4)
uniform_sample
```

```
## [1] 3.5209862 1.4563675 1.1529571 0.6825809 0.6886870
```

B.4.3 Beispiel: die Normalverteilung

Die wahrscheinlich bekannteste stetige Verteilung ist die Normalverteilung. Das liegt nicht nur daran, dass viele natürliche Phänomene als die Realisierung einer normalverteilten ZV modelliert werden können, sondern auch weil es sich mit der Normalverteilung in der Regel sehr einfach rechnen ist. Sie ist also häufig auch einfach eine bequeme Annahme.

Bei der Normalverteilung handelt es sich um eine **zwei-parametrige** Verteilung über den Wertebereich $W_X = \mathbb{R}$. Die beiden Parameter sind μ und σ^2, welche unmittelbar als Erwartungswert ($\mathbb{E}(X) = \mu$) und Varianz ($Var(X) = \sigma^2$) gelten. Wir schreiben $X \propto \mathcal{N}(\mu, \sigma^2)$ wenn für die PDF von X gilt:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

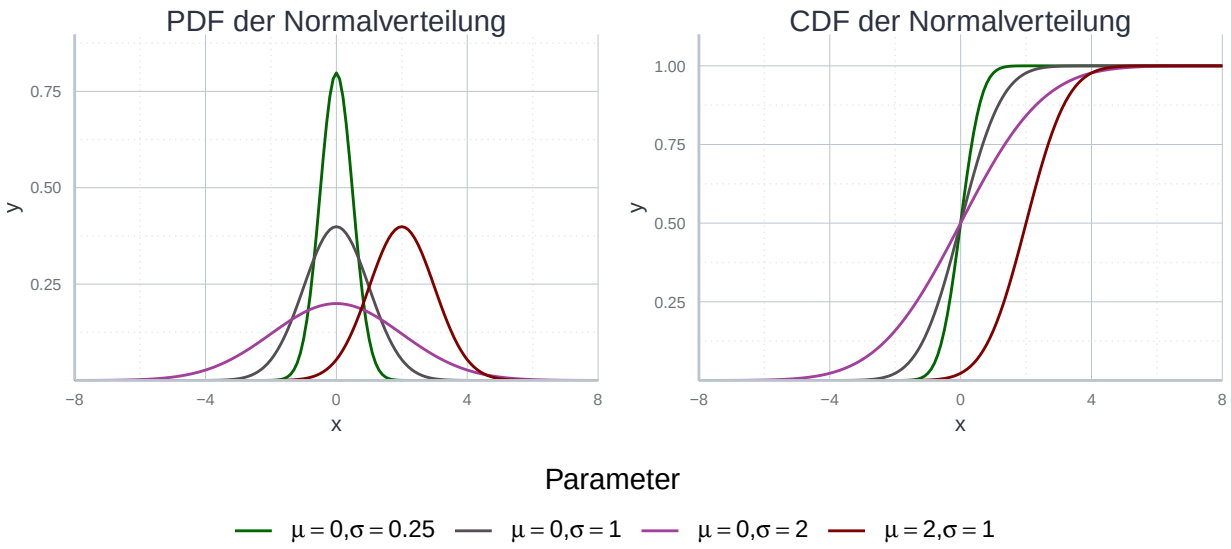
Unter der **Standard-Normalverteilung** verstehen wir eine Normalverteilung mit den Paramtern $\mu = 0$ und $\sigma = 1$.[7] Sie verfügt über die deutlich vereinfachte PDF:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$$

Die CDF der Normalverteilung ist analytisch nicht einfach darzustellen, die Werte können in R aber leicht über die Funktion `pnorm` (s.u.) abgerufen werden.

Im folgenden sind die PDF und CDF für exemplarische Parameterkombinationen dargestellt:

[7] Viele Tabellen mit bestimmten Kennzahlen der Normalverteilung beziehen sich auf die Standard-Normalverteilung. Wenn man diese Werte verwenden will, muss man die tatsächlich verwendete Stichprobe ggf. erst z-transformieren. Unter letzterem versteht man die *Normalisierung* einer ZV sodass sie den Erwartungswert 0 und die Varianz 1 besitzt. Dies geht i.d.R. für jede ZV X recht einfach über die Formel $Z = \frac{X-\mu}{\sigma}$, wobei Z die standartisierte ZV, μ den Erwartungswert und σ die Standardabweichung von X bezeichnet



Die Abkürzung in R ist `norm`. Alle Funktionen nehmen die Paramter μ und σ (nicht σ^2) über `mean` und `sd` als notwendige Argumente. Ansonsten ist die Verwendung äquivalent zu den vorherigen Beispielen:

```
dnorm(c(0.5, 0.75), mean = 1, sd = 2) # relative Wahrscheinlichkeiten über PDF
```

```
## [1] 0.1933341 0.1979188
```

```
pnorm(c(0.5, 0.75), mean = 1, sd = 2) # Werte der CDF
```

```
## [1] 0.4012937 0.4502618
```

```
qnorm(c(0.5, 0.75), mean = 1, sd = 2) # Werte der I-CDF
```

```
## [1] 1.00000 2.34898
```

```
norm_sample <- rnorm(5, mean = 1, sd = 2) # 5 Realisierungen der ZV
norm_sample
```

```
## [1]  0.9099446 -0.5698089 -2.3358839  0.2395470  2.8379932
```

Beispiel zum Zusammenhang `dnorm()` und `qnorm()`

B.4.4 Beispiel: die Exponentialverteilung

Sehr häufig wird uns auch die Exponentialverteilung begegnen. Außerhalb der Ökonomik wird sie v.a. zur Modellierung von Zerfallsprozessen oder Wartezeiten verwendet, in der Ökonomik spielt sie in der Wachstumstheorie eine zentrale Rolle. Es handelt sich bei der Exponentialverteilung um eine **ein-parametrige** Verteilung mit Parameter $\lambda \in \mathbb{R}^+$ und mit dem Wertebereich $W_X = [0, \infty]$.

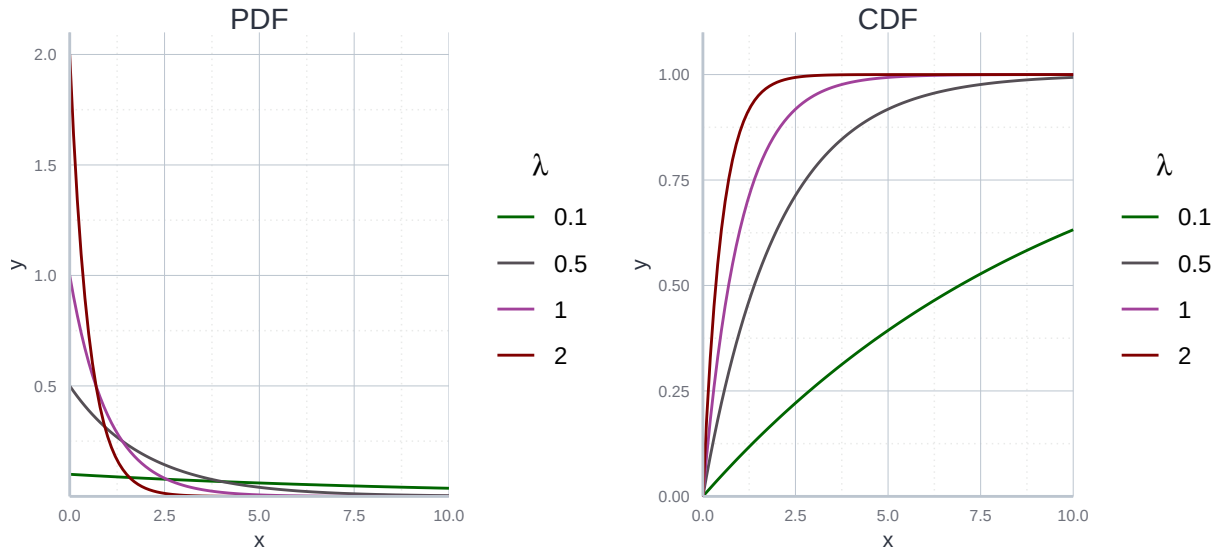
Die PDF der Exponentialverteilung ist:

$$f(x) = \begin{cases} 0 & x < 0 \\ \lambda e^{-\lambda x} & x \geq 0 \end{cases}$$

wobei e die Eulersche Zahl ist. Die CDF ist entsprechend:

$$F(x) = \begin{cases} 0 & x < 0 \\ 1 - e^{-\lambda x} & x \geq 0 \end{cases}$$

Beide Verteilungen sind im folgenden dargestellt:



Der Erwartungswert und die Varianz sind für die Exponentialverteilung äquivalent und hängen ausschließlich von λ ab: $\mathbb{E}(X) = \sigma_X = \frac{1}{\lambda}$.

Die Abkürzung in R ist `exp`. Alle Funktionen nehmen den Paramter λ über das Argument `rate` an:

