

Einführung in R

José Carlos García Alanis, Jan Goettmann, Cordula Hunt,
Florian Kobylka, Anna-Lena Schubert, Meike Steinhilber

2022-11-01

Inhaltsverzeichnis

1 Über dieses Skript	5
1.1 Feedback	6
2 Einführung	7
2.1 Installation	7
2.2 Erste Schritte	7
2.3 Das R-Skript	9
2.4 Struktur des Skripts	9
2.5 Pakete	10
2.6 Working Directory	10
3 Erste Schritte	13
3.1 Zahlen	13
3.2 Variablen	14
3.3 Funktionen	16
3.4 Vektoren	18
3.5 Auswahl von Vektorelementen	24
3.6 Logische Vergleiche	26
4 Datenstruktur	33
4.1 RMarkdown	34
4.2 Hilfe	35
4.3 Wiederholung: Werte & Vektoren	35
4.4 Der Workspace	39

4.5 Wiederholung: Einfache Berechnungen	40
4.6 Matrizen	41
4.7 tidy Daten	46
4.8 data.frames (df) und tibbles (tib)	47
4.9 Einlesen und Speichern von Daten	54
4.10 Datensätze (dat) anschauen	56
5 Datenaufbereitung mit dplyr	59
5.1 Einführung	60
5.2 dplyr: Die wichtigsten Befehle	62
5.3 Beispiel: Filtern von Beobachtungen mit filter()	63
5.4 dplyr: Der Piping Operator %>%	66
5.5 Beispiel	67
5.6 dplyr: Neue Variablen mit mutate() berechnen	69
5.7 Beispiel	70
6 Deskriptive Statistik	73
6.1 Allgemeine Informationen eines Datensatzes	74
6.2 LagemaSSe	75
6.3 StreuungsmaSSe	80
7 Graphiken mit ggplot2	81
7.1 Exkurs: Warum viridis?	82
7.2 ggplot2 – Einführung	83
7.3 Vor der Visualisierung: Die Daten	84
7.4 Ästhetische Mapping	86
7.5 Graphischen Funktionen	88
7.6 Sonstige Funktionalität	110
7.7 Facetten	123

Kapitel 1

Über dieses Skript

Dies ist eine Einführung in die Programmiersprache R, die in verschiedenen Lehrveranstaltungen der Arbeitseinheiten *Analyse und Modellierung komplexer Daten* und *Methodenlehre und Statistik* des Psychologischen Instituts der Johannes Gutenberg-Universität Mainz genutzt werden kann. R ist eine unter Psycholog:innen weit verbreitete Programmiersprache, mit deren Hilfe unterschiedlichste Datenarten aufbereitet und analysiert werden können. Das Programm ist *Open Source* und es gibt eine riesige Community von Entwickler:innen, die an Erweiterungen für die Software arbeiten, um Forschenden das (datenanalytische) Leben zu erleichtern.

Dieses Skript wurde als Begleitmaterial für verschiedene Lehrveranstaltungen entwickelt, in denen mit R gearbeitet wird. Es kann als Kursmaterial, aber auch als Nachschlagewerk verwendet werden. Nicht alle Inhalte des Skripts sind für jeden Kurs relevant. Ihre Kursleitung wird Ihnen mitteilen, wann Sie welche Teile des Skripts bearbeiten sollen. Sie können sich natürlich darüber hinaus jederzeit weiter umschauen, welche Möglichkeiten R noch bietet!



Artwork by Allison Horst

1.1 Feedback

Derzeit ist das Skript ein lebendiges Dokument, das auf Basis von Rückmeldungen ständig überarbeitet und verbessert wird. Sollten Ihnen Fehler im Skript auffallen oder sollten Sie sonstiges Feedback haben, freuen wir uns sehr, wenn Sie uns eine E-Mail senden. Schreiben Sie einfach eine E-Mail an amd_lab@uni-mainz.de.

Kapitel 2

Einführung

2.1 Installation

Um R nutzen zu können, müssen Sie sich sowohl R als auch R-Studio installieren.

R ist eine Open-Source-Software, die zur Datenanalyse genutzt wird. Die große Stärke von R liegt in der Verfügbarkeit zahlreicher Zusatzfunktionen, in Form sogenannter Packages. Entwickler:innen überall auf der Welt bieten Packages als Lösungen für die unterschiedlichsten Probleme an, von der Datenaufbereitung über APA-formatierten Tabellen bis hin zu zahlreichen statistischen Analysen. Laden Sie sich R unter dem folgenden Link herunter und installieren Sie sich das Programm auf Ihrem PC/Laptop: <https://cran.r-project.org/>.

Außerdem benötigen Sie noch R-Studio. R-Studio ist eine integrierte Entwicklungsumgebung“ (engl: integrated development environment; *IDE*), die es deutlich einfacher macht, mit R zu arbeiten. Installieren Sie sich daher auch R-Studio auf Ihrem PC/Laptop, indem Sie folgendem Link folgen: <https://www.rstudio.com/products/rstudio/download/#download>

2.2 Erste Schritte

Nachdem Sie R und R-Studio installiert haben, können Sie Ihre ersten Schritte in R machen. Öffnen Sie dazu R-Studio und geben Sie etwas in das Feld „Konsole“ (oder „Console“) ein. Im Normalfall finden Sie die Konsole in der Anzeige auf der linken Seite (ggfs. befindet sich die Konsole auch links unten). Sie erkennen die Konsole daran, dass die Zeile, in die Sie etwas eingeben können, mit > beginnt. Diese Klammer fordert Sie auf, R-Code einzugeben! Geben Sie Folgendes in die Konsole ein:

```
> "Hallo R!"
```

Wenn folgende Ausgabe erscheint, hat die Installation funktioniert:

```
## [1] "Hallo R!"
```

Sie können auch Zahlen in die Konsole eingeben:

```
42
```

```
## [1] 42
```

Jetzt können Sie R schon als einfachen Taschenrechner benutzen!

Addition:

```
3 + 2
```

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

Subtraktion:

```
3 - 2
```

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

Multiplikation:

```
3 * 2
```

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

Division:

```
3 / 2
```

```
## [1] 1.5
```

Beachten Sie dabei, dass Sie auch Klammern setzen können:

```
(3 + 2) * 5
```

```
## [1] 25
```

2.3 Das R-Skript

In der Regel werden Sie Ihre Analysen nicht direkt in die Konsole eingeben, sondern ein *Skript* schreiben, in dem Sie Ihre Analysen dokumentieren. Dieses Skript dokumentiert Ihre Analysen, was den groSSen Vorteil hat, dass Ihre Analysen dadurch reproduzierbar werden. Wenn Sie Daten aufbereiten oder analysieren und dabei unterbrochen werden, können Sie sich – auch noch Tage oder Wochen später – einfach wieder an das Skript setzen, die bisherigen Analyse-schritte erneut ausführen und dort weitermachen. So können Sie Ihre Analysen auch mit anderen Studierenden oder Lehrenden, die ein Projekt betreuen, teilen. Viele Forschende laden ihre R-Skripte regelmäSSig ins Open Science Framework (OSF) hoch, wenn sie Artikel zur Begutachtung einreichen, damit andere Ihren Code und Ihre Analysen auf Fehler überprüfen und nachvollziehen können.

Um ein solches Skript zu erstellen, nutzen wir den in R-Studio verfügbaren Texteditor. Sie können ein neues Skript unter „Datei NeueDatei R Skript“ („File New File R Script“ öffnen. Skripte, die R-Code enthalten, speichern wir mit der Dateiendung „.r“ oder „.R“ ab.

Das Praktische: Wenn Sie Code im Skript schreiben, können Sie diesen Code direkt ausführen. Wenn sich Ihr Cursor in einer Zeile befindet, in der Code steht, können Sie STRG + Enter bzw. bei macOS + Enter drücken (oder oben auf das Run-Symbol klicken), um diese Zeile auszuführen. Wenn Sie einen Teil des Skripts markieren, wird nur genau dieser Teil ausgeführt, wenn Sie STRG + Enter bzw. + Enter drücken. Das Ergebnis wird Ihnen wie gewohnt in der Konsole angezeigt.

2.4 Struktur des Skripts

Nichts ist wichtiger als gute Organisation! Damit Sie sich auch später noch daran erinnern, welche Analysen Sie durchgeführt haben, können (und sollten) Sie sich Kommentare ins Skript schreiben. In R wird `#` benutzt, um den Code zu kommentieren. Wenn Sie `#` vor dem Code setzen, wird dieser Code beim Ausführen einfach ignoriert! So können Sie sich ganze Abschnitte dazu notieren, welche Analysen Sie gemacht haben, was die Kernbefunde waren, usw., ohne dass Sie Probleme beim Ausführen Ihres Skriptes bekommen. Sie sollten Ihren Code *immer* kommentieren, um sich auch später noch daran erinnern zu können, was Sie vor einiger Zeit geschrieben haben.

```
# 3 + 2
# Nichts passiert - die Rechnung wurde nicht ausgeführt, weil sie
auskommentiert ist
```

Sie können Kommentare auch nutzen, um das Skript in Abschnitte zu gliedern. Wenn Sie hinter ein `#` noch vier `-` (also: `# ----`) setzen, fügt der Editor einen logischen Abschnitt ein, den Sie ein- oder ausklappen können. Das ist insbesondere zur Strukturierung längerer Skripte enorm hilfreich.

2.5 Pakete

Für R gibt es unzählige nützliche Pakete, die von Entwickler:innen auf der ganzen Welt weiterentwickelt werden. Auch an der JGU werden R-Pakete entwickelt! Diese Pakete erweitern das Grundprogramm und geben Ihnen Tools an die Hand, die Ihnen bei der Bearbeitung ganz konkreter Fragestellungen helfen können. Das Paket `psych` wurde von William Revelle entwickelt, um verschiedene Funktionen zu bündeln, die für verschiedene psychologische Fragestellungen nützlich sind. Installieren Sie das Paket mit der Funktion `install.packages()`.

```
install.packages("psych")
```

Sobald Sie das Paket installiert haben, müssen Sie es nur noch laden, um es nutzen zu können:

```
library(psych)
```

Und schon haben Sie Ihr erstes Paket installiert und geladen! Sobald Sie ein Paket auf Ihrem PC/Laptop installiert haben, können Sie es immer wieder verwenden. Beachten Sie aber unbedingt, dass Sie Pakete nach jedem Neustart von R neu laden müssen.

Wenn Sie keine Idee haben, welche Funktionen ein Paket umfasst und wofür es nützlich sein könnte, können Sie ganz einfach die Hilfe-Funktion nutzen, die Sie aufrufen können, indem Sie ein Fragezeichen vor den Paketnamen setzen.

```
?(psych)
```

2.6 Working Directory

Wenn Sie in R arbeiten, arbeiten Sie immer in einem Verzeichnis – einem sog. „Working Directory“. Es beschreibt den Dateipfad auf Ihrem Computer, auf den

R per Default zugreift, um Daten einzulesen oder zu speichern. Sie können sich Ihr aktuelles Working Directory mit dem Befehl „`getwd()`“ anzeigen lassen.

`getwd()`

```
## [1] "/Users/runner/work/R-Kurs-Buch/R-Kurs-Buch"
```

In der Regel werden Sie für jedes Projekt ein eigenes Working Directory anlegen, in dem Sie die Daten und Skripte speichern, die zu dem Projekt gehören. Um in das richtige Verzeichnis zu wechseln, können Sie den Befehl `setwd()` nutzen (In manchen Fällen, z.B. bei Computern mit Windows-Betriebssystem, müssen Sie alle im Dateipfad enthaltenen „`"` zu“`/` oder „`\`“ umändern). Noch einfacher geht es per Mausklick über Session Set Working Directory Choose Directory.

Tipp: Noch einfacher geht es mit dem R-Paket `here`: Wenn Sie bereits ein Verzeichnis für Ihr Projekt angelegt und dort ein Skript gespeichert haben (**wichtig: es muss unbedingt im Verzeichnis gespeichert sein!**), liest das Paket `here` automatisch den Pfad aus, an dem ein Skript gespeichert ist. Das ist auch dann enorm praktisch, wenn Sie Ihre Skripte mit anderen austauschen!