```
dexp(c(0.5, 0.75), rate = 1) # relative Wahrscheinlichkeiten über PDF
```

```
## [1] 0.6065307 0.4723666
```

```
pexp(c(0.5, 0.75), rate = 1) # Werte der CDF
```

```
## [1] 0.3934693 0.5276334
```

```
qexp(c(0.5, 0.75), rate = 1) # Werte der I-CDF
```

```
## [1] 0.6931472 1.3862944
```

```
exp_sample <- rexp(5, rate = 1) # 5 Realisierungen der ZV
exp_sample
```

```
## [1] 0.8232605 0.4757590 3.4635949 1.2740277 1.0814852
```

Es gibt übrigens einen wichtigen Zusammenhang zwischen der stetigen Exponential- und der diskreten Poisson-Verteilung.

B.5 Zusammenfassung Wahrscheinlichkeitsmodelle

Die folgende Tabelle fasst noch einmal alle Wahscheinlichkeitsmodelle zusammen, die wir bislang betrachtet haben:

Verteilung	Art	Abkürzung	Parameter
Binomialverteilung	Diskret	`binom`	`size`, `prob`
Poisson-Verteilung	Diskret	`pois`	`lambda`
Uniform-Verteilung	Kontinuierlich	`punif`	`min`, `max`
Normalverteilung	Kontinuierlich	`norm`	`mean`, `sd`
Exponential-Verteilung	Kontinuierlich	`exp`	`rate`

In der statistischen Praxis sind das die Modelle, die wir verwenden, die DGP (*data generating processes*) zu beschreiben - also die Prozesse, welche die Daten, die wir in unserer Forschung verwenden, generiert haben.

Deswegen sprechen Statistiker*innen auch häufig von *Populationsmodellen*. Am besten stellt man es sich mit Hilfe der `r*()` Funktionen vor: man nimmt an, dass es einen DGP gibt, und unsere Daten der Output der `r*()`-Funktion zum Ziehen von Realisierungen sind. Mit dem Begriff des Populationsmodells macht man dabei deutlich, dass unsere Stichprobe nur eine Stichprobe darstellt - und nicht die gesamte Population aller möglichen Realisierungen des DGP.

Nun wird auch deutlich, warum Kenntnisse in der Wahrscheinlichkeitsrechnung so wichtig sind: wenn wir statistisch mit Daten arbeiten, dann versuchen wir in der Regel über die Daten Rückschlüsse auf den DGP zu schließen. Dafür müssen wir zunächst einmal eine grobe Struktur für den DGP annehmen, und dafür brauchen wir Kenntnisse in der Wahrscheinlichkeitsrechnung und für den entsprechenden Anwendungsfall konkrete Vorannahmen. Dann können wir, gegeben unsere Daten, unsere Beschreibung des DGP verfeinern.

Im Großteil dieses Kurses bedeutet das, dass wir für den DGP ein bestimmtes Wahrscheinlichkeitsmodell annehmen und dann auf Basis unserer Daten die Parameter für dieses Modell schätzen wollen. Dieses Vorgehen nennen wir *parametrisch*, weil wir hier vor allem Parameter schätzen wollen.[8]

[8] Die Alternative, *nicht-parametrische* Verfahren, nehmen kein konkretes Wahrscheinlichkeitsmodell an, sondern wählen das Modell auch auf Basis der Daten.

Appendix C

Wiederholung: Deskriptive Statistik

Bevor wir uns im nächsten Anhang mit dem Schluss von den Daten auf die Parameter des zugrundeliegenden Wahrscheinlichkeitsmodells beschäftigen, wollen wir uns im Folgenden noch mit Methoden der deskriptiven Statistik beschäftigen: denn zum einen setzt dieser Rückschluss der Daten auf das Populationsmodell voraus, dass wir uns überhaupt mit den Daten auseinandergesetzt haben, zum anderen sollte die Wahl des zugrundeliegenden Populationsmodell und der Art der Schätzung auf Basis der Daten erfolgen - und auch dafür benötigen wir Methoden der deskriptiven Statistik.

Die Methoden der deskriptiven Statistik helfen uns die Daten, die wir erhoben haben möglichst gut zu *beschreiben*. Die *deskriptive* Statistik grenzt sich von der *induktiven* Statistik davon ab, dass wir keine Aussagen über unseren Datensatz hinaus treffen wollen: wenn unser Datensatz also z.B. aus 1000 Schüler*innen besteht treffen wir mit den Methoden der deskriptiven Statistik nur Aussagen über genau diese 1000 Schüler*innen. Mit Methoden der *induktiven* Statistik würden wir versuchen Aussagen über Schüler*innen im Allgemeinen, zumindest über mehr als diese 1000 Schüler*innen zu treffen. Das ist genau der am Ende des vorherigen Anhangs angesprochene Schluss von den Daten auf den *data generating process* (DGP).

In diesem Abschnitt beschäftigen wir uns zunächst nur mit der deskriptiven Statistik. Das ist konsistent mit dem praktischen Vorgehen: bevor wir irgendwelche Methoden der induktiven Statistik anwenden müssen wir immer zunächst unsere Daten mit Hilfe deskriptiver Statistik besser verstehen.

Der Code in diesem Kapitel verwendet die folgenden Pakete:

```
library(here)
library(tidyverse)
library(data.table)
library(ggpubr)
library(latex2exp)
library(icaeDesign)
library(MASS)
```

Für die direkte Anwendung in R verwenden wir einen Datensatz zu ökonomischen Journalen:

```
journal_daten <- fread(here("data/tidy/journaldaten.csv"))
head(journal_daten)
```

```
##    Kuerzel                                              Titel
## 1:    APEL                    Asian-Pacific Economic Literature
## 2:  SAJoEH             South African Journal of Economic History
## 3:      CE                                 Computational Economics
## 4:  MEPiTE MOCT-MOST Economic Policy in Transitional Economics
## 5:    JoSE                             Journal of Socio-Economics
## 6:   LabEc                                       Labour Economics
##                   Verlag Society Preis Seitenanzahl Buchstaben_pS
## 1:              Blackwell      no   123          440          3822
## 2: So Afr ec history assn      no    20          309          1782
## 3:                 Kluwer      no   443          567          2924
## 4:                 Kluwer      no   276          520          3234
## 5:               Elsevier      no   295          791          3024
```

```
## 6:              Elsevier      no   344          609          2967
##    Zitationen Gruendung Abonnenten           Bereich
## 1:         21      1986         14           General
## 2:         22      1986         59  Economic History
## 3:         22      1987         17       Specialized
## 4:         22      1991          2      Area Studies
## 5:         24      1972         96 Interdisciplinary
## 6:         24      1994         15             Labor
```

Dieser Datensatz enthält Informationen über Preise, Seiten, Zitationen und Abonennten von 180 Journalen aus der Ökonomik im Jahr 2004.[1]

C.1 Kennzahlen zur Lage und Streuung der Daten

Die am häufigsten verwendeten Kennzahlen der deskriptiven Statistik sind das **arithmetische Mittel**, die **Standardabweichung** und die **Quantile**. Für die folgenden Illustrationen nehmen wir an, dass wir es mit einem Datensatz mit N kontinuiertlichen Beobachtungen $x_1, x_2, ..., x_n$ zu tun haben.

Das **arithmetische Mittel** ist ein klassisches Lagemaß und definiert als:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

In R wird das arithmetische Mittel mit der Funktion `mean()` berechnet:

```
avg_preis <- mean(journal_daten[["Preis"]])
avg_preis
```

```
## [1] 417.7222
```

Der durchschnittliche Preis der Journale ist also 417.7222222.

Die **Standardabweichung** ist dagegen ein Maß für die Streuung der Daten und wird als die Quadratwurzel der *Varianz* definiert:[2]

$$s_x = \sqrt{Var(x)} = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})^2}$$

Wir verwenden in R die Funktionen `var()` und `sd()` um Varianz und Standardabweichung zu berechnen:

```
preis_var <- var(journal_daten[["Preis"]])
preis_sd <- sd(journal_daten[["Preis"]])
cat(paste0(
  "Varianz: ", preis_var, "\n",
  "Standardabweichung: ", preis_sd
))
```

```
## Varianz: 148868.335816263
## Standardabweichung: 385.834596448094
```

Das α-**Quantil** eines Datensatzes ist der Wert, bei dem $\alpha \cdot 100\%$ der Datenwerte kleiner und $(1-\alpha) \cdot 100\%$ der Datenwerte kleiner sind. In R können wir Quantile einfach mit der Funktion `quantile()` berechnen. Diese Funktion akzeptiert als erstes Argument einen Vektor von Daten und als zweites Argument ein oder mehrere Werte für α:

```
quantile(journal_daten[["Preis"]], 0.5)
```

```
## 50%
## 282
```

[1] Bei den hier verwendeten Daten handelt es sich um eine Übersetzung des Datensatzes `Journals` aus dem Paket `AER` (Kleiber and Zeileis, 2008).

[2] Man beachte den im Vergleich zur Varianzformel für theoretische Modelle modifizierten Nenner $N-1$!

```
quantile(journal_daten[["Preis"]], c(0.25, 0.5, 0.75))
```

```
##    25%    50%    75%
## 134.50 282.00 540.75
```

Diese Werte können folgendermaßen interpretiert werden: 25% der Journale kosten weniger als 134.5 Dollar, 50% der Journale kosten weniger als 282 Dollar und 75% kosten weniger als 540.75 Dollar.

Dabei wird das 0.5-Quantil auch **Median** genannt. Wie beim Mittelwert handelt es sich hier um einen Lageparameter, der allerdings robuster gegenüber Extremwerten ist, da es sich nur auf die Reihung der Datenpunkte bezieht, nicht auf ihren numerischen Wert.[3]

Wie im Kapitel Erste Schritte in R für `mean()` und `sd()` erklärt, akzeptieren auch die Funktionen `mean()`, `var()`, `sd()` und `quantile()` das optionale Argument `na.rm`, mit dem fehlende Werte vor der Berechnung eliminiert werden können:

```
test_daten <- c(1:10, NA)
quantile(test_daten, 0.75)
```

```
## Error in quantile.default(test_daten, 0.75): missing values and NaN's not allowed if 'na.rm' is F
```

```
quantile(test_daten, 0.75, na.rm = T)
```

```
##  75%
## 7.75
```

Ein häufig verwendetes Steuungsmaß, das im Gegensatz zu Standardabweichung und Varianz robust gegen Ausreißer ist, ist die **Quartilsdifferenz**:

```
quantil_25 <- quantile(journal_daten[["Preis"]], 0.25, names = F)
quantil_75 <- quantile(journal_daten[["Preis"]], 0.75, names = F)
quant_differenz <- quantil_75 - quantil_25
quant_differenz
```

```
## [1] 406.25
```

Das optionale Argument `names=FALSE` unterdrückt die Benennung der Ergebnisse. Wenn wir das nicht machen würde, würde `quant_differenz` verwirrenderweise den Namen `75%` tragen.

C.2 Korrelationsmaße

Wie im Beispiel der Journale in diesem Kapitel erheben wir für einzelne Untersuchungsobjekte in der Regel mehr als eine Ausprägung. Im vorliegenden Falle haben wir das einzelne Journal z.B. Informationen unter anderem über Preis, Dicke und Zitationen. Häufig möchten wir wissen wie diese verschiedene Ausprägungen miteinender in Beziehung stehen. Zum Beispiel möchten wir wissen, ob dickere Journale tendenziell teurer sind. Neben der wichtigen grafischen Inspektion der Daten, zu der es ein eigenes Kapitel geben wird, gibt es dafür wichtige quantitative Maße, die häufig in den Bereich der Korrelationsmaße fallen.

Das einfachste Korrelationsmaß ist die empirische **Ko-Varianz**, die zwei stetige Ausprägungen x und y folgendermaßen definiert ist:

$$s_{xy} = \frac{1}{N-1} \sum_{n=1}^{N} (x_i - \bar{x})(y_i - \bar{y})$$

Wenn wir die empirische Kovarianz für den Bereich $[-1, 1]$ normieren erhalten wir die **empirische Korrelation** dieser Ausprägungen Handelt es sich bei den beiden Ausprägung um stetige Ausprägungen nennen wir das resultierende Maß den **Pearson-Korrelationskoeffizienten**:

$$\rho_{x,y} = \frac{s_{xy}}{s_x s_y}, \quad \rho \in [-1, 1]$$

wobei s_{xy} die Kovarianz der Ausprägungen x und y und s_x und s_y deren Standardabweichung bezeichnet.

[3] Wenn das teuerste Journal sich im Preis verdoppelt erhöht dies den Mittelwert beträchtlich, ändert den Median aber nicht.

Der so definierte Korrelationskoeffizient informiert uns über die Richtung und die Stärke des **linearen Zusammenhangs** zwischen x und y. Wenn $\rho_{x,y} > 0$ liegt ein positiver linearer Zusammenhang vor, d.h. größere Werte von x_i treten in der Tendenz mit größeren Werten von y_i auf. Hierbei gilt, dass $\rho_{x,y} = 1 \leftrightarrow y_i = a + bx_i$ für $a \in \mathbb{R}$ und $b > 0$ Umgekehrt gilt, dass wenn $\rho_{x,y} < 0$ ein negativer linearer Zusammenhang vorliegt und $\rho_{x,y} = -1 \leftrightarrow y_i = a + bx_i$ für $a \in \mathbb{R}$ und $b < 0$. Bei $\rho_{x,y} = 0$ liegt **kein linearer** Zusammenhang zwischen den Ausprägungen vor.

Wie wir unten sehen werden, enthält ρ keine Informationen über nicht-lineare Zusammenhänge zwischen x und y. Vorsicht bei der Interpretation ist also angebracht.

In unserem Datensatz haben wir z.B. Informationen über die Seitenzahl (Spalte `Seiten`) und den Preis von Journalen (Spalte `Preis`). Wir könnten uns nun fragen, ob dickere Journale tendenziell teurer sind. Dazu können wir, wenn wir uns nur für den linearen Zusammenhang interessieren, den Pearson-Korrelationskoeffizienten mit der Funktion `cor()` berechnen:

```
cor(journal_daten[["Preis"]], journal_daten[["Seitenanzahl"]],
    method = "pearson")
```

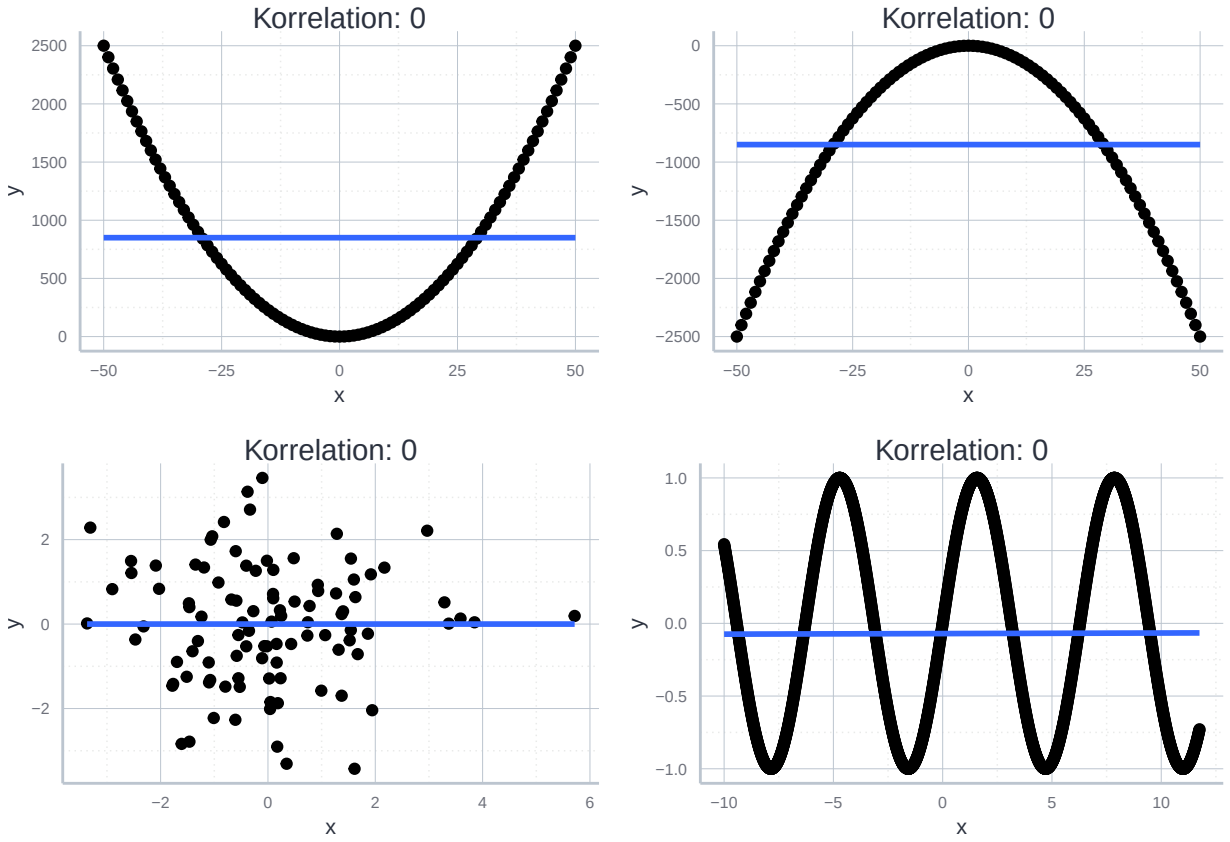
```
## [1] 0.4937243
```

Wir sehen also, dass es tatsächlich einen mittleren positiven linearen Zusammenhang zwischen Preis und Seitenzahl zu geben scheint.

Über das Argument `method` der Funktion `cor()` können auch andere Korrelationsmaße berechnet werden: der Spearman-Korrelationskoeffizient (`method='spearman'`) oder der Kendall-Korrelationskoeffizient (`method='kendall'`) sind beides Maße, die nur die Ränge der Ausprägungen und nicht deren numerische Werte berücksichtigen. Dies macht sie immun gegen Ausreißer und wir müssen keine Annahme über die Art der Korrelation machen wie beim Pearson-Korrelationskoeffizient, der nur lineare Zusammenhänge quantifiziert. Gleichzeitig gehen uns natürlich auch viele Informationen verloren. Das richtige Maß ist wie immer kontextabhängig und muss entsprechend theoretisch begründet werden.

Darüber hinaus erlaubt die Funktion `cor()` über das Argument `use` noch den Umgang mit fehlenden Werten genauer zu spezifizieren. Wenn Sie an der (nicht-standartisierten) Ko-Varianz interessiert sind, können Sie diese über die Funktion `cov()` berechnen, die analog zu `cor()` funktioniert.

In jedem Fall ist bei der Interpretation von Korrelationen Vorsicht angebracht: da der Korrelationskoeffizient nur die Stärke des *linearen* Zusammenhangs misst, können dem gleichen Korrelationskoeffizienten sehr unterschiedliche nicht-lineare Zusammenhänge zugrunde liegen:



Daher ist es immer wichtig die Daten auch visuell zu inspizieren. Datenvisualisierung ist aber so wichtig, dass sie in einem eigenen Kapitel behandelt wird.

C.3 Hinweise zur quantitativen und visuellen Datenbeschreibung

Wie das Beispiel der Korrelationsmaße gerade demonstriert hat, ist bei der Verwendung von quantitativen Maßen zur Beschreibung von Datensätzen immer große Vorsicht geboten. Sie sollten daher *immer* gemeinsam mit grafischen Darstellungsformen, wie Streudiagrammen oder Histogrammen verwendet werden.

Eine schöne Illustration ist Anscombe's Quartett (Anscombe, 1973). Dabei handelt es sich um vier Datensätze, die alle (fast exakt) gleiche deskriptive Statistiken aufweisen, jedoch offensichtlich sehr unterschiedlich sind. Diese offensichtlichen Unterschiede werden aber nur durch grafische Inspektion deutlich.

Der Datensatz ist in jeder R Installation vorhanden:

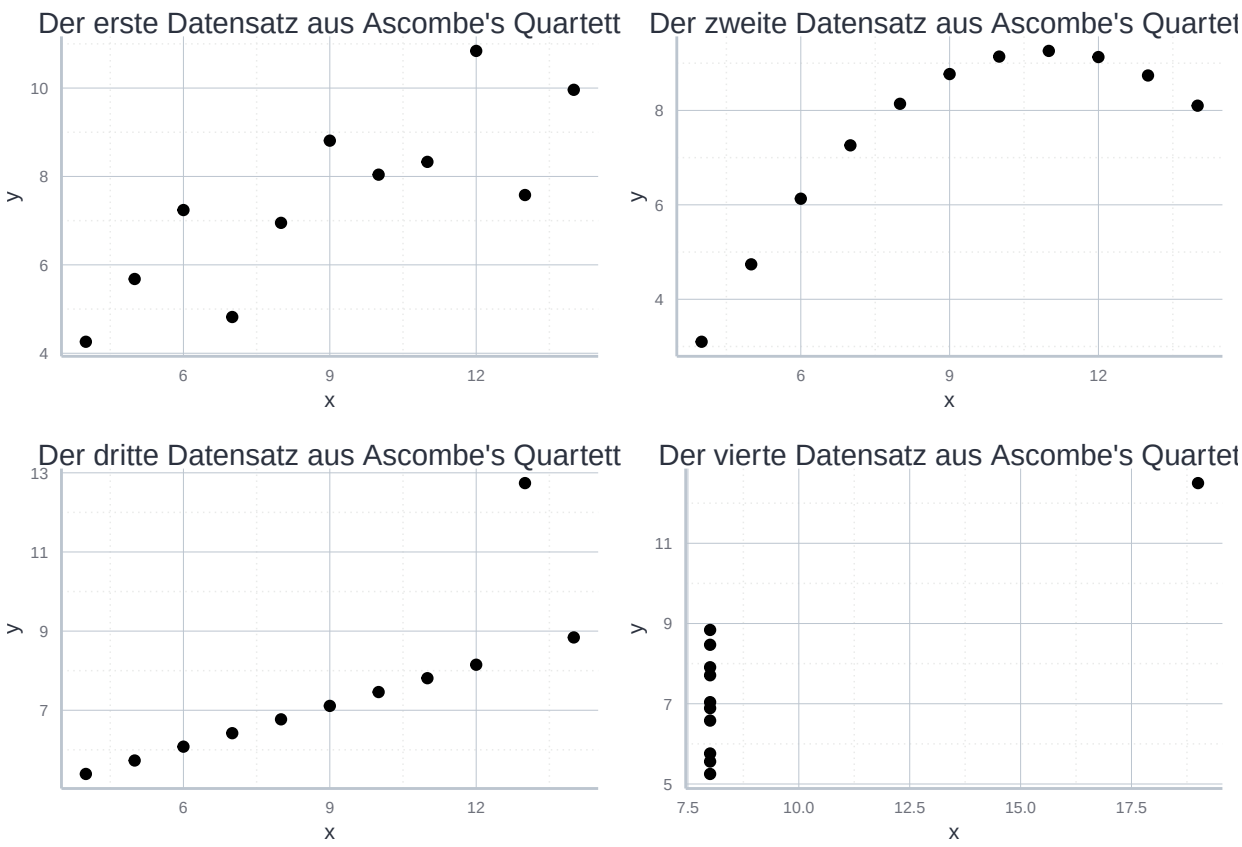
```
data("anscombe")
head(anscombe)
```