Artwork by Allison Horst

Dazu müssen Ihr allerdings zunächst *einmalig* das Paket „`here`“ installieren und dann die Funktion `here()` in die Funktion `setwd()` einfügen.

```
install.packages("here")
setwd(here::here())
```

Kapitel 3

Erste Schritte

3.1 Zahlen

Wir haben in der Einleitung bereits das Rechnen mit Zahlen in R kennengelernt. Zahlen gehören zu den grundlegendsten Arten von Daten, die wir in R verarbeiten können. Mit ihnen können wir eine Reihe von arithmetischen Berechnungen durchführen:

- $x + y$ – Addition von x und y
- $x - y$ – Subtraktion von x und y
- $x * y$ – Multiplikation von x und y
- x / y – Division von x durch y
- $x ^ y$ oder alternativ $x ** y$ – Potenzierung, x hoch y

Bis jetzt haben wir nur mit *ganzen Zahlen* gerechnet. Wir können aber auch mit Zahlen rechnen, die Nachkommastellen haben. Zahlen mit Nachkommastellen haben eine Besonderheit in R: Wir verwenden nicht ein Komma, sondern einen Punkt als Dezimaltrennzeichen, wie im Englischen.

```
2.5 * 10
```

```
## [1] 25
```

```
2.5 + (9 / 4)
```

```
## [1] 4.75
```

Zahlen werden aber auch manchmal in der sogenannten **wissenschaftlichen Notation** dargestellt – vor allem, wenn sie sehr groß oder sehr klein sind (z.B. *p*-Werte). Wenn wir beispielsweise die Zahl 0.0000000012 in R eingeben, erhalten wir 1.2e-09 als Ausgabe. Dies ist kein Fehler, sondern lediglich eine andere Darstellung dieser Zahl. Bei der wissenschaftlichen Notation hat eine Zahl zwei Teile, die von einem e voneinander getrennt werden: Der erste Teil ist ein Faktor, während der zweite Teil der Exponent einer Zehnerpotenz ist. Ausgeschrieben wäre diese Zahl aus der wissenschaftlichen Notation: 1.2 10^9 und damit äquivalent zu 0.0000000012.

3.2 Variablen

Sicherlich wollen Sie Ihre Daten nicht nur in der Konsole ausgeben lassen und bearbeiten, sondern auch in Variablen speichern. Variablen sind Bezeichnungen, mit deren Hilfe Sie auf gespeicherte Daten zugreifen. Sobald eine Variable definiert wurde, können Sie immer wieder darauf zugreifen. Variablen werden mit Hilfe des <- Operators definiert.

```
# Hier werden zwei Variablen - x und y - definiert
x <- 2
y <- 3
```

Sie können sich die Werte dieser Variablen ausgeben lassen, indem Sie die Variablennamen in der Konsole eingeben und Enter drücken.

```
x
```

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

Jetzt können Sie bereits erste Berechnungen mit Variablen durchführen:

```
x + y
```

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

```
x * y
```

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

In R lassen sich sämtliche Objekte – nicht nur einzelne Zahlen, sondern auch Datentabellen, Wortlisten oder sogar Ergebnisse komplizierter Analysen – in Variablen speichern. Der Workflow ist so ausgelegt, dass Sie die Ergebnisse einer Analyse in einer Variable speichern und von dort aus weiterverarbeiten können, z.B. um APA-konforme Tabellen oder Grafiken zu erstellen oder sich EffektstärkemaSSe ausgeben zu lassen. Im weiteren Verlauf des Kurses lernen Sie andere Datentypen kennen, die in Variablen gespeichert werden können.

Variablennamen können auch Umlaute, Unterstriche ("_") und Punkte (".") enthalten, sie dürfen aber nicht mit einer Zahl oder einem Unterstrich beginnen.

Achtung: Wenn Sie eine Berechnung mit einer Variable durchführen und das Ergebnis dieser Berechnung speichern möchten, müssen Sie es wieder eine Variable zuweisen – es wird nicht automatisch gespeichert!

```
z <- x * 10
```

Wenn Sie sich jetzt `x` und `z` ausgeben lassen, sehen Sie, dass sich nichts an `x` geändert hat – das Ergebnis der Berechnung wurde nicht in der Variable `x` abgespeichert. In der Variable `z` hingegen sehen Sie das Ergebnis Ihrer Berechnung.

```
x
```

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

```
z
```

```
## [1] 20
```

`x`, `y` und `z` sind denkbar schlechte Variablennamen! Gute Variablennamen sprechen zu den R Benutzer:innen, d.h. dass der Name eine Variable verrät, was sich „in“ dieser Variable verbirgt. Deswegen sollten Sie sich immer bemühen, möglichst klare und eindeutige Variablennamen zu vergeben.

```
# Beispiele für gute Variablennamen

durchschnittliches_Alter <- 23

MW_Alter <- 23

durchschnittliches_Evaluationsergebnis_KursA <- 2

MW_Evaluationsergebnis_KursA <- 2
```

Scheuen Sie sich nicht vor langen Variablennamen! Je besser und klarer Sie Ihre Variablen definieren, desto einfacher wird es sowohl Ihnen als auch anderen fallen, Ihren Code nachzuvollziehen. Besonders einfach lesbar sind längere Variablennamen, wenn Sie unterschiedliche Elemente des Variablenamens mit einem Unterstrich trennen. Diese Konvention wird auch in diesem Kurs verwendet.

Achtung: Variablennamen beachten GroSS- und Kleinschreibung! Wenn wir eine Variable den Namen `durchschnittliches_Alt` zuweisen, müssen wir auch diesen Namen – genau so, wie wir ihn geschrieben haben – verwenden, um wieder auf die Variable zuzugreifen. Würden wir stattdessen beispielsweise `durchschnittliches_alter` (kleingeschriebenes A, bei Alter) oder `Durchschnittliches_Alter` (groSSgeschriebenes D am Anfang) schreiben, würden wir eine Fehlermeldung bekommen, dass die Variable nicht gefunden werden konnte.

3.3 Funktionen

Wir sind alle bereits mit Funktionen aus dem Mathematikunterricht vertraut. Trigonometrische, logarithmische und Wurzelfunktionen sind auf jedem Schul-taschenrechner vorhanden und geben uns eine Berechnung basierend auf unserer Eingabe zurück.

Funktionen in R haben eine ganz ähnliche Aufgabe und die meisten Funktionen, die wir bereits aus dem Mathematikunterricht kennen, finden sich auch wieder in R. Funktionen stellen in erster Linie eine Reihe von Anweisungen dar, die ausgeführt werden. Beispielsweise besteht die Berechnung des Mittelwerts aus zwei Anweisungen: (1) wir müssen alle Zahlen zu einer Summe addieren und (2) diese Summe durch die Anzahl der Zahlen dividieren.

Anstatt beide Anweisungen immer wieder separat zu schreiben, ist es deutlich einfacher und übersichtlicher eine Funktion hierfür zu definieren, die diese Schritte ausführt. Daher sind Funktionen besonders für **wiederkehrende Abläufe** sinnvoll.

Funktionen erkennt man in der Regel daran, dass der Name der Funktion von Klammern () gefolgt wird. Oft stehen zwischen den Klammern auch noch Daten oder Variablen, die als *Eingabe* verwendet werden und an die Funktion übergeben werden, die sogenannten *Funktionsargumente*.

In R sind bereits über tausend Funktionen eingebaut. Als statistische Programmiersprache, sind die meisten Funktionen darauf ausgelegt, mathematische und statistische Berechnungen durchzuführen.

3.3.1 Funktionsargumente

Funktionen haben sogenannte **Funktionsargumente** und damit eine Art *Platzhalter*, die wir einer Funktion als Eingabe übergeben können. Viele Funktionen haben nur ein einziges Funktionsargument, beispielsweise die Wurzelfunktion oder die Exponentialfunktion. Es gibt aber auch Funktionen, die mehr als ein Funktionsargument besitzen.

Nehmen wir als Beispiel die Funktion `round()` aus R. Sie rundet Zahlen auf eine gewisse Anzahl an Nachkommastellen, die wir selbst festlegen können. `round()` hat daher auch zwei Funktionsargumente: Das erste Argument entspricht den Zahlen, die gerundet werden sollen und das zweite Argument, der Anzahl der Nachkommastellen, auf die gerundet werden soll. Einzelne Funktionsargumente werden jeweils mit einem Komma , voneinander getrennt in der Klammer geschrieben.

Wenn wir also beispielsweise die Zahl 0.0249 auf drei Nachkommastellen runden wollten, könnten wir dies in R mit folgendem Aufruf tun:

```
round(0.0249, 3)
```

```
## [1] 0.025
```

3.3.1.1 Benannte Funktionsargumente

Funktionsargumente in R haben auch *Namen*. Sie funktionieren damit nicht nur wie einfache Platzhalter, sondern analog zu Variablen, die dann innerhalb der Funktion verwendet werden. Wir können die Namen der Funktionsargumente auch mit angeben, wenn wir eine Funktion aufrufen.

Um herauszufinden, wie die Funktionsargumente benannt sind, können wir entweder in RStudio mit dem Cursor auf eine Funktion gehen und F1 auf der Tastatur drücken oder die Dokumentation anschauen (z.B. indem wir ein Fragezeichen gefolgt von dem Funktionsnamen in die Konsole eingeben: `?round`).

Wenn wir dies für `round()` tun, erhalten wir die Funktionsdefinition: `round(x, digits = 0)`. Die Funktion `round()` hat damit zwei Argumente: `x` und `digits`, die wir auch entsprechend angeben könnten:

```
round(x = 0.0249, digits = 3)
```

```
## [1] 0.025
```

Wenn wir Funktionsargumente mit ihrem Namen angeben, können wir dies in jeder beliebigen Reihenfolge tun! Wenn wir allerdings Funktionsargumente ohne Bezeichnung angeben, muss dies in der Reihenfolge erfolgen, wie sie in der Dokumentation steht.

```
round(digits = 3, x = 0.0249)
```

```
## [1] 0.025
```

3.3.1.2 Standardparameter

Viele Funktionen in R haben Funktionsargumente mit **Standardparametern**. Das heißt, dass wir das Funktionsargument nicht *zwangsläufig* angeben müssen. Wenn wir ein Funktionsargument nicht angeben, das einen Standardparameter hat, wird dieser Standardparameter automatisch als Funktionsargument genommen. Schauen wir uns dazu nochmal ein Beispiel an.

`round()` hat beispielsweise einen Standardparameter für das zweiten Funktionsargument (`digits`), dass die Anzahl der Nachkommastellen angibt, auf die gerundet werden soll. Dies erkennen wir daran, dass bei der Funktionsdefinition ein Gleichheitszeichen nach dem Argument steht, welches den Standardwert des Parameters angibt: `round(x, digits = 0)`. Das Funktionsargument `digits` hat demnach einen Standardparameter von 0. Das heißt, dass `round()` jeweils auf eine ganze Zahl runden würde, wenn wir den zweiten Parameter nicht mit angeben.

```
round(0.0249, 3)
```

```
## [1] 0.025
```

```
round(0.0249)
```

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

Bei manchen Funktionsaufrufen wollen wir nur einige Funktionsargumente selbst angeben – für alle Übrigen sollen die Standardparameter verwendet werden. Hierfür können wir unser Wissen aus dem vorigen Abschnitt nutzen: Wir geben einfach nur jeweils die Funktionsargumente mit ihren Namen an, die wir selbst angeben wollen und lassen den Rest frei. Für die freigelassenen Funktionsargumente wird dann jeweils ihr Standardparameter verwendet.

3.4 Vektoren

Jede Spalte eines Datensatzes ist ein Vektor. In einem Vektor befinden sich mehrere Elemente eines Datentyps, also z.B. mehrere Zahlen oder mehrere Wörter. Vektoren werden mit Hilfe der „combine“-Funktion `c` erstellt.

```
# Hier wird ein numerischer Vektor mit den Elementen 1 bis 10 erstellt.
# Mit einem Doppelpunkt können alle ganzen Zahlen zwischen den beiden angegebenen Zahlen angesprochen werden.
Vektor_numeric <- c(1, 2, 3, 4, 5, 6:10)
```

Mit jedem Datentyp können Sie unterschiedliche Operationen durchführen. Um sich den Datentyp eines Vektors anzeigen zu lassen, können Sie die Funktion `mode` verwenden.

```
mode(Vektor_numeric)
```

```
## [1] "numeric"
```

3.4.1 Datentyp numeric

Mit Vektoren vom Typ `numeric` (kurz `num`) können Sie verschiedene mathematische Operationen durchführen. Sie können diese Vektoren addieren, multiplizieren, usw. Wenn Sie eine Operation wie `* 2` auf den Vektor anwenden, wird diese Operation auf alle Elemente des Vektors angewendet.

```
Vektor_numeric * 2
```

```
## [1] 2 4 6 8 10 12 14 16 18 20
```

Wenn Sie hingegen zwei gleich lange Vektoren haben, wird jedes i -te Element des ersten Vektors mit dem i -ten Element des zweiten Vektors verrechnet. Das Element an Position 1 in einem Vektor wird dann mit dem Element an Position 1 im anderen Vektor gepaart, das Element an Position 2 in einem Vektor mit dem Element an Position 2 im anderen Vektor – und so weiter.

```
punkte_MC_Fragen <- c(5, 2, 2, 4, 3, 2, 1, 0, 1, 4)
punkte_offene_Fragen <- c(3, 4, 5, 0, 2, 3, 1, 3, 5, 1)
klausurergebnis <- punkte_MC_Fragen + punkte_offene_Fragen
klausurergebnis

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

3.4.2 Datentyp character

Wenn Sie in einer Variable Text abspeichern wollen, definieren Sie eine Variable vom Datentyp **character** (kurz **chr**). (Variablen vom Datentyp **character** werden auch oft *Strings* genannt.) Text wird mit doppelten oder einfachen Anführungszeichen angegeben:

```
"Alpha"
```

```
## [1] "Alpha"
```

```
'Beta'
```

```
## [1] "Beta"
```

```
Vektor_character <- c("Lea", "Luke", "Han", "Chewy")
```

```
mode(Vektor_character)
```

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

Mit Vektoren des Typs **character** können Sie natürlich keine mathematischen Operationen durchführen. Sie sind aber nützlich, um bestimmte Daten zu kodieren, wie bspw. das Geschlecht oder das Studienfach von Versuchspersonen.