```
##   x1 x2 x3 x4   y1   y2    y3   y4
## 1 10 10 10  8 8.04 9.14  7.46 6.58
## 2  8  8  8  8 6.95 8.14  6.77 5.76
## 3 13 13 13  8 7.58 8.74 12.74 7.71
## 4  9  9  9  8 8.81 8.77  7.11 8.84
## 5 11 11 11  8 8.33 9.26  7.81 8.47
## 6 14 14 14  8 9.96 8.10  8.84 7.04
```

Die folgende Tabelle gibt die Werte der quantitativen Kennzahlen an:

Kennzahl	Wert
Mittelwert von x	9
Mittelwert von y	7.5
Varianz von x	11
Varianz von y	4.13
Korrelation zw. x und y	0.82

Die grafische Inspektion zeigt, wie unterschiedlich die Datensätze tatsächlich sind:



Interessanterweise ist bis heute nicht bekannt wie Anscombe (1973) seinen Datensatz erstellt hat. Für neuere Sammlungen von Datensätzen, die das gleiche Phänomen illustrieren siehe z.B. Chatterjee and Firat (2007) oder Matejka and Fitzmaurice (2017) . Eine sehr schöne Illustration der Idee findet sich auch auf dieser Homepage, die vom Autor von Matejka and Fitzmaurice (2017) gestaltet wurde.

C.4 Zusamenfassung

In der folgenden Tabelle wollen wir noch einmal die hier besprochenen Funktionen für den Themenbereich 'Deskriptive Statistik' zusammenfassen:

Maßzahl	Funktion	Beschreibung
Mittelwert	`mean()`	Wichtiges Lagemaß; arithmetisches Mittel der Daten
Varianz	`var()`	Maß für die Streuung; Einheit oft schwer interpretiertbar
Standardabweichung	`sd()`	Üblichstes Maß für die Streuung
α-Quantil	`quantile()`	$\alpha \cdot 100\%$ der Werte sind kleiner α
Median	`quantile(0.5)`	Robustes Lagemaß; die Hälfte der Daten sind größer/kleiner
Ko-Varianz (num. Daten)	`cov(method = 'pearson')`	Nicht-normierter linearer Zusammenhang
Ko-Varianz (Ränge)	`cov(method = 'kendall')`	Ko-Varianz der Ränge nach der Kendall-Methode
Ko-Varianz (Ränge)	`cov(method = 'spearman')`	Ko-Varianz der Ränge nach der Spearman-Methode
Pearson Korrelationskoeffizient	`cor(method = 'pearson')`	In $[-1, 1]$ normierter linearer Zusammenhang
Spearman-Korrelationskoeffizient	`cor(method = 'kendall')`	Korrelation der Ränge nach der Kendall-Methode
Kendall-Korrelationskoeffizient	`cor(method = 'spearman')`	Korrelation der Ränge nach der Spearman-Methode

Appendix D

Wiederholung: Drei Verfahren der schließenden Statistik

In diesem Kapitel werden wir drei zentrale Verfahren der schließenden Statistik wiederholen. Dabei schließen wir unmittelbar an die beiden vorangegangenen Kapitel zur Wahrscheinlichkeitstheorie und deskriptiven Statistik an: mit Hilfe der Wahrscheinlichkeitstheorie beschreiben wir mögliche Prozesse, die unsere Daten generiert haben könnten (DGP - *data generating processes*). Mit Hilfe der deskriptiven Statistik beschreiben wir unsere Daten und wählen auf dieser Basis Kandidaten für den DGP und sinnvolle Schätzverfahren aus. In der *schließenden Statistik* geht es nun genau um diese Schätzverfahren, die es uns erlauben von unseren Daten Rückschlüsse auf die DGP zu ziehen. Eine andere Art dies auszudrücken ist: mit Hilfe der schließenden Statistik wollen wir durch Analyse unserer Stichprobe auf die Gesamtpopulation, aus der die Stichprobe gezogen wurde schließen - und dabei möglichst die Unsicherheit, die diesem Schließprozess inhärent ist genau quantifizieren.

Natürlich ist wie immer Vorsicht geboten: wie bei der deskriptiven Statistik suggerieren viele der quantitativen Methoden der schließenden Statistik eine Genauigkeit und Exaktheit, die in der Wirklichkeit an der Korrektheit vieler Annahmen hängt. Man darf daher nicht den Fehler machen, die 'genauen' Ergebnisse der schließenden Statistik unhinterfragt zu glauben. Gleichzeitig darf man sie auch nicht verteufeln, denn viele Annahmen kann man mit ein wenig formalem Geschick und theoretischen Kenntnissen auch sinnvoll hinsichtlich ihrer Angemessenheit überprüfen.

Dafür ist es wichtig, die Grundlagen der schließenden Statistik gut verstanden zu haben. In diesem Kapitel wiederholen wir diese Grundlagen grob und kombinieren die Wiederholung mit einer Einführung in die entsprechenden Befehle in R.

Wie oben bereits angekündigt gehen wir in der Regel gehen wir davon aus, dass die von uns beobachteten Daten das Resultat eines gewissen Zufallsprozesses ist, den wir mit Hilfe der Wahrscheinlichkeitstheorie mathematisch beschreiben können. Da wir den DGP aber nicht direkt beobachten können, müssen wir auf Basis von empirischen Hinweisen und theoretischem Wissen entscheiden, welches Wahrscheinlichkeitsmodell wir unserer Analyse zugrunde legen. Sobald wir das getan haben, versuchen wir die Parameter, die für das von uns ausgewählte wahrscheinlichkeitstheoretische Modell relevant sind, so zu wählen, dass sie die Daten möglichst gut erklären können. Man nennt derlei Ansätze in der Statistik **parametrische Verfahren**, weil man mit den Daten die Parameter eines Modells bestimmen will, das man vorher selbst ausgewählt hat. Alternativ gibt es auch **nicht-parametrische Verfahren**: hier wird auch das Modell auf Basis der Daten bestimmt. Hier beschäftigen wir uns jedoch nur mit den parametrischen Verfahren.

In diesem Kontext sind drei Vorgehen in der statistischen Analyse besonders gängig:

1. **Punktschätzung:**
2. **Statistische Tests:**
3. **Konfidenzintervalle**

Wir wollen die verschiedenen Vorgehensweisen auf anhand eines Beispiels durchspielen: Nehnem wir an wir haben einen Datensatz und wir nehmen an, dass diese Daten von einer *Binominalverteilung* stammen.[1] Wir wissen, dass die Binominalverteilung durch zwei Parameter spezifiziert wird: n als die Anzahl der Versuche

[1] Wenn Sie nicht mehr wissen, was eine Binominalverteilung ist finden Sie im Anhang zur Wahrscheinlichkeitstheorie eine Erläuternug.

und p als die Erfolgswahrscheinlichkeit für den einzelnen Versuch. Wir sind nun daran interessiert auf Basis von unseren Daten Aussagen über den Paramter p der zugrundeliegenden Binominalverteilung zu treffen.[2]

Wenn wir einen konkreten Wert für p herausbekommen wollen müssen wir ein Verfahren der *Punktschätzung* wählen.
Wenn wir wissen wollen ob ein bestimmter Wert für p gegeben der Daten plausibel ist, dann sollten wir mit *statistischen Tests* (oder 'Hypothesentests') arbeiten. Wenn wir schließlich ein Intervall für p spezifizieren wollen, das mit den Beobachtungen kompatibel ist, dann suchen wir nach einem *Konfidenzintervall* für p.

Im folgenden werden die drei Verfahren in größerem Detail besprochen. Der Code in diesem Kapitel verwendet dabei die folgenden Pakete:

```
library(here)
library(tidyverse)
library(ggpubr)
library(latex2exp)
library(icaeDesign)
library(AER)
library(MASS)
```

D.1 Punktschätzung

Bei der Punktschätzung geht es darum auf Basis der Daten konkrete Werte für die Parameter der den Daten zugrundeliegenden Verteilung zu schätzen. In der Regel bezeichnet man den Parameter, den man schätzen möchte, mit dem Symbol θ. Der Grund ist Faulheit und bessere Lesbarkeit: man kann dann nämlich die selbe Notation verwenden, egal welche zugrundeliegende Verteilung man vorher ausgewählt hat.

Im vorliegenden Fall wollen wir also einen konkreten Wert für θ auf Basis der Daten schätzen. Dabei ist ganz wichtig zu beachten, dass wir den wahren Wert von θ in der Regel nicht kennen und auch nie genau kennen lernen werden.

Um zwischen dem wahren Wert von θ und dem Schätzer für θ in unserer Notation unterscheiden zu können, verwenden wir das $\hat{}$-Symbol. Entsprechend bezeichnet $\hat{\theta}$ einen **Schätzer** für θ.

Ein Schätzer ist dabei eine Funktion, die als Input unsere Daten nimmt, und als Output einen Wert ausgibt, der eine möglichst gute Schätzung für θ darstellt. Entsprechend können wir für eine Stichprobe vom Umfang n schreiben:

$$\hat{\theta} : \mathbb{R}^n \to \mathbb{R}, \quad \hat{\theta} = \hat{\theta}(x_1, ..., x_n)$$

Damit ist auch klar, dass es sich bei einem Schätzer um eine Zufallsvariable (ZV) handelt: Funktionen von ZV sind selbst ZV und unsere Daten $x_1, ..., x_n$ interpretieren wir ja als Realisierungen von ZV $X_1, ..., X_n$. Der unbekannt wahre Wert θ ist dagegen keine ZV.

> **Hinweis: Schätzer vs. geschätzter Wert** Die Unterscheidung zwischen einem Schätzer (*estimator*) und einem geschätzten Wert (*estimate*) ist in der Statistik zentral: der Schätzer beschreibt die Prozedur einen geschätzten Wert zu bekommen. Er nimmt in der Regel die Form einer Formel oder eines Algorithmus an. Der *geschätzte Wert* ist für einen konkreten Anwendungsfall der Wert, den der Schätzer liefert.

Die Konstruktion von Schätzern ist keine einfache Aufgabe. Wir lernen im Laufe der Veranstaltungen verschiedene Methoden, wie die *Momentenmethode* und die *Maximum-Likelihood Methode* genauer kennen.

D.2 Hypothesentests

Wir verwenden statistische Tests um Fragen der folgenden Art zu beantworten: gegeben der Daten die wir sehen und der Annahmen, die wir treffen, ist ein bestimmter Wert für Parameter θ plausibel?

[2] Die Annahme, dass die Daten *überhaupt* von einer Binominalverteilung stammen wird hier nicht in Frage gestellt! Das ist genau die Vor-Annahme, die wir bei parametrischen Verfahren treffen müssen.

Beispiel: Das klassische Beispiel ist die Frage, ob eine Münze manipuliert wurde oder nicht. Wenn wir beim Ereignis 'Zahl' von Erfolg sprechen, dann können wir n Münzwürfe als Binomialverteilung mit $B(n, p)$ modellieren. Bei einer nicht manipulierten Münze wäre $p = 0.5$: die Wahrscheinlichkeit, dass wir das Ereignis 'Zahl' erleben liegt beim einzelnen Wurf bei 50%. Nennen wir das unsere Ausgangs-, oder *Nullhypothese*. Zur Überprüfung dieser Hypothese werfen wir die Münze nun 100 mal. Nehmen wir nun an, dass wir das Ereignis 'Zahl' in 60 von 100 Würfen beobachten. Bedeutet das, dass unsere Nullhypothese von $p = 0.5$ plausibel ist? Um diese Frage zu beantworten fragen wir uns, wie wahrscheinlich es bei $p = 0.5$ wäre, tatsächlich 60 mal Zahl zu beobachten. Diese Wahrscheinlichkeit können wir berechnen, aus Tabellen auslesen oder von R bestimmen lassen (die genaue Verwendung der Funktion `binom.test()` wird unten genauer besprochen):

```
b_test_object <- binom.test(x = 60, n = 100, p = 0.5)
b_test_object[["p.value"]]
```