```
Geschlecht <- c("männlich", "weiblich", "weiblich", "divers",
"weiblich")
```

```
Studienfach <- c("Psychologie", "Medizin", "Informatik",
"Sportwissenschaft", "Biologie")
```

3.4.3 Datentyp logical

Der Datentyp **logical** (kurz **logi**) kodiert binäre Informationen – diese sind entweder **TRUE** oder **FALSE**.

```
TRUE
```

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

```
FALSE

## [1] FALSE

Vektor_logical <- c(TRUE, TRUE, FALSE, TRUE)

mode(Vektor_logical)

## [1] "logical"
```

Dabei werden TRUE und FALSE als logische Bedingungen interpretiert, die erfüllt oder nicht erfüllt sein können. Vektoren dieses Datentyps sind für die Datenaufbereitung unglaublich nützlich! Mit Hilfe eines Vektors vom Typ `logical` können Sie beispielsweise kodieren, welche Versuchspersonen die Studie vollständig abgeschlossen haben oder ob einzelne Beobachtungen Ausreißer darstellen. Sie werden diesen Datentyp häufig benötigen, wenn Sie in Datentabellen einzelne Fälle auswählen oder Versuchspersonen ausschließen möchten. Hierbei wird dann jeweils im Einzelfall überprüft, welche Fälle eine gewünschte Bedingung erfüllen (z.B. maximal 30 Minuten zur Bearbeitung eines Tests gebraucht haben) und daher in den weiteren Analysen eingeschlossen werden.

```
# Hier wird festgehalten, welche Versuchspersonen (id) die Studie
# vollständig abgeschlossen haben (Studie_abgeschlossen)

id <- 1:10

Studie_abgeschlossen <- c(TRUE, TRUE, FALSE, TRUE, TRUE, FALSE,
                           FALSE, TRUE, TRUE, TRUE)
```

Oft sollen mehrere Bedingung gleichzeitig überprüft werden, z.B. Personen, die jünger als 18 sind und maximal 30 Minuten zur Bearbeitung eines Tests gebraucht haben. Mehrere logische Bedingungen können durch **logische Operationen** miteinander verknüpft werden.

Die gängigsten logischen Operationen sind UND (`&`), ODER (`l`) und NICHT (`!`). UND und ODER verknüpfen jeweils zwei logische Bedingungen (sprich: zwei logische Werte, also TRUE/FALSE) miteinander und geben selbst einen logischen Wert zurück.

Die Verknüpfung UND ergibt dann TRUE, wenn beide Bedingungen erfüllt sind, d.h. nur wenn die erste und die zweite Bedingung jeweils auch TRUE sind.

```
## Logisches UND (&)
```

```
TRUE & TRUE
```

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

```
TRUE & FALSE
```

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

```
FALSE & FALSE
```

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

Die Verknüpfung ODER ergibt dann TRUE, wenn mindestens eine der beiden Bedingungen erfüllt ist, d.h. wenn die erste oder die zweite Bedingung oder beide Bedingungen TRUE sind. Ganz wichtig: ODER gibt auch dann TRUE aus, wenn beide Bedingungen erfüllt sind!

```
## Logisches ODER (|)
```

```
TRUE | TRUE
```

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

```
TRUE | FALSE
```

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

```
FALSE | FALSE
```

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

Das logische NICHT invertiert eine logische Variable: Aus TRUE wird FALSE und umgekehrt. Das ist hier noch etwas abstrakt, wird aber später in den Kapiteln zur Datenaufbereitung noch klarer.

```
## Logisches NICHT (!)
```

```
!TRUE
```

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

```
!FALSE
```

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

Wie mathematischen Ausdrücke auch, haben logische Operationen ebenfalls eine Rangfolge, nach der sie ausgewertet werden. Daher kann es sinnvoll sein, Klammern () zu verwenden, um logische Operationen zu gruppieren.

```
!TRUE & FALSE | TRUE
```

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

```
!TRUE & (FALSE | TRUE)
```

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

3.4.4 Datentyp factor

Variablen vom Datentyp **factor** sind nützlich, um kategoriale Variablen zu kodieren. Dabei wird zunächst ein Vektor vom Typ **numeric** erstellt. Den einzelnen Werten dieses numerischen Vektors werden dann kategoriale Bezeichnungen zugewiesen.

```
Bedingung <- c (0, 1, 1, 0, 1, 0, 0, 1)
```

```
Vektor_factor <- factor(x = Bedingung,
                           levels = c(0, 1),
                           labels = c("Kontrollgruppe",
                                     "Experimentalgruppe"))
```

```
Vektor_factor
```

```
## [1] Kontrollgruppe Experimentalgruppe Experimentalgruppe
## [4] Kontrollgruppe Experimentalgruppe Kontrollgruppe
## [7] Kontrollgruppe Experimentalgruppe
## Levels: Kontrollgruppe Experimentalgruppe
```

```
mode(Vektor_factor)
```

```
## [1] "numeric"
```

Dazu verwenden Sie die Funktion **factor**. Dieser Funktion übergeben Sie den numerischen Vektor **Bedingung** und definieren anschlieSSend die Stufen des Faktors mit Hilfe des Argument **levels** (hier 0 und 1) sowie die Bezeichnungen mit Hilfe des Argument **labels** (hier Kontrollgruppe und Experimentalgruppe).

Es ist immer sinnvoll, Variablen als Faktoren zu definieren, wenn sie endliche Ausprägungen haben. Experimentelle Bedingungen, Messzeitpunkte oder das Geschlecht von Versuchspersonen sind typische Kandidaten, die als Vektoren vom Typ **factor** gespeichert werden. Der Vorteil dieses Variablentyps besteht darin, dass Sie dort die Zuordnung von numerischen zu verbalen Bezeichnungen direkt vornehmen können und nachher nicht mehr nachschlagen müssen, ob „0“ oder „1“ nun die Experimentalgruppe kodiert. Spätestens bei der Auswertung Ihrer Daten sollten Sie solche Variablen also immer als **factor** rekodieren.

Achtung: Sie können mit Faktoren keine mathematischen Operationen durchführen, auch wenn ihnen eigentlich numerische Kodierungen zugrunde liegen. In R werden Vektoren vom Variablentyp **factor** diesbezüglich ebenso wie Vektoren vom Variablentyp **character** behandelt.

3.4.5 Datentyp NA

Ein Vektor besteht in der Regel nicht nur aus Variablen vom Datentyp **NA**. In echten Datensätzen werden Sie aber immer mal wieder fehlende Daten haben. Diese werden in R als **NA** kodiert.

```
alter <- c(21, 25, 29, 24, NA, 19, 23, 24, 20)
```

Achtung: In anderer Statistiksoftware wie z.B. SPSS hat sich die Konvention entwickelt, fehlende Werte nicht mit **NA**, sondern mit bestimmten nicht beobachtbaren numerischen Werten wie „-9“ oder „-99“ zu beschreiben. Wenn Sie mit einem solchen Datensatz arbeiten, der zuvor in einer anderen Statistiksoftware bearbeitet wurde, sollten Sie immer überprüfen, ob fehlende Werte anhand solcher numerischen Werte kodiert sind und diese als **NA** rekodieren.

3.5 Auswahl von Vektorelementen

Sie können auf einzelne Elemente eines Vektor zugreifen, indem Sie den Operator **[]** verwenden. In die eckigen Klammern wird die Position des Elements

eingefügt, das Sie auswählen möchten. Dieses Vorgehen wird **Indizierung** genannt.

```
daten <- c(3, 7, 9, 0, 1, 1, 4, 5)

# Hier wird das dritte Element ausgewählt

daten[3]
```

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

Sie können auch eine sogenannte **Negativindizierung** durchführen, d.h. im Index festhalten, welches Element Sie *nicht* auswählen möchten.

```
# Hier werden alle Elemente auer dem dritten ausgewählt

daten[-3]
```

```
## [1] 3 7 0 1 1 4 5
```

Selbstverständlich können Sie auch gleich mehrere Elemente auswählen. Wenn Sie z.B. alles vom zweiten bis fünften Element auswählen wollen, können Sie das im Index so vermerken: `[2:5]`. Ganz allgemein gilt: Die Zahl *vor* dem Doppelpunkt gibt an, wo die Indizierung beginnt, und die Zahl *nach* dem Doppelpunkt gibt an, wo die Indizierung endet.

```
# Hier werden die Elemente 2, 3, 4 und 5 ausgewählt

daten[2:5]

## [1] 7 9 0 1
```

Wenn Sie mehrere Elemente auswählen möchten, die nicht direkt nebeneinander stehen, müssen Sie diese mit dem „combine“-Operator `c` verknüpfen:

```
# Hier werden die Elemente 2 und 7 ausgewählt

daten[c(2, 7)]
```

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

Die so ausgewählten Daten können Sie natürlich wieder in einer neuen Variable speichern:

```
ausgewahlte_daten <- daten[c(2, 7)]
```

3.6 Logische Vergleiche

Logische Vergleiche können genutzt werden, um bestimmte Fälle – z.B. einzelne Versuchspersonen – auszuwählen. Welche logischen Vergleiche für einen Datentyp sinnvoll sind, hängt von diesem Datentyp ab.

3.6.1 Datentyp numeric

Wenn Sie Daten vom Typ `numeric` haben, können Sie numerische Vergleiche durchführen. Sie können zum Beispiel überprüfen, welche Ihrer Versuchspersonen 18 Jahre oder älter sind. Dazu verwenden Sie den logischen Operator `>=`, den Sie als „gröSSer oder gleich“ interpretieren können (analog dazu liest sich `<=` als „kleiner oder gleich“.)

```
alter <- c(20, 21, 18, 25, 32, 17, 65, 22)
alter >= 18
```

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

Bei numerischen Vektoren können Sie folgende logischen Vergleiche durchführen: (a) gleich (`==`), (b) gröSSer (`>`), (c) kleiner (`<`), (d) gröSSer gleich (`>=`), (e) kleiner gleich (`<=`) oder (f) ungleich (`!=`). Wenn Sie einen logischen Vergleich mit einem gesamten Vektor durchführen, wird für jedes einzelne Element überprüft, ob es der logischen Bedingung entspricht (`TRUE`) oder nicht (`FALSE`).

```
# Hier überprüfen wir, welche Versuchspersonen jünger als 18
# Jahre alt sind
```

```
alter < 18
```

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

```
# Wir können auch überprüfen, welche Versuchspersonen genau 18
# Jahre alt sind
```

```
alter == 18
```

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

3.6.2 Datentyp character

Vektoren vom Typ `character` lassen sich darauf überprüfen, ob diese identisch bzw. nicht identisch mit einer bestimmten Bedingung sind.

```
# Hier wird überprüft, welche Versuchspersonen Psychologie
studieren...
```

```
Studienfach <- c("Psychologie", "Medizin", "Informatik",
  "Sportwissenschaft", "Biologie")
```

```
Studienfach == "Psychologie"
```

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

```
# ... bzw. welche nicht.
```

```
Studienfach != "Psychologie"
```

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

3.6.3 Datentyp factor

Dieser logische Vergleich lässt sich auch mit Daten vom Typ `factor` durchführen.

```
Bedingung <- c(0, 1, 1, 0, 1, 0, 0, 1)
```

```
Bedingung <- factor(x = Bedingung,
  levels = c(0, 1),
  labels = c("Kontrollgruppe",
  "Experimentalgruppe"))
```

```
# Hier wird überprüft, welche Versuchspersonen der
Experimentalgruppe angehören
```

```
Bedingung == "Experimentalgruppe"
```

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

3.6.4 Rekodieren von Variablen mit Hilfe logischer Vergleich

Sie werden logische Vergleiche häufig anwenden, um Daten zu rekodieren. Schauen Sie sich dazu die folgenden Beispiele an.

Mit Hilfe logischer Vergleiche können Sie überprüfen, ob fehlende Werte nicht mit NA, sondern mit „-9“ kodiert wurden, um diese anschlieSSend zu rekodieren.

```
Testwerte <- c(80, 57, 93, 85, 72, 65, -9)
```

```
Testwerte == -9
```

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

Sie können das Ergebnis dieses logischen Vergleichs nun nutzen, um alle Werte, die als „-9“ kodiert wurden, zu rekodieren. Dazu speichern Sie das Ergebnis des logischen Vergleichs einfach in einer neuen Variable ab.

```
fehlende_Werte <- Testwerte == -9
```

Im nächsten Schritt benutzen Sie die neue erstellte Variable vom Typ `logical`, um alle Werte des Vektors `Testwerte`, die den Wert „-9“ haben, durch NA zu ersetzen.

```
Testwerte[fehlende_Werte] <- NA
```

Hier wurde der Werte -9 durch NA ersetzt:

```
Testwerte
```

```
## [1] 80 57 93 85 72 65 NA
```

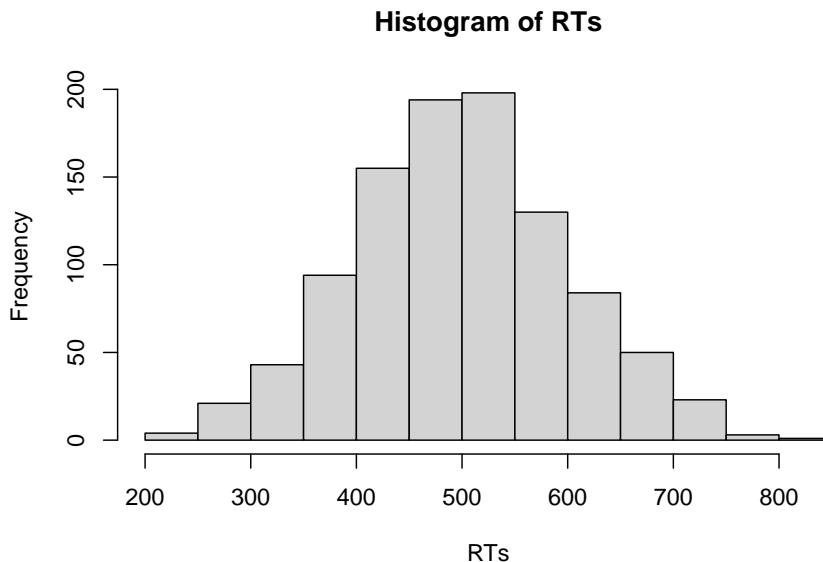
Ganz ähnlich können Sie bei einem Reaktionszeitexperiment alle Trials entfernen, in denen Versuchspersonen sehr langsam waren, weil sie möglicherweise mit ihrer Aufmerksamkeit abgeschweift sind.

Als erstes werden 1000 Trials einer Versuchsperson simuliert

```
RTs <- rnorm(n = 1000, mean = 500, sd = 100)
```

Histogram für die simulierten Daten

```
hist(RTs)
```



```
# Der folgende Vektor gibt an, welche Reaktionszeiten länger 700
# ms sind

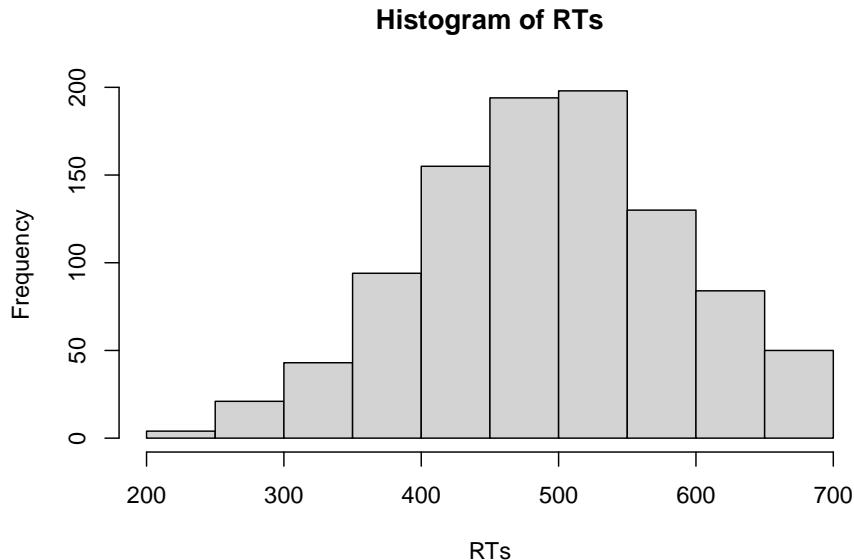
Ausreisser <- RTs > 700

# und hier werden sie durch einen `NA` ersetzt (d.h. gelöscht)

RTs[Ausreisser] <- NA

# Histogram für die Daten ohne Ausrelier

hist(RTs)
```



Für die ganz Eiligen: Sie können dies auch in einem einzigen Schritt durchführen, es ist dann aber fehleranfälliger. Dazu geben Sie den logischen Vergleich direkt als Index an:

```
RTs <- rnorm(n = 1000, mean = 500, sd = 100)

# Hier wird der logische Vergleich direkt als Index von RTs
# genutzt.

RTs[RTs > 700] <- NA
```

Alternativ könnten Sie Extremwerte natürlich durch einen sinnvollen Maximalwert ersetzen. Beispielsweise lassen sich Reaktionszeiten, die länger als 700 ms sind, auf 700 ms „deckeln“.

```
RTs <- rnorm(n = 1000, mean = 500, sd = 100)

# Hier wird der logische Vergleich direkt als Index von RTs
# genutzt.

RTs[RTs > 700] <- 700
```

Sie können auch mehrere logische Vergleiche kombinieren. In Reaktionszeitexperimenten wollen Sie meist extrem langsame sowie extrem schnelle Durchgänge

ausschlieSSen, weil Sie nicht sicher sein können, ob die Versuchspersonen vorschnell reagiert haben (extrem schnell) oder mit ihren Gedanken nicht bei der Sache waren (extrem langsam). Dazu können Sie mehrere logische Vergleiche kombinieren.

```
# Als erstes werden 1000 Trials einer Versuchsperson simuliert

RTs <- rnorm(n = 1000, mean = 500, sd = 100)

# Dann werden untere und obere Grenzen als Variablen definiert.
# Dies hat den Vorteil, dass Sie einfach diese Variablen im Code anpassen
# können, wenn Sie die Kriterien zur Ausreieranalyse anpassen möchten.

untere_Grenze <- 300
obere_Grenze <- 700

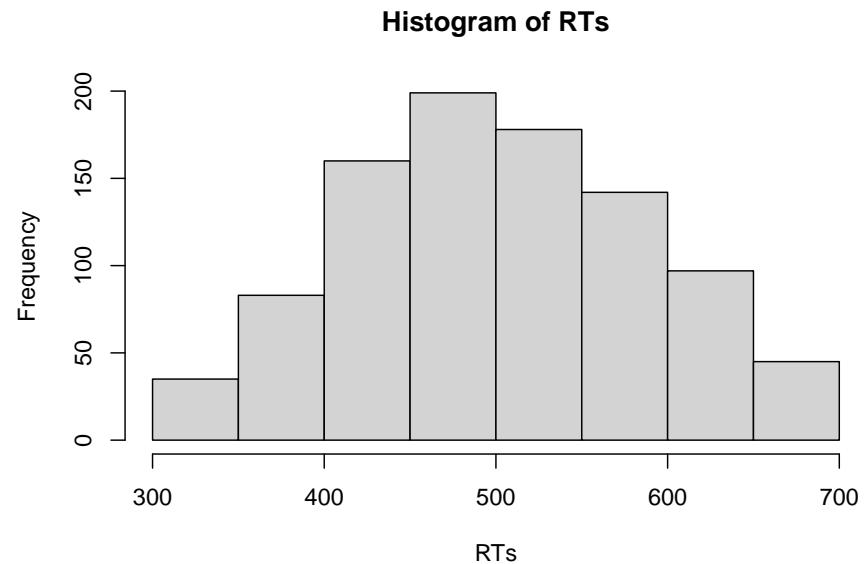
# In diesem Vektor von Reaktionszeiten werden alle RTs, die schneller als 300 ms
# oder länger als 700 ms sind, entfernt. Dazu wird der logische ODER-Operator
# benötigt; es sollen solche Trials identifiziert werden, die < 300 ms ODER
# > 700 ms sind.

Ausreisser <- (RTs > obere_Grenze | RTs < untere_Grenze)

RTs[Ausreisser] <- NA

# Im Histogramm ist zu sehen, dass sowohl sehr schnelle als auch sehr langsame
# Trials aus den Daten entfernt wurden.

hist(RTs)
```



Kapitel 4

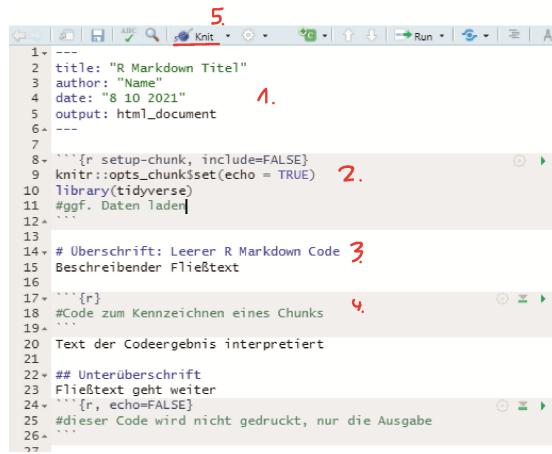
Datenstruktur

Thema	Inhalte
Werte, Vektoren, Listen	<code>chr, num, logi,</code> <code>c(), list(),</code> <code>mode(), coercion,</code> <i>Abruf von Elementen,</i> <code>list(list())</code>
Workspace Berechnungen	<code>rm(), Besen</code> <i>Übersicht Berechnungsfunktionen,</i> <i>z-Standardisierung</i>
Matrizen	<code>matrix(), 2D</code> <i>Indizierung</i>
tidy Daten	<i>Zeilen:</i> <i>Beobachtungen,</i> <i>Spalten: Variablen</i>
<code>tidyverse</code>	<i>Installation und library(package)</i>
<code>data.frame</code> und <code>tibble</code>	<i>Unterschiede,</i> <code>as.data.frame(),</code> <code>as_tibble(), \$,</code> <code>[]</code> , <i>Zugriff auf Zeile, Spalte & Zelle, Zeilennamen</i>
Daten laden und speichern	<i>Import per klick,</i> <code>read./{*}, sep=,</code> <code>dec=, .xlsx, .svs,</code> <code>write./{*}.csv()</code>

Thema	Inhalte
Daten anschauen	<code>View()</code> , <code>head()</code> , <code>str()</code> , <code>count()</code>

4.1 RMarkdown

Das R Markdown Skript ist ein besonderes Dateiformat für R Skripte. Es enthält FlieSStext und eingebetteten R Code:



The screenshot shows an RStudio interface with an R Markdown file open. The code is annotated with numbers 1 through 5:

- 1. Header section: `title: "R Markdown Titel"`, `author: "Name"`, `date: "8.10.2021"`, `output: html_document`.
- 2. Library inclusion: `library(tidyverse)`.
- 3. Blank header chunk: `## Überschrift: Leerer R Markdown Code`.
- 4. Text block: `Beschreibender Fließtext`.
- 5. Knit button: The 'Knit' button in the toolbar is highlighted.