```
## [1] 0.05688793
```

Die Wahrscheinlichkeit liegt also bei 5.7 %. Dies ist der so genannte p-Wert. In der Regel lehnt man eine Hypothese ab, wenn $p < 0.1$ oder $p < 0.05$. Im vorliegenden Falle ist unsere Hypothese einer fairen Münze aber kompatibel mit der Beobachtung von 60 mal Zahl.

Wir wollen nun das Vorgehen aus dem Beispiel generalisieren und das standardmäßige Vorgehen bei einem statistischen Test zusammenfassen:[3]

1. Schritt: Aufstellen eines wahrscheinlichkeitstheoretischen Modells Zunächst müssen wir eine Annahme über den Prozess treffen, welcher der Generierung unserer Daten zugrunde liegt. Im Beispiel oben haben wir eine Binomialverteilung $\mathcal{B}(n, p)$ angenommen. Diese Entscheidung muss auf Basis von theoretischen und empirischen Überlegungen getroffen werden. Für diskrete Daten macht es z.B. keinen Sinn eine stetige Verteilung anzunehmen und umgekehrt.

2. Schritt: Formulierung der Nullhypothese Die Hypothese, die wir mit unseren Daten testen wollen wird **Nullhypothese** genannt. Wir wollen also immer fragen, ob H_0 gegeben der Daten plausibel ist. Die Formulierung von H_0 wird also durch unser Erkenntnisinteresse bestimmt. In der Regel formulieren wir eine Hypothese, die wir verwerfen wollen als H_0.[4] Wenn wir also die Hypothese bezüglich eines Parameters θ testen wollen, dass $\beta \neq 0$, dann formulieren wir $H_0 : \theta = 0$. Anders formuliert: wir möchten andere mit den Daten überzeugen, dass H_0 falsch ist.

Aus der Nullhypothese und unserem Erkenntnisinteresse ergibt sich die **Alternativhypothese** H_1. Sie umfasst alle interessierenden Ereignsse, die H_0 widersprechen. Je nach dem wie wir H_1 formulieren unterscheiden wir folgende Arten von Hypothesentests:

$H_0 : \theta = 0$ und $H_1 : \theta \neq 0$: hier sprechen wir von einem **zwei-seitigen Test**, denn wir machen keine Aussage darüber ob die Alternative zu H_0 entweder in $\theta > 0$ oder $\theta < 0$ liegt. Gemeinsam decken H_0 und H_1 hier alle möglichen Ereignisse ab.

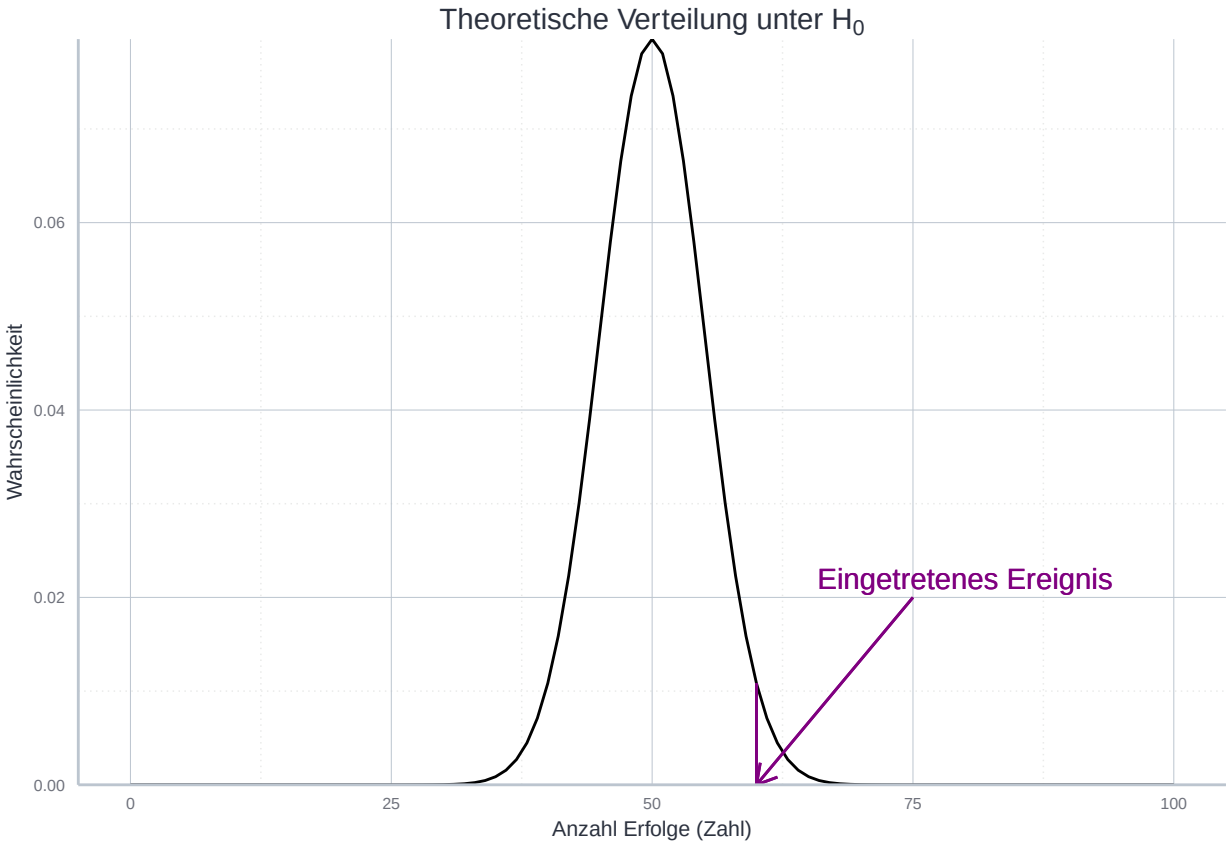
$H_0 : \theta = 0$ und $H_1 : \theta > 0$: Hier sprechen wir von einem **einseitigen Test nach oben**. Wir fragen uns hier nur ob θ größer ist als 0. Der Fall, dass $\theta < 0$, wird nicht beachtet. Natürlich können wir den einseitigen Test auch andersherum formulieren als $H_0 : \theta = 0$ und $H_1 : \theta < 0$. Dann sprechen wir von einem **einseitigen Test nach unten**.

Beispiel: Wenn wir unser Münzbeispiel von oben betrachten können wir die drei verschiedenen Testarten folgendermaßen konkretisieren: beim *zweiseitigen Test* wäre $H_0 : p = 0.5$ und $H_1 : p \neq 0.5$ und wir würden ganz allgemein fragen ob die Münze manipuliert ist. Beim **einseitigen Test nach oben** würden wir $H_0 : p = 0.5$ und $H_1 : p > 0.5$ testen und damit fragen ob die Münze *zugunsten von Zahl* manipuliert wurde. Wir lassen dabei die Möglichkeit, dass die Münze zugunsten von Kopf manipuliert wurde völlig außen vor. Beim **einseitigen Test nach unten** wäre es genau umgekehrt: $H_0 : p = 0.5$ und $H_1 : p < 0.5$. KONKRETE BERECHNUNGEN FÜR DEN P WERT

[3] Wir beschränken uns hier auf so genannte *parametrische* Tests. Das bedeutet, dass wir zunächst ein bestimmtes Modell für den Datengenerierungsprozess annehmen. Im Beispiel war dieses Modell die Binomialverteilung. Es gibt auch Tests, die ohne eine solche Annahme auskommen. Sie werden *nicht-parametrisch* genannt und später in dem Kurs besprochen.

[4] An machen Stellen der sozial- und wirtschaftswissenschaftlichen Literatur wird anstelle von "verwerfen" auch das Wort "falsifizieren" benutzt um die Zurückweisung der Null-Hypothese zu umschreiben. Diese Wortwahl ist allerdings irreführend, da hier nicht Aussagen aus einer Theorie widerlegt werden, die einen gewissen Zusammenhang behaupten. Im Gegensatz wird die Hypothese zurückgewiesen, dass der vermutete Zusammenhang eben nicht besteht - die zu Grunde gelegte Theorie wird also durch die Zurückweisung der Null-Hypothese im Normalfall nicht widerlegt sondern vielmehr bestätigt.

3. Schritt: Berechnung einer Teststatistik Wir überlegen nun welche Verteilung unserer Daten wir erwarten würden *wenn die Nullhypothese korrekt wäre.* Wenn wir im ersten Schritt also eine Binomialverteilung mit $n = 100$ angenommen haben und $H_0 : p = 0.5$, dann würden wir vermuten, dass unsere Daten gemäß $B(n, 0.5)$ verteilt sind.[5] Diese theoretische Verteilung können wir dann mit den tatsächlichen Daten vergleichen und fragen, wie wahrscheinlich es ist diese Daten tatsächlich so beobachten zu können wenn H_0 wahr wäre:



4. Schritt: Festlegung des Signifikanzniveaus: Wir müssen nun festlegen welches Risiko wir bereit sind einzugehen, unsere Nullhypothese H_0 zu verwerfen, obwohl sie eigentlich richtig ist. Die maximale Wahrscheinlichkeit für dieses unglückliche Ereigns bezeichnen wir mit α uns sie bestimmt unser Signifikanzniveau. Typischweise nimmt man als Standardwert $\alpha = 0.05$, d.h. wir konstruieren unsere Test so, dass die Wahrscheinlichkeit, dass wir H_0 fälschlicherweise verwerfen maximal $\alpha = 0.05$ beträgt. Mit anderen Worten, wir legen hier die Wahrscheinlichkeit für einen **Fehler 1. Art** explizit fest.[6]

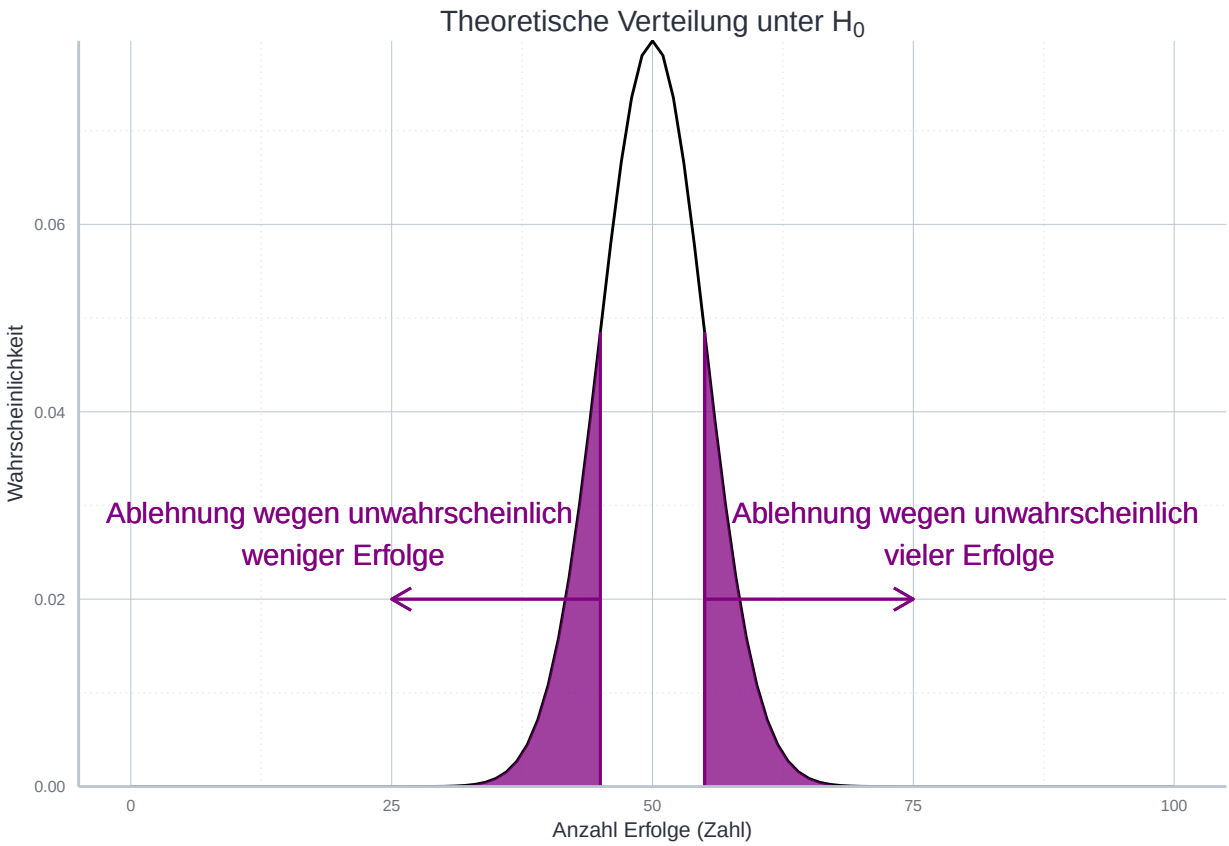
Aus dem gewählten Signifikanzniveau ergibt sich dann der **Verwerfungsbereich** für unsere Nullhypothese. Wenn unsere beobachteten Daten im Verwerfungsbereich liegen wollen wir H_0 als verworfen betrachten.[7] Es ergibt sich logisch aus dem vorher gesagten, dass ein höheres α mit einem größeren Verwerfungsbereich einhergeht.

Der Verwerfungsbereich für das oben darstellte Beispiel mit $H_0 : \theta = 0$ und $H_1 : \theta \neq 0$ ergibt sich für $\alpha = 0.05$ also folgendermaßen:

[5] In der Praxis wird die Berechnung der Teststatistik durch eine R Funktion in einem der nächsten Schritte übernommen, aber es macht Sinn, sich das grundsätzliche Vorgehen dennoch in dieser Sequenz bewusst zu machen.

[6] Wir sprechen von einem *Fehler 1. Art* wenn wir auf Basis eines Tests H_0 verwerfen obwohl sie eigentlich richtig ist. Von einem *Fehler 2. Art* sprechen wir, wenn wir H_0 nicht verwerfen, obwohl H_0 eigentlich falsch ist.

[7] Ganz im Sinne von Popper können mit Hilfe von statistischen Tests alle Hypothesen immer nur verworfen werden. Verifizieren können wir nichts!



5. Schritt: Die Entscheidung Wenn sich die beobachtbaren Daten im Verwerfungsbereich befinden wollen wir H_0 verwerfen und die Nullhypothese entsprechend als verworfen ansehen. Falls nicht kann die Nullhypothese nicht verworfen werden - was aber nicht bedeutet, dass sie *verifiziert* wurde. Letzteres ist mit statistischen Tests nicht möglich.

In `R` werden die gerade besprochenen Tests in der Regel in einer Funktion zusammengefasst. Die Wahl der Funktion wird dabei von der im ersten Schritt angenommenen Verteilung bestimmt. Im Falle der Binomialverteilung verwenden wir die Funktion `binom.test()`, welche eine Liste mit relevanten Informationen über den Test erstellt. Es macht Sinn, dieser Liste einen Namen zuzuweisen und dann die relevanten Informationen explizit abzurufen:

```
b_test_object <- binom.test(x = 60, n = 100, p = 0.5, alternative = "two.sided")
typeof(b_test_object)
```

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

Bevor wir uns mit dem Ergebnis befassen wollen wir uns die notwendigen Argumente von `binom.test()` genauer anschauen (eine gute Erläuterung liefert wie immer `help(binom.test)`).

Über das Argument `x` informieren wir R über die tatsächlich beobachtete Anzahl von Erfolgen (in unserem Fall hier 60). Das Argument `n` spezifiziert die Anzahl der Beobachtungen. Mit `p` geben wir den unter H_0 angenommenen Wert für die Erfolgswahrscheinlichkeit an. Mit dem Argument `alternative` informieren wir R schließlich darüber ob wir einen zweiseitigen (`alternative = "two.sided"`), einen einseitigen Test nach oben (`alternative = "greater"`) oder einen einseitigen Test nach unten (`alternative = "less"`) durchführen wollen.

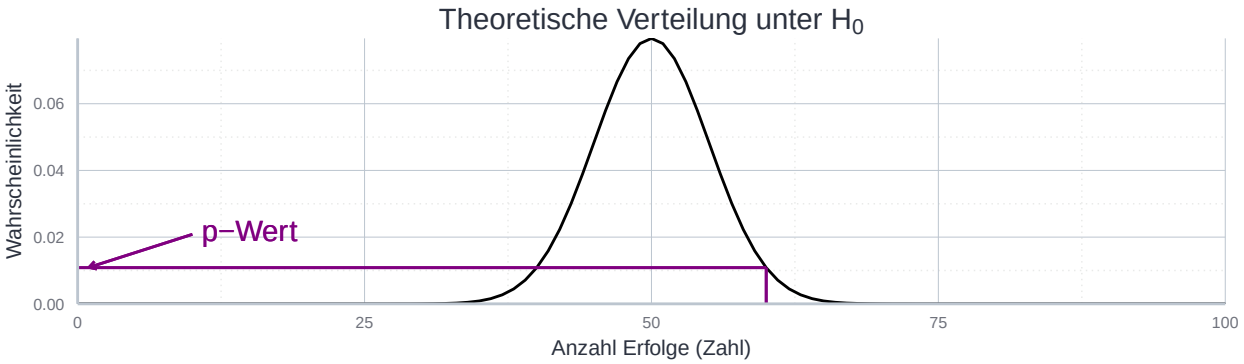
Wenn wir einen Überblick über die Ergebnisse bekommen wollen können wir das Objekt direkt aufrufen. Die Liste wurde innerhalb der Funktion `binom.test` so modifiziert, dass uns die Zusammenfassung visuell ansprechend aufbereitet angezeigt wird:

```
b_test_object
```

```
##
##  Exact binomial test
##
## data:  60 and 100
## number of successes = 60, number of trials = 100, p-value =
```