```

1<!--
2 title: "R Markdown Titel"
3 author: "Name"
4 date: "8.10.2021" 1.
5 output: html_document
6---
7
8```{r setup-chunk, include=FALSE}
9 knitr::opts_chunk$set(echo = TRUE) 2.
10 library(tidyverse)
11 #ggf. Daten laden|
12```
13
14# Überschrift: Leerer R Markdown Code 3.
15Beschreibender Fließtext
16
17```{r}
18#Code zum Kennzeichnen eines Chunks 4.
19```
20Text der Codergebnis interpretiert
21
22## Unterüberschrift
23Fließtext geht weiter
24```{r, echo=FALSE}
25#dieser Code wird nicht gedruckt, nur die Ausgabe
26```

```

Knittet man dieses Skript mit dem Wollknäuel Button (5.) in der oberen Leiste, integriert es den ausgeführten Code mit dem FlieSStext und druckt ein übersichtliches Dokument (html, pdf, txt oder doc). Das ist praktisch um z.B. Auswertungsergebnisse zu präsentieren.

1. Im Header werden Titel und Dokumententyp für das Ausgabe-Dokument festgelegt
 2. Die Code Blöcke (**Chunks**) sind mit je drei rückwärts gestellten Hochkomma (Backticks) am Anfang und Ende des Chunks eingerahmt. Werden sie vom R Markdown Skript als solche erkannt, wird auch die Hintergrundfarbe automatisch abgeändert. Im ersten Chunk sollten **globale Chunk Optionen festgelegt**(z.B. Working Directory setzen oder ändern), alle notwendigen **Packages geladen** und die **Daten eingelesen** werden.
 3. Den FlieSStext kann man mit Überschriften (#) und Unterüberschriften (##) strukturieren, im Code kennzeichnet `#` Kommentare
 4. Zu Beginn eines Chunks muss man innerhalb einer geschwungenen Klammer spezifizieren(``{...}``):
- Es ist möglich Codechunks von anderen Programmiersprachen (z.B. Python oder TeX) einzubetten, Standard ist `r`

- (optional) Nach einem Leerzeichen: Einzigartiger Chunk-Name
- (optional) Nach einem Komma: Befehle, um die Ausgabe des Chunks in das neue Dokument zu steuern:
 - `include = FALSE` Weder Code noch Ergebnis erscheinen
 - `echo = FALSE` Nur das Code-Ergebnis erscheint
 - `message = FALSE` Nachrichten zum Code erscheinen nicht
 - `warning = FALSE` Warnungen zum Code erscheinen nicht
 - `fig.cap = "..."` Hiermit lassen sich Grafiken beschriften

4.2 Hilfe

Sie merken, dass die Befehle und Funktionen zum Teil sehr spezifisch sind und Sie sich kaum alles behalten können. Am wichtigsten ist die Reihenfolge und Vollständigkeit der Zeichen: Vergessen Sie ein Komma, ein Backtick oder eine Klammer zu setzen, dann kann R den Code schon nicht interpretieren. Zum Glück erkennt R Studio das oft und weist einen darauf während des Codens mit einem **roten x** neben der Zeilennummer hin. Andernfalls erscheint eine Fehlermeldung beim Ausführen des Codes.

Wenn Sie den Namen einer Funktion oder eines Packages nicht direkt erinnern, können Sie den Anfang des Namens im `Chunk` oder in der `Console` eingeben, RStudio bietet einem nach einem kurzen Moment eine Liste möglicher Optionen an, aus der Sie wählen können. Haben Sie eine Funktion gewählt, können Sie die `Tab`-Taste drücken und es werden die verschiedenen Funktionsargumente angezeigt, um die Funktion zu spezifizieren, was oft sehr hilfreich ist. Möchten Sie wissen, was eine Funktion macht oder in welcher Reihenfolge die Funktionsargumente eingegeben werden, können Sie `?FUN` in die `Console` eintippen, wobei FUN Platzhalter für den Funktionsnamen ist. Alternativ können Sie im `Help`-tab unten rechts suchen. Die Dokumentation ist oft sehr ausführlich.

Im Zweifelsfall haben Sie immer die Möglichkeit einschlägige Suchmaschinen im Internet zu verwenden. Oft werden Sie dabei auf `StackOverflow` weitergeleitet. Auf Englisch gestellte Fragen oder Probleme führen zu besseren Treffern. Noch trivialer ist es, im Skript des Kurses oder im eigenen Code nachzuschauen, die Tasten `STRG + F` habe ich schon 1000 Mal gedrückt, sie könnten dabei hilfreich sein. Falls Sie bei anderen Autoren nachlesen möchten, gibt es Bücher zu R, die meist sogar kostenlos online zur Verfügung stehen und einem eine Einführung in R geben: Z.B. `R Cookbook` oder `R for Data Science`.

4.3 Wiederholung: Werte & Vektoren

Datenformate in R sind von einfach zu komplex: `Value`, Vektor, `matrix`, `(array)`, `data.frame`, `tibble` und `list`. Ihnen noch unbekannte Datenformate

werde im Laufe des Kapitels erklärt. Die kleinste Objekteinheit in R ist ein **Value**. Die unterschiedlichen Datentypen von **Values** sind bereits bekannt:

1. Text, bzw. Charakter (**chr**), auch String genannt,
2. numerische Werte (**num**), auch **double**
3. logische Werte (**logi**)
4. fehlende Werte (**NA**), **Not Available**

Sie weisen einem Objektnamen einen Wert per `<-` zu (Shortkey: ALT + -), der Datentyp des **Values** wird automatisch erkannt.

```
var1 <- TRUE      # Typ logi
var2 <- 3.5        # Typ num
var3 <- "kreativ" # Typ chr
```

Mit der Funktion `mode()` können Sie sich den Datentypen anzeigen lassen. Vektoren reihen Werte desselben Datentyps auf `c(Wert1, Wert2, ...)`:

```
# c() kombiniert die Werte zu einem Vektor, der dem
# Variablenamen vec1 zugewiesen wird:
vec1 <- c(3, 6, 3.4)
```

Wenn Sie versuchen Elemente mit unterschiedlichen Datentypen in einen Vektor zusammenzufassen, z.B. `c("kreativ", 3.5)`, wobei "kreativ" einen komplexeren Typ (Text oder *character*) als 3.5 (numerisch oder *double*) besitzt, werden die Typen der Elemente vereinheitlicht. Dabei wird der Typ des weniger komplexen Element in den Typ des komplexeren Elements umgewandelt, sodass alle Elemente des Vektors den komplexeren Datentyp besitzen. Die Datentypenübersichtstabelle ist von komplex zu einfach (1–4) geordnet. Die Umwandlung des Datentypen nennt sich **coercion**.

```
# Versuche `chr` und `num` zu einem Vektor zu kombinieren:
c("kreativ", 3.5)
```

```
## [1] "kreativ" "3.5"
```

3.5 wird in "" ausgegeben, der numerische Wert wurde in Text umgewandelt.

4.3.1 Coercion (Umwandlung von Typen)

Sie können den Datentypen auch per Funktion ändern, z.B. `as.character()`, `as.double()`:

```
# Verändere die Werte unterschiedlicher Typen zum Typ chr:
```

```
as.character(c(1, TRUE, "abc", 4.1627))
```

```
## [1] "1"      "TRUE"   "abc"    "4.1627"
```

```
# Coerce zum Typ num. Ist dies nicht möglich, erscheinen NAs:
```

```
as.double(c(2, TRUE, "abc", 4.1627))
```

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

```
## [1] 2.0000 NA     NA 4.1627
```

Coercion gibt es auch in Matrizen, Arrays (Mehrdimensionale Matrizen) und in Spaltenvektoren von Datensätzen (data.frames und tibbles). Nur Listen können verschiedene Datentypen und Elemente enthalten `list(Element1, Element2, ...)`. Das können Strings, Vektoren, aber auch Datensätze, Funktionen und sogar andere Listen sein.

4.3.2 Aufruf per Index von Daten aus mehreren Ebenen

Ihnen ist bekannt, dass Sie zum Zugriff auf einzelne Elemente deren Indexnummer verwenden können:

```
vec_4 <- c(1, 3, 3, 7) # Definition eines Vektors
```

```
vec_4[2]          # Abruf des 2. Elements von vec_4
```

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

Das geht auch in verschachtelten Listen:

```
# Definiere eine Liste, die eine Liste und einen Vektor enthält:  

# (Anm.: hier habe ich keine Namen für die Elemente vergeben)  
  

mylist <- list(list(1, "a"),  
              vec_4)  
  

# Rufe Element 1 der äueren Liste: (1, "a"), und davon dann  

# Element 2 ab:  
  

mylist[[1]][2]  
  

## [[1]]  

## [1] "a"
```

Ich habe jetzt mehrere Variablen (Values, Vektoren, Listen) definiert, sie sind in meinem RStudio im **Environment**-tab oben rechts aufgetaucht.

4.3.3 Nullwerte

In (fast) jedem Experiment und jeder Erhebung kommt es mal vor, das Daten fehlen. Es kann sein, dass Versuchspersonen einen Fragebogen nicht ausgefüllt haben oder man die Schrift einfach nicht lesen konnte. Solche Werte müssen ebenfalls kodiert werden, aber entsprechend als *fehlend*. Wir haben bereits gesehen, dass wir solche fehlenden Werte in R mit dem Wert **NA** („not available“) kodieren können. R hat allerdings noch zwei weitere Arten von **Nullwerten**.

Neben **NA** gibt es auch noch **NULL**. Dieser Wert wird verwendet, wenn wir eine Variable definieren möchten, ihr aber noch keinen Wert zuweisen wollen. In diesem Fall können wir der Variable eine Art *Platzhalter* als Wert zuweisen, der gar keinen Wert hat, also quasi *leer* ist. Dieser „leere Wert“ ist **NULL**.

Gleichzeitig können wir **NULL** verwenden, wenn wir ein Objekt aus einer **list** oder eine Spalte aus einem **data.frame** entfernen wollen – wir weisen dem Element einfach den Wert **NULL** zu:

```
Experiment_Reaktionszeit <- data.frame("x" = 1:10, "y" = 11:20)  

Experiment_Reaktionszeit$x <- NULL  

Experiment_Reaktionszeit  
  

##      y  

## 1   11  

## 2   12  

## 3   13  

## 4   14
```

```
## 5 15
## 6 16
## 7 17
## 8 18
## 9 19
## 10 20
```

Der letzte Nullwert, den R kennt, ist `NaN` („Not a Number“). `NaN` taucht immer dann auf, wenn wir Zahlen nicht darstellen können oder das Ergebnis einer Berechnung nicht definiert ist, beispielsweise wenn wir versuchen, durch Null zu teilen oder den Logarithmus einer negativen Zahl berechnen. `NaN` ist technisch gesehen ein numerischer Wert – wir können ihn daher auch zusammen mit anderen numerischen Werten, beispielsweise in einem numerischen Vektor, verwenden.

4.4 Der Workspace

Rechts oben im Fenster ist das `Environment`-tab. Hier sieht man alle im `global Workspace` definierten Objekte (Datenstrukturen: Werte, Vektoren, Matrizen, Arrays, Listen, `data.frames`, `tibbles`; und Funktionen) aufgelistet:

The screenshot shows the RStudio interface with the 'Environment' tab selected. The top bar includes tabs for 'Environment', 'History', 'Connections', and 'Tutorial'. Below the tabs, there are buttons for 'Import Dataset' and a file size indicator '194 MB'. A search bar and a 'List' dropdown are also present. The main area is titled 'Global Environment' and contains a table of objects:

Data	mylist	List of 2
Values	var1	"kreativ"
	var2	3.5
	var3	TRUE
	vec_4	num [1:4] 1 3 3 7
	vec1	num [1:3] 3 6 3.4

Bei Werten, Vektoren und Matrizen steht sogar der Datentyp des Objektes mit dabei. Per Doppelklick können Sie die Objekte jeweils einzeln oben links im extra Fenster (`Datenansicht-tab`) anschauen. `rm(Objektname)` ist die Funktion zum Entfernen einzelner Objekte aus dem `globalen Workspace`. Das Besensymbol im `Environment`-tab oben rechts fegt den `globalen Workspace` leer. Es ist zu beachten, dass R Markdown beim `knit`n nicht auf den `globalen Workspace` zugreift, sondern einen eigenen Workspace aus dem Code in den `Chunks` erstellt. Beim Ausführen einzelner `Chunks` per Markieren und `STRG/CTRL&Enter` oder `grüner Pfeil rechts` wird jedoch auf den `globalen Workspace` zugegriffen. Beim Schließen von RStudio werden Sie gefragt, ob Sie den `globalen Workspace` in die `.RData` als img speichern lassen, dann stehen die Objekte in der nächsten Sitzung wieder zur Verfügung, solange Sie dieselbe Projektdatei öffnen. Offene Skripte und offene `Datenansicht`-tabs werden beim Schließen ebenfalls mit der Projektdatei assoziiert. Geladene Packages gehen leider verloren, diese müssen Sie jedes Mal beim Starten von RStudio neu laden: `library(Packagename)`. Deshalb ist es Konvention am Anfang jedes Skriptes

erstmal die Packages zu laden. Haben Sie Objekte im Workspace gespeichert, können Sie deren Namen verwenden, um sich auf diese zu beziehen und z.B. weitere Berechnungen vorzunehmen.

4.5 Wiederholung: Einfache Berechnungen

```

x <- 5      # definiert den Wert der Variable x
y <- 5      # definiert den Wert der Variable y
x + y      # Summe von x und y
x * y      # Produkt von x und y
sqrt(x)    # Wurzel aus x
x**(1/2)   # x hoch 0.5

# Weise der Variable z das Ergebnis der Gleichung `x + y` zu. `z`
# erscheint im Workspace:

z <- x + y

# Multipliziere die Elemente von vec_4 mit 5 und speichere als
# Variable e:

e <- vec_4 * 5

```

In R gibt es einige integrierte Funktionen um mit Vektoren zu rechnen:

4.5.1 Übersicht Berechnungsfunktionen

Folgende Funktionen können Sie auf `num`-Vektoren und Matrizen anwenden, je nach Funktion auch auf `chr` Vektoren oder Datensätze, wobei diese sich dann meist nur auf die Einträge in der oberen Ebene, z.B. auf die Anzahl der Spalten und nicht auf die Spalteneinträge beziehen.

Funktion	Bedeutung	Funktion	Bedeutung
<code>min(x)</code>	Minimum	<code>mean(x)</code>	Mittelwert
<code>max(x)</code>	Maximum	<code>median(x)</code>	Median
<code>range(x)</code>	Range	<code>var(x)</code>	Varianz
<code>sort(x)</code>	sortiert x	<code>sd(x)</code>	Standardabweichung
<code>sum(x)</code>	Summe aller Elemente	<code>quantile(x)</code>	Quantile von x

Funktion	Bedeutung	Funktion	Bedeutung
<code>cor(x, y)</code>	Korrelation von x und y	<code>length()</code>	Länge von x

4.5.1.1 Beispiel einer z-Standardisierung eines Vektors mit 3 Einträgen

```
# Def. der Variable geschwister:
geschwister <- c(8, 4, 12)

# MW:
mw_geschwister <- mean(geschwister)

mw_geschwister

## [1] 8

# SD :
sd_geschwister <- sd(geschwister)

sd_geschwister

## [1] 4

# z-Standardisierung des Vektors:
z_geschwister <- (geschwister-mw_geschwister) / sd_geschwister

z_geschwister

## [1] 0 -1  1
```

4.6 Matrizen

Matrizen sind 2D-Datenstrukturen, sie bestehen aus Vektoren gleicher Länge und enthalten einen Datentyp. Mit dem Befehl `matrix()` können sie erstellt werden:

```
# Erstellt die Matrix bsp_mat mit 4 Zeilen, 4 Spalten und leeren
# Einträgen:

bsp_mat <- matrix(NaN, nrow = 4, ncol = 4) # NaN (Not a Number)
# ist vom Typ `num` 

bsp_mat

##      [,1] [,2] [,3] [,4]
## [1,]   NaN   NaN   NaN   NaN
## [2,]   NaN   NaN   NaN   NaN
## [3,]   NaN   NaN   NaN   NaN
## [4,]   NaN   NaN   NaN   NaN
```

Ich habe eine 4×4 Matrix erstellt, die mit NaNs gefüllt ist. Hätte ich diverse Datentypen zugeordnet, wären diese zum komplexeren coerced worden. Auf Werte in Matrizen kann mit `matrixname[Zeilennummer, Spaltennummer]` zugegriffen werden. Praktischerweise stehen die entsprechenden Indizes neben der oben angezeigten Matrix. Beispiele zum Auswählen und Zuweisen neuer Werte:

```
# Weil Spalte 1 von bsp_mat und vec_4 dieselbe Länge haben,
# kann ich der Spalte 1 die Einträge von vec_4 zuweisen.
# Dadurch, dass der Eintrag für die Zeilennummer leer ist,
# beziehe ich mich auf alle Zeilen:

bsp_mat[, 1] <- vec_4

bsp_mat

##      [,1] [,2] [,3] [,4]
## [1,]     1   NaN   NaN   NaN
## [2,]     3   NaN   NaN   NaN
## [3,]     3   NaN   NaN   NaN
## [4,]     7   NaN   NaN   NaN

# Recycling: Wird einem Bereich ein einzelner Wert zugeordnet,
# wird dieser vervielfacht (wie oben bei NaN):

bsp_mat[, 2] <- 8

bsp_mat

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

```

## [1,]    1    8   NaN   NaN
## [2,]    3    8   NaN   NaN
## [3,]    3    8   NaN   NaN
## [4,]    7    8   NaN   NaN

# Definiere `logi` Einträge in `num` Matrix:
bsp_mat[, 3] <- c(FALSE, TRUE, FALSE, TRUE)
bsp_mat

##      [,1] [,2] [,3] [,4]
## [1,]    1    8    0   NaN
## [2,]    3    8    1   NaN
## [3,]    3    8    0   NaN
## [4,]    7    8    1   NaN

```

Coercion: TRUE wurde zu 1 und FALSE wurde zu 0. Wenn man nun eine bestimmte Zeile oder Spalte betrachten möchte, kann man dies auch über die Indizierung tun, hierbei kann man sich beliebig austoben. Die Regeln dafür sind dieselben wie bei Vektoren, nur in 2D, wobei stets [Zeile, Spalte] gilt.

```
bsp_mat[, 1] # Wählt alle Zeilen von Spalte 1
```

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

```
bsp_mat[4, 1] # Wählt Zeile 4 von Spalte 1
```

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

Hier wird es turbulent:

```
bsp_mat[c(1, 3), ] # Wählt Zeilen 1 & 3 von allen Spalten
```

```

##      [,1] [,2] [,3] [,4]
## [1,]    1    8    0   NaN
## [2,]    3    8    0   NaN

```

```
bsp_mat[-1, 2:4] # Wählt alle Zeilen außer 1, und Spalten 2-4
```

```

##      [,1] [,2] [,3]
## [1,]    8    1   NaN
## [2,]    8    0   NaN
## [3,]    8    1   NaN

```

Da ich jetzt Bereiche der Matrix auswählen kann, kann ich vielleicht Berechnungen vornehmen:

```
mode(bsp_mat)  # Ist bsp_mat numerisch?

## [1] "numeric"

bsp_mat

##      [,1] [,2] [,3] [,4]
## [1,]     1     8     0   NaN
## [2,]     3     8     1   NaN
## [3,]     3     8     0   NaN
## [4,]     7     8     1   NaN

# Spalte 2 minus Spalte 1 und dann mal Spalte 3:

(bsp_mat[ , 2] - bsp_mat[ , 1]) * bsp_mat[ , 3]
```

```
## [1] 0 5 0 1
```

Es sind immer noch nicht angegebene Nummernwerte in der Matrix. Da ich mich beim Berechnen auf Bereiche der Matrix beschränke, die vergebene numerische Werte haben (Spalten 1 bis 3 ohne `NaN` Einträge), habe ich ein sinnvolles Ergebnis bekommen. Was passiert, wenn ich mit `NaN` Einträgen rechnen möchte?

```
bsp_mat[1, ]      # wählt Zeile 1

## [1] 1 8 0 NaN

# Bilde die Summe über Zeile 1 mit NaN:

sum(bsp_mat[1, ])
```

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

Die Summe der Zeile führt zu keinem interpretierbaren Ergebnis. Zum Auslassen der `NaN`s wird das Funktionsargument `na.rm = TRUE` verwendet (`rm` steht für `remove`):

```
# Bilde die Summe über Zeile 1 ohne NaN:  
sum(bsp_mat[1, ], na.rm = TRUE)
```

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

```
# Bilde den MW der Matrix ohne NaN:  
mean(bsp_mat, na.rm = TRUE)
```

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

Nun, da wir mit dem Rechnen in Matrizen vertraut sind, möchte ich die letzte Spalte mit Einträgen füllen:

```
# Speichere bsp_mat unter sav_mat zur späteren Verwendung:
```

```
sav_mat <- bsp_mat
```

```
# Weise Spalte 4 einen `chr`-Vektor zu:
```

```
bsp_mat[,4] <- c("coercion", "kann", "nervig", "sein")
```

```
bsp_mat
```

```
##      [,1] [,2] [,3] [,4]  
## [1,] "1"  "8"  "0"  "coercion"  
## [2,] "3"  "8"  "1"  "kann"  
## [3,] "3"  "8"  "0"  "nervig"  
## [4,] "7"  "8"  "1"  "sein"
```

```
mode(bsp_mat) # Bestimmt Typ der Matrix
```

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

Konnte ich eben noch den Mittelwert einer Spalte bilden, so geht das jetzt nicht mehr, da alle Einträge der Matrix zu `chr` coerced wurden. In einem typischen Datensatz sind aber Variablen verschiedener Typen (`num` und `chr`) enthalten. Dieses Problem ließt sich mit Listen lösen, welche aber unübersichtlich sind. Datensätze bestehen manchmal aus unüberschaubar vielen Einträgen und deshalb sollten sie wenigstens übersichtlich strukturiert sein.

4.7 tidy Daten

Es gibt eine Konvention dafür, wie man Datensätze, die mehreren Beobachtungseinheiten (Fällen) verschiedene Parameter (Variablen) zuordnet. Wichtig für die eigene strukturierte Arbeit ist in erster Linie Konsistenz, z.B. dass Sie bei Variablennamen aus mehreren Wörtern immer den Unterstrich als Trennzeichen verwenden. Es hat sich als überlegen für die Auswertung von Daten herausgestellt, Fälle in Zeilen und Variablen in Spalten einzurichten. Dieses Prinzip dürfte einige schon von SPSS bekannt sein.

	Variable1	Variable2	Was ist „tidy“ data?
Fall1	Wert11	Wert12	Eine Zeile pro Beobachtung
Fall2	Wert21	Wert22	Eine Spalte pro Variable
Fall3	Wert31	Wert32	Eine Tabelle pro Untersuchung
Fall4	Wert41	Wert42	eindeutige Namen
Fall5	Wert51	Wert52	Konsistenz
Fall6	Wert61	Wert62	...

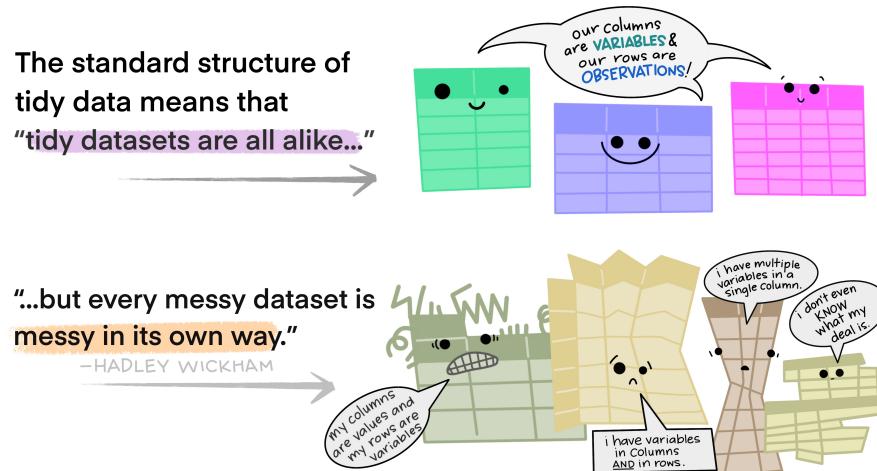


Abbildung 4.1: Illustration by Allison Horst

Es gibt noch weitere Konventionen und Empfehlungen für konsistentes und ordentliches Arbeiten in R und mit Datensätzen im Allgemeinen, z.B. dass man keine Farbcodierungen verwenden sollte. Vorerst genügt es, wenn Sie sich an die Basics hier halten. Diese Art Daten zu strukturieren lässt sich im `data.frame` und noch besser im `tibble` umsetzen: Beides sind Tabellen mit Spaltenvektoren,

die jeweils verschiedene Datentypen enthalten können. Deswegen stellen beide das bevorzugte und für unsere Zwecke wichtigste Datenformat dar.

4.7.1 tidyverse

Bevor wir uns dem übersichtlichsten Datenformat, den **tibbles** widmen, müssen wir das entsprechende Package einmalig in der **Console** installieren. Ich habe den Code auskommentiert, weil das Package bei mir bereits installiert ist:

```
# R kennt den Namen von zu installierenden Packages noch nicht,  
deswegen in "":  
  
#install.packages("tidyverse")
```

Das Package **tidyverse** enthält mehrere nützliche Packages, die eine saubere Datenverarbeitung zum Ziel haben. Packages müssen bei jeder Sitzung neu aktiviert bzw. angehängt werden. Für Sie relevante Packages im **tidyverse** sind **tibble**, **readr**, **stringr**, **dplyr**, **purr** und **ggplot2**.

```
# Bitte gewöhnen Sie sich an, Packages am Anfang eines Skriptes  
zu laden,  
# da Sie dies nach jedem Neustart einer R Session wiederholen  
müssen:  
  
library(tidyverse)
```

4.8 data.frames (df) und tibbles (tib)

Beides sind Tabellen mit Spaltenvektoren (Variablen), die je verschiedene Datentypen enthalten können. Hier zunächst die Übersicht über die Funktionen zum Managen des Datensatzes:

Funktion zum	data.frame()	tibble()
Datenformat konvertieren	as.data.frame()	as_tibble()
Definieren	data.frame(var1, ...)	tibble(var1, ...)
Aufrufen des Datensatzes	df	tib
Auswählen einer Variable	df\$var	tib\$var

Funktion zum	<code>data.frame()</code>	<code>tibble()</code>
Auswählen eines Bereiches	<code>df[rowIdx, colIdx]</code>	<code>tib[rowIdx, colIdx]</code>
Definieren neuer Variablen	<code>df\$var_neu <- c(...)</code>	<code>tib\$var_neu <- c(...)</code>
Ergänzen von Zeilen	<code>rbind(df, Zeilen)</code>	<code>rbind(tib, Zeilen)</code>
Ergänzen von Spalten	<code>cbind(df, Spalten)</code>	<code>cbind(tib, Spalten)</code>
Zeilennamen vergeben	<code>row.names(df) <- c("name1", ...)</code>	<code>relocate(tib, namevec)</code>

Sie können die beiden Datensatz-Formate einfach in das jeweils andere konvertieren. Datensätze lassen sich auch per Formel definieren: `data.frame()` oder `tibble()`, wobei hier die Spaltenvektoren aneinander gereiht werden. Es bietet sich an, dabei direkt Namen für die Spaltenvektoren zu vergeben:

```
# Erzeuge einen data.frame durch Verwendung der Spaltenvektoren
# aus vorherigen Matrizen,
# denen Namen zugewiesen werden (Denken Sie beim Definieren der
# Spaltenvektoren daran
# diese mit einem Komma voneinander zu trennen):

test_df <- data.frame("text" = bsp_mat[, 4],
                      "ist_Verb" = sav_mat[, 3])
test_df

##      text ist_Verb
## 1 coercion      0
## 2 kann         1
## 3 nervig       0
## 4 sein         1
```

In Bezug auf weitere Funktionen des Packages `tidyverse` sind tibbles ein wenig praktischer. Große tibbles werden beim Aufrufen etwas übersichtlicher angezeigt (nur die ersten 10 Zeilen).

```
# Konvertiere data.frame zu tibble:

test_tib <- as_tibble(test_df)

test_tib

## # A tibble: 4 x 2
```

```
##   text      ist_Verb
##   <chr>     <dbl>
## 1 coercion      0
## 2 kann          1
## 3 nervig        0
## 4 sein          1
```

Einzelne Spalten können ganz einfach aufgerufen werden, in dem man den \$-Operator benutzt. Schreibt man diesen direkt hinter den Namen des Datensatzes, klappt automatisch eine Liste mit allen Spalten auf:

```
# Rufe Spalte text aus Datensatz test_tib auf:
test_tib$text
```

```
## [1] "coercion" "kann"      "nervig"    "sein"
```

Es ist auch möglich, mehrere Zeilen und/oder Spalten auszugeben. Dies funktioniert wie bei Matrizen per Indexnummer:

```
# Gibe Zeilen 2 bis 4 aus Spalte 1 aus:
test_tib[2:4, 1]
```

```
## # A tibble: 3 x 1
##   text
##   <chr>
## 1 kann
## 2 nervig
## 3 sein
```

Die Adressierung einzelner Spalten und Zeilen ermöglicht dann zum Beispiel die Berechnung von Kennwerten nur für einzelne Spalten. Z.B. kann man die Kosten für Konzertkarten im Jahr 2022 aufsummieren lassen:

```
# Definiere ein tibble mit 2 Variablen:
tickets_2022 <- tibble("Artist" = c("Ed Sheeran", "Billy Ellish",
                           "The Weeknd", "Dua Lipa",
                           "Imagine Dragons"),
                        "Kosten" = c(79.32, 282, 116, 136, 68.71))

# Berechne die Summe einer dieser beiden Variablen:
sum(tickets_2022$Kosten)
```

```
## [1] 682.03
```

Der \$-Operator wird für fast alle höheren Datentypen verwendet, um auf diese zuzugreifen. Dies gilt zum Beispiel auch für die meisten Outputs von Funktionen (*t*-Test, Anova, SEMs) und Listen, es müssen aber wie im tibble, Namen für die Listeneinträge vergeben worden sein:

```
# Erstelle eine Liste mit diversen Objekten aus meinem Workspace
# und vergabe Namen
# (über welche Sie später auch auf die Listeneinträge zugreifen
# können):

list_of_thingis <- list(tibbi = test_tib,
                        ticki = tickets_2022,
                        geschwi = geschwister,
                        vari = var1)

# Per $-Operator und Name in der Liste wird ein Eintrag gewählt:

list_of_thingis$geschwi
```

```
## [1] 8 4 12
```

```
# Wurde kein Name für den Listeneintrag vergeben, führt der
# Aufruf per Name ins Leere:
```

```
mylist$vec_4
```

```
## NULL
```

```
# Mit mehreren $ können Sie tiefere Ebenen erreichen:
```

```
list_of_thingis$ticki$Artist
```

```
## [1] "Ed Sheeran"      "Billy Ellish"     "The Weeknd"
## [4] "Dua Lipa"        "Imagine Dragons"
```

Mir fällt auf, dass ich den Namen einer Künstlerin in `tickets_2022` falsch geschrieben habe, das möchte ich ändern:

```
# $-Operator und Indexing per Nummer lassen sich auch
# kombinieren:

tickets_2022$Artist[2] <- "Billy Eilish"
```

Sie können also nicht nur Elemente aus Datensätzen abrufen, sondern diese mit dem <- neu zuweisen. Sie können das \$ auch verwenden um ganz neue Spalten in die Datensätze einzufügen:

```
# Definiere eine neue Spalte im Datensatz:

tickets_2022$Priorität <- c(2, 4, 3, 5, 1)

# Besser ae statt ä im Variablenamen:

tickets_2022$Prioritaet <- tickets_2022$Priorität

tickets_2022
```

```
## # A tibble: 5 x 4
##   Artist      Kosten Priorität Prioritaet
##   <chr>     <dbl>    <dbl>     <dbl>
## 1 Ed Sheeran  79.3      2         2
## 2 Billy Eilish 282       4         4
## 3 The Weeknd  116       3         3
## 4 Dua Lipa    136       5         5
## 5 Imagine Dragons 68.7     1         1
```

Nun gibt es eine Spalte zu viel. Ich möchte sie wieder löschen - vorsichtig hiermit(!):

```
tickets_2022$Priorität <- NULL # Entfernt die Spalte
```

Mit den Funktionen `rbind()` und `cbind()` lassen sich die data.frames und tibbles um gleichbreite bzw. gleichlange Vektoren oder Datensätze ergänzen. Verbinden Sie ein tibble und einen `data.frame` miteinander, werden diese zum Format `tibble` coerced. Zunächst füge ich zwei weitere Zeilen hinzu, dann zwei weitere Spalten:

```
# Zwei weitere Konzerte in neuem data.frame:

extra_Konzerte <- data.frame("Artist" = c("Elton John",
"Coldplay", "RHCP"),
```

```

"Kosten" = c(139.00, 259.00, 200),
"Prioritaet" = c(2.7, 2.3, 1.7))

# Zeilenweises Aneinanderbinden:

tickets_2022 <- rbind(tickets_2022, extra_Konzerte)

# Zwischenergebnis:

tickets_2022

## # A tibble: 8 x 3
##   Artist      Kosten Prioritaet
##   <chr>     <dbl>     <dbl>
## 1 Ed Sheeran    79.3      2
## 2 Billy Eilish   282       4
## 3 The Weeknd    116       3
## 4 Dua Lipa      136       5
## 5 Imagine Dragons 68.7      1
## 6 Elton John    139      2.7
## 7 Coldplay      259      2.3
## 8 RHCP          200      1.7

# Ort und Begleitung in neuem tibble:

ort_begleitung <- tibble("Ort" = c("Frankfurt", "London",
                                    "Muenchen", "Berlin", "Gdynia", "Frankfurt", "Frankfurt",
                                    "Koeln"),
                         "Begleitung" = c("Juergen", "Samu",
                                         "Anni", "Jan und Laura", "Oma",
                                         "Papa", "Schwester", "die Girls"))

# Spaltenweises Aneinanderbinden:

tickets_2022 <- cbind(tickets_2022, ort_begleitung)

# Endergebnis:

tickets_2022

## # A tibble: 8 x 6
##   Artist      Kosten Prioritaet      Ort Begleitung
##   <chr>     <dbl>     <dbl> <dbl> <chr>        <chr>
## 1 Ed Sheeran    79.32      2.0  2.0 Frankfurt   Juergen
## 2 Billy Eilish   282.00      4.0  4.0 London      Samu
## 3 The Weeknd    116.00      3.0  3.0 Muenchen    Anni

```

```

## 4      Dua Lipa 136.00      5.0    Berlin Jan und Laura
## 5 Imagine Dragons 68.71      1.0    Gdynia          Oma
## 6      Elton John 139.00     2.7 Frankfurt        Papa
## 7      Coldplay 259.00      2.3 Frankfurt    Schwester
## 8      RHCP 200.00         1.7 Koeln       die Girls

```

Ein Unterschied zwischen tibbles und data.frames ist, dass tibbles keine Zeilennamen kennen. Das vereinfacht das Format. Möchten Sie trotzdem gerne Zeilennamen vergeben, müssen Sie sich mit einer neuen Variable (z.B. `namevec`) behelfen, die Sie mit `relocate(tib, namevec)` an den Anfang des Datensatzes stellen können.

```

# Definiere ein einfaches tibble mit Zeilennamen in einer Spalte:

newtib <- tibble("sp1" = c(1, 2, 3), "namevec" = c("Zeile1",
                                                 "Zeile2", "Zeile3"))

newtib

## # A tibble: 3 x 2
##   sp1 namevec
##   <dbl> <chr>
## 1     1 Zeile1
## 2     2 Zeile2
## 3     3 Zeile3

# Sortiere den Namenvector in Spalte 1:

newtib <- relocate(newtib, namevec)

newtib

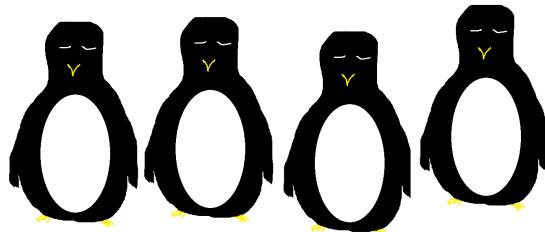
## # A tibble: 3 x 2
##   namevec   sp1
##   <chr>     <dbl>
## 1 Zeile1     1
## 2 Zeile2     2
## 3 Zeile3     3

```

Mit dem Hinzufügen und Abändern und Vertauschen von Spaltenvektoren haben wir schon ein bisschen an der Oberfläche der Möglichkeiten zur Datenaufbereitung gekratzt, die in der nächsten Sitzung ausführlich behandelt werden. Jetzt, wo Sie mit dem Management von Datensätzen vertraut sind, wollen wir vorhandene Datensätze einlesen:

4.9 Einlesen und Speichern von Daten

Daten können in R Studio auf unterschiedliche Weise eingelesen werden.



Es gibt Packages mit frei verfügbaren Datensätzen, z.B. einen Datensatz zu Pinguinen: `palmerpenguins`.

Horst AM, Hill AP, Gorman KB (2020). `palmerpenguins`: Palmer Archipelago (Antarctica) penguin data. R package version 0.1.0. <https://allisonhorst.github.io/palmerpenguins/>

Nach einmaliger Installation des Packages (`install.packages("palmerpenguins")`) muss es geladen werden:

```
# Jedes Mal beim Durchlaufen des Skripts soll das Package
# angehängt werden
# (ohne "", weil R das Package bereits installiert hat und
# kennt):
library(palmerpenguins)
# penguins ist zwar schon ein tibble, aber im Workspace rechts
# ist es nicht zu sehen,
# daher weise ich es dem Namen pingus zu:

pingus <- penguins
```

In der Regel werden Sie aber einen selbst erhobenen oder einen aus dem Internet heruntergeladenen Datensatz einlesen wollen. Mein Tipp ist, den Datensatz in das Working Directory zu speichern, dann finden Sie ihn schneller und er ist in der Nähe Ihrer Auswertung. Noch eleganter ist es einen Unterordner namens `data` in den Ordner des Working Directories anzulegen, in den Sie alle Datensätze zu Ihrem Projekt speichern können. Im `File`-tab unten rechts navigieren Sie zu der Datei mit dem Datensatz und dann klicken Sie diese zum Importieren des Datensatzes an (alternativ können Sie im `Environment`-tab über den Button `Import Dataset` einen Datensatz zum Importieren auf Ihrem Computer suchen). In RStudio erscheint ein Fenster zum Importieren, unten rechts wird der automatisch der dem Dateiformat und unten links angegebenen Optionen entspricht, ggf. werden sogar benötigte Packages geladen.

Um einen Datensatz per Code zu importieren sind Dateiformat, die Trennzeichen (`sep`) und die Dezimalzeichen (`dec`) besonders relevant. Das Standard-Dateiformat ist `.csv`, hier sind Kommata Trennzeichen (`sep=","`) und Punkte kennzeichnen Dezimalstellen (`dec=". "`). Sie können die Funktionen `read_csv()` oder `read_delim()` für dieses Dateiformat verwenden, letztere sollte Trenn- und Dezimalzeichen automatisch erkennen. Hier ist eine Übersicht zu den Einlesefunktionen in base R (also ohne zusätzlich geladene Packages) und im tidyverse Package, der Unterschied ist, dass base R Funktionen die Daten in einen `data.frame` laden, tidyverse Funktionen in ein `tibble`:

Funktion zum	<code>sep</code>	<code>dec</code>	in base R	im tidyverse
autolesen	auto	"."	<code>read.delim()</code>	<code>read_delim()</code>
autolesen	auto	", "	<code>read.delim2()</code>	<code>read_delim2()</code>
lesen von	", "	". "	<code>read.csv()</code>	<code>read_csv()</code>
lesen von	leer	". "	<code>read.table()</code>	<code>read_table()</code>
schreiben	", "	". "	<code>write.csv()</code>	<code>write_csv()</code>

Wichtigstes und oft einziges Funktionsargument ist der vollständige Dateiname, er wird in " angegeben. Alternativ können Sie statt dem Dateinamen auch die Funktion `file.choose()` angeben, die dann ein interaktives Fenster zur Dateiauswahl öffnet. Falls Sie die Datei in einem Unterordner vom Working Directory gespeichert haben (ich erstelle mir meist einen Unterordner namens `data`), wird der Name des Unterordners mit einem / dem Dateinamen vorangestellt. Vergessen Sie nicht, innerhalb der Anführungszeichen sowohl den Namen des Datensatzes als auch das Dateiformat nach einem Punkt zu nennen (z.B.,„data/Datensatz1.csv“). Das Einlesen von Daten funktioniert nur, wenn der einzulesende Datensatz per <- einem Namen zugewiesen wird. Beispiel zum Laden eines `.csv` Datensatzes:

```
# Lese meinen socken.csv Datensatz aus dem Unterordner data in
# ein tibble namens socken:
socken <- read_delim("data/socken.csv")

socken

## # A tibble: 2 x 3
##   Stoff Gewicht Bewertung
##   <chr>   <dbl>     <dbl>
## 1 Seide    0.03      10
## 2 Wolle    0.08       9
```

Excel Dateien werden mit Funktionen `read_excel()`, `read_xls()` oder `read_xlsx()` aus dem Package `readxl`, SPSS Dateien mit der Funktion

`read_svs()` aus dem Package `haven` eingelesen. Auch zum Einlesen von SAS, Stata oder anderen Dateiformaten gibt es entsprechende Funktionen. Die Standardfunktion zum Abspeichern von Datensätzen in eine Datei ist `write_csv()`, bzw. in base R `write.csv()`, da dieses Dateiformat die beste Kompatibilität mit anderer Software aufweist. Beim Speichern müssen Sie neben dem Dateinamen und ggf. dem Dateipfad noch den Namen des Datensatzes, den Sie speichern möchten, als erstes Funktionsargument angeben:

Es gibt noch ein weiteres erwähnenswertes Dateiformat, das von R selbst: `.RDS`. Die Funktionen `saveRDS()` und `readRDS()` bieten die beste Funktionalität in R.

4.10 Datensätze (dat) anschauen

Um sich einen geladenen Datensatz komplett anzuschauen, können Sie diesen im `Workspace` anklicken, oder deren Namen an die Funktion `view(dat)` übergeben. Der Datensatz `pingus` hat 344 Zeilen, das kann ich im `Workspace` sehen. Da er als `tibble` gespeichert ist, könnte ich diesen per Name aufrufen (nur die ersten 10 Zeilen würden dargestellt werden). Mit der Funktion `head(dat)` wird einem der Kopf des Datensatzes ausgegeben, genau genommen die ersten 6 Zeilen. Die Funktion ist besonders nützlich für große `data.frames`, da diese beim Aufrufen per Name die Console überfüllen.

```
# Zeige die ersten 6 Zeilen jeder Variable an:  
  
head(pingus)
```

```
## # A tibble: 6 x 8  
##   species island bill_~1 bill_~2 flipp~3 body_~4 sex     year  
##   <fct>    <fct>    <dbl>    <dbl>    <int>    <int> <fct>    <int>  
## 1 Adelie   Torge~    39.1     18.7     181     3750 male    2007  
## 2 Adelie   Torge~    39.5     17.4     186     3800 fema~  2007  
## 3 Adelie   Torge~    40.3      18       195     3250 fema~  2007  
## 4 Adelie   Torge~     NA       NA       NA      NA <NA>   2007  
## 5 Adelie   Torge~    36.7     19.3     193     3450 fema~  2007  
## 6 Adelie   Torge~    39.3     20.6     190     3650 male   2007  
## # ... with abbreviated variable names 1: bill_length_mm,  
## #   2: bill_depth_mm, 3: flipper_length_mm, 4: body_mass_g
```

Einen Überblick über die Datenstruktur, inklusive Factor-levels (der Faktorstufen) erhalten Sie mit der Funktion `str(dat)`:

```
# Zeige die Struktur der Daten:
```

```
str(pingus)
```

```
## # tibble [344 x 8] (S3: tbl_df/tbl/data.frame)
## $ species      : Factor w/ 3 levels "Adelie","Chinstrap",...: 1 1 1 1 1 1 1 1 1 ...
## $ island       : Factor w/ 3 levels "Biscoe","Dream",...: 3 3 3 3 3 3 3 3 3 ...
## $ bill_length_mm: num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
## $ bill_depth_mm : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
## $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
## $ body_mass_g   : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
## $ sex          : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
## $ year         : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

Zeile 1 gibt Auskunft über GröSSe und die Klasse des Objektes, tibbles sind eine Art data.frame. In den weiteren Zeilen werden die Datentypen bzw. Faktorlevel und die ersten Werte der Spaltenvektoren angezeigt.

4.10.1 Häufigkeit von Factorlevels

Faktoren und die Zuweisung mit der Funktion `factor()` kennen Sie bereits aus dem vorherigen Kapitel. Die Funktion `levels(Faktor)` gibt die möglichen Ausprägungen einer Faktorvariable wieder. Faktoren eignen sich oft besser als Vektoren zum Plotten, Gruppieren oder für Häufigkeitstabellen. Mit der Funktion `count(dat, var)` lassen sich beispielsweise die Häufigkeiten der Levels eines Faktors ausgeben:

```
# Zähle in pingu die Häufigkeiten der Levels des Faktors species:
```

```
count(pingus, island)
```

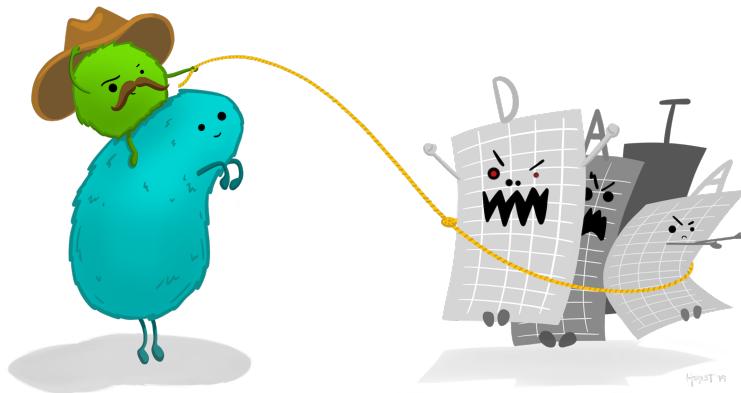
```
## # A tibble: 3 x 2
##   island     n
##   <fct>    <int>
## 1 Biscoe    168
## 2 Dream     124
## 3 Torgersen 52
```

Mit der ersten deskriptiven Statistik zu einem Datensatz in R schlieSSen wir dieses Kapitel ab. Es folgen geschickte Methoden zur Datenaufbereitung.

Kapitel 5

Datenaufbereitung mit **dplyr**





5.1 Einführung

Dplyr ist Teil des *tidyverse* Packages und ermöglicht es, Daten sehr einfach zu manipulieren und in eine Form zu bringen, um diese zu analysieren. Der größte Vorteil dabei ist die einfache Syntax des Packages. Diese ermöglicht es, komplexe Operationen und Umformungen mit relativ wenigen Codezeilen zu realisieren. Um dplyr kennenzulernen, werden wir mit dem Starwars Datensatz arbeiten. Dieser enthält verschiedenen Informationen zu unterschiedlichen Charakteren der Star Wars Saga, wie zum Beispiel das Alter, Geschlecht, Heimatplanet oder Alienrasse. Zunächst lesen wir den Datensatz mit `readRDS()` ein und verschaffen uns dann einen ersten Überblick über den Datensatz:

```
# Der Datensatz befindet sich im Buch im Verzeichnis "Data".
# Daher muss der Pfad beim einlesen des Datensatzes mit angegeben
# werden.

starwars <- readRDS("starwars.RDS") %>% drop_na()
```

Wir benutzen den `head()` Befehl, um uns die ersten 5 Zeilen des Datensatzes anzeigen zu lassen.

```
# Wir lassen uns zunächst die ersten 5 Zeilen des Datensatzes
# ausgeben.

head(starwars, 5)
```

```
## # A tibble: 5 x 11
##   name      height  mass hair_~1 skin_~2 eye_c~3   Age sex
##   <chr>     <int> <dbl> <fct>   <fct>   <fct> <dbl> <fct>
## 1 Luke Sky~    172    77 blond   fair    blue     19 male
## 2 Darth Va~    202   136 none    white   yellow   41.9 male
## 3 Leia Org~    150    49 brown   light   brown    19 fema~
## 4 Owen Lars    178   120 brown,~ light   blue     52 male
## 5 Beru Whi~    165    75 brown   light   blue    47 fema~
## # ... with 3 more variables: gender <fct>, homeworld <chr>,
## #   species <chr>, and abbreviated variable names
## #   1: hair_color, 2: skin_color, 3: eye_color
```

Analog können wir auch mit den Befehl tail(), die letzten n Zeilen eines Datensatzes anzeigen.

```
tail(starwars, 5)
```

```
## # A tibble: 5 x 11
##   name      height  mass hair_~1 skin_~2 eye_c~3   Age sex
##   <chr>     <int> <dbl> <fct>   <fct>   <fct> <dbl> <fct>
## 1 Luminara~    170  56.2 black   yellow  blue     58 fema~
## 2 Barriss ~    166    50 black   yellow  blue     40 fema~
## 3 Dooku        193    80 white   fair    brown   102 male
## 4 Jango Fe~    183    79 black   tan    brown    66 male
## 5 Padmé Am~    165    45 brown   light   brown   46 fema~
## # ... with 3 more variables: gender <fct>, homeworld <chr>,
## #   species <chr>, and abbreviated variable names
## #   1: hair_color, 2: skin_color, 3: eye_color
```

Nachdem wir nun eine erste „Augapfeldiagnostik“ des Datensatzes betrieben haben, sollten wir uns nun die einzelnen Variablen genauer ansehen. Um einen ersten Überblick in die unterschiedlichen Parameter der Variablen zu bekommen, eignet sich der `summary()` Befehl. Dieser berechnet die wichtigstens Kennwerte der im Datensatz enthaltenen Variablen wie Mittelwert, Median, Quantile, Minimum, Maximum und fehlende Werte.

```
summary(starwars)
```

	name	height	mass
##	Length:29	Min. : 88	Min. : 20.00
##	Class :character	1st Qu.:170	1st Qu.: 75.00
##	Mode :character	Median :180	Median : 79.00
##		Mean :178	Mean : 77.77
##		3rd Qu.:188	3rd Qu.: 83.00

```

##                               Max.    :228    Max.    :136.00
##
##      hair_color   skin_color   eye_color
##  none       :9    fair     :7    brown   :10
##  brown      :7    light    :6    blue    : 8
##  black      :6    dark     :2    yellow  : 4
##  blond      :2    orange   :2    hazel   : 2
##  white      :2    pale     :2    orange  : 2
##  auburn, white:1  yellow   :2    black   : 1
##  (Other)     :2    (Other):8    (Other): 2
##      Age                  sex          gender
##  Min.   : 8.00  female      : 6  feminine : 6
##  1st Qu.:31.00 hermaphroditic: 0  masculine:23
##  Median :46.00  male        :23
##  Mean   :51.29  none        : 0
##  3rd Qu.:57.00
##  Max.   :200.00
##
##      homeworld           species
##  Length:29            Length:29
##  Class :character    Class :character
##  Mode   :character    Mode  :character
##
##      
```

Wir sehen nun die unterschiedlichen Verteilungsinformationen jeder Variable. Für `factor` bzw. `character` Variablen, werden hier die Häufigkeiten der einzelnen Kategorien angezeigt.

5.2 dplyr: Die wichtigsten Befehle

Wie am Anfang des Kapitels bereits erwähnt, ermöglicht es `dplyr` mit relativ einfachen Mitteln, komplexe Operationen und Transformationen in Datensätzen vorzunehmen. Hierzu hat `dplyr` eine eigene Syntax entwickelt, die sich sehr stark von der ursprünglichen R-Syntax unterscheidet. Diese baut auf wenigen, relativ intuitiven Befehlen auf, welche verkettet werden können. Zunächst eine Übersicht der wichtigsten Befehle:

- `filter()` – Filtern von Beobachtungen nach einem bestimmten Kriterium
- `