```
## 0.05689
## alternative hypothesis: true probability of success is not equal to 0.5
## 95 percent confidence interval:
##  0.4972092 0.6967052
## sample estimates:
## probability of success
##                    0.6
```

Die Überschrift macht deutlich was für ein Test durchgeführt wurde und die ersten beiden Zeilen fassen noch einmal die Daten zusammen. In der zweiten Zeile findet sich zudem der **p-Wert**. Der p-Wert gibt die Wahrscheinlichkeit an, mit der die beobacheten Daten unter H_0 tatsächlich beobachtet werden können. Wir können den p-Wert aus der theoretischen Verteilung von oben auf der y-Achse ablesen, wenn wir den beobachteten Wert auf der x-Achse suchen:



Die nächste Zeile formuliert dann die Alternativhypothese aus (und hängt entsprechend vom Argument `alternative` ab). Die Zeilen danach geben das 95%-Intervall an (mehr dazu im nächsten Abschnitt) und den Punktschätzer für den zu testenden Parameter (siehe vorheriger Abschnitt).

Wenn wir wissen wollen welche Informationen die so erstellte Liste sonst noch für uns bereit hält, bzw. wie wir diese Informationen direkt ausgeben lassen können, sollten wir uns die Struktur der Liste genauer ansehen:

```
str(b_test_object)
```

```
## List of 9
##  $ statistic  : Named num 60
##   ..- attr(*, "names")= chr "number of successes"
##  $ parameter  : Named num 100
##   ..- attr(*, "names")= chr "number of trials"
##  $ p.value    : num 0.0569
##  $ conf.int   : num [1:2] 0.497 0.697
##   ..- attr(*, "conf.level")= num 0.95
##  $ estimate   : Named num 0.6
##   ..- attr(*, "names")= chr "probability of success"
##  $ null.value : Named num 0.5
##   ..- attr(*, "names")= chr "probability of success"
##  $ alternative: chr "two.sided"
##  $ method     : chr "Exact binomial test"
##  $ data.name  : chr "60 and 100"
##  - attr(*, "class")= chr "htest"
```

Wir sehen hier, dass wir viele der Werte wie bei Listen üblich direkt anwählen können, z.B. den p-Wert:

```
b_test_object[["p.value"]]
```

```
## [1] 0.05688793
```

Oder das den Punktschätzer für p:

```
b_test_object[["estimate"]]
```

```
## probability of success
##                    0.6
```

Wenn wir eine andere Verteilung annehmen, verwenden wir auch eine andere Testfunktion, das Prinzip ist aber sehr ähnlich. Wollen wir z.B. für einen beobachtbaren Datensatz die Hypothese testen, ob der Datensatz aus einer Normalverteilung mit dem Erwartungswert $\mu = 0.5$ stammen könnte, würden wir die Funktion `t.test()` verwenden.

Zum Abschluss dieses Abschnitts wollen wir kurz auf die *Macht von statistischen Tests* (engl: *Power*) und auf die *Wahl zwischen einseitigen und zweiseitigen Tests* eingehen.

Die Macht eines Tests und Fehler 1. und 2. Art:

Wir sprechen von einem *Fehler 1. Art* wenn wir auf Basis eines Tests H_0 verwerfen obwohl sie eigentlich richtig ist. Von einem *Fehler 2. Art* sprechen wir, wenn wir H_0 nicht verwerfen, obwohl H_0 eigentlich falsch ist.

In der Wissenschaft hat es sich ergeben, dass man vor allem auf den Fehler 1. Art schaut. Denn man möchte auf gar keinen Fall eine Nullhypothese verwerfen, obwohl sie eigentlich richtig ist. In der Praxis würde dies bedeuten, eine Aussage zu vorschnell zu treffen. Deswegen wählt man in den empirischen Studien das Signifikanzniveau so, dass die Wahrscheinlichkeit für einen Fehler 1. Art sehr klein ist, in der Regel 5%.

Leider geht damit eine vergleichsweise hohe Wahrscheinlichkeit für einen *Fehler 2. Art* einher, denn die beiden Fehler sind untrennbar miteinender verbunden: reduzieren wir bei gleichbleibender Stichprobengröße die Wahrscheinlichkeit für einen Fehler 1. Art, erhöhen wir damit die Wahrscheinlichkeit für einen Fehler 2. Art und umgekehrt.

Dennoch ist auch ein Fehler 2. Art relevant. Die Wahrscheinlichkeit für einen solchen Fehler ist invers mit der **Macht** (engl: *power*) eines Tests verbunden, die definiert ist als:

$$\text{Macht} = 1 - \mathbb{P}(\text{Fehler 2. Art})$$

Eine vertiefte Diskussion von Macht und dem Trade-Off zwischen Fehlern 1. und 2. Art findet zu einem späteren Zeitpunkt in der Vorlesung statt.

Die Wahl zwischen einseitigen und zweiseitigen Tests:

Wir haben oben am Beispiel der potenziell manipulierten Münze folgendermaßen zwischen einseitigen und zweiseitigen Tests unterschieden: Beim zwei-seitigen Test testen wir $H_0 : p = 0.5$ gegen $H_1 : p \neq 0.5$. Wir überprüfen also ob die Münze entweder zugunsten oder zulasten von Zahl manipuliert wurde.

Beim einseitigen Test testen wir nur gegen eine Alternative: $H_0 : p = 0.5$ bleibt gleich allerdings ist die Alternativhypothese nun entweder $H_1 : p < 0.5$ oder $H_1 : p > 0.5$. Im ersten Fall überprüfen wir also nur ob die Münze zugunsten von Zahl manipuliert wurde, im zweiten Fall nur ob die Münze zugunsten von Kopf manipuliert wurde.

Man mag sich nun fragen wo der Vorteil von einseitigen Tests liegt, erscheint der zweiseitige Test doch allgemeiner. Letzteres ist zwar richtig, allerdings ist die Macht des zweiseitigen Tests im Vergleich zum einseitigen Tests deutlich geringer. Das bedeutet, dass wenn möglich immer der einseitige Test verwendet werden soll. Die Beurteilung ob ein einseitiger oder zweiseitiger Test angemessen ist, muss auf Basis von Vorwissen getroffen werden, und häufig spielen theoretische Überlegungen oder Kontextwissen eine wichtige Rolle.

D.3 Berechnung von Konfidenzintervallen

Konfidenzintervalle für einen Parameter geben eine Antwort auf die Frage: *"Welche Werte für den interessierenden Parameter sind mit unseren Daten kompatibel?"* Wie bei Hypothesentests müssen wir zur Berechnung von Konfidenzintervallen ein Signifikanzniveau α festlegen. Das liegt daran, dass zwischen Konfidenzintervallen und Hypothesentests eine enge Verbindung besteht: ein Konfidenzintervall I_α besteht aus allen Parameterwerten, die bei einem zweiseitigen Hypothesentest zum Signifikanzniveau α als Nullhypothese nicht verworfen werden können.

Wir haben oben auch schon gesehen, dass das Konfidenzintervall ganz leicht aus den typischen Test-Funktionen in R ausgelesen werden kann. Für das Beispiel der Binomialverteilung schreiben wir daher nur:

```
b_test_object <- binom.test(x = 60, n = 100, p = 0.5, alternative = "two.sided")
b_test_object[["conf.int"]]
```

```
## [1] 0.4972092 0.6967052
## attr(,"conf.level")
## [1] 0.95
```

Die Interpretation dieses Intervals ist dabei die folgende: wenn der zugrundeliegende Datengenerierungsprozess sehr häufig wiederholt werden würde, dann würde 95% der jeweils berechneten 95%-Konfidenzintervalle diesen wahren Wert enthalten. Wir können **auf gar keinen Fall** behaupten, dass ein bestimmtes Konfidenzintervall den wahren Parameterwert mit einer Wahrscheinlichkeit von 95% enthält. Eine solche Aussage macht auch keinen Sinn: der wahre Wert ist - wie eingangs beschrieben - keine Zufallsvariable.[8]

[8]Diese Interpretation ist etwas sperrig und das hängt mit dem frequentistischen Wahrscheinlichkeitsbegriff zusammen, den wir hier verwenden. Einen philosophisch attraktiveren Weg stellt der bayessche Wahrscheinlichkeitsbegriff, auf dem die die Bayesianische Statistik aufbaut. Letztere werden wir hier allerdings nicht behandeln können

Appendix E

Referenzen

Bibliography

Anscombe, F. J. (1973). Graphs in statistical analysis. *The American Statistician*, 27:17–21. DOI 10.2307/2682899.

Chatterjee, S. and Firat, A. (2007). Generating data with identical statistics but dissimilar graphics. *The American Statistician*, 61(3):248–254. DOI 10.1198/000313007X220057.

Kleiber, C. and Zeileis, A. (2008). *Applied Econometrics with R*. Springer-Verlag, New York. ISBN 978-0-387-77316-2.

Matejka, J. and Fitzmaurice, G. (2017). Same stats, different graphs: Generating datasets with varied appearance and identical statistics through simulated annealing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 1290–1294, New York, NY. ACM. DOI 10.1145/3025453.3025912.

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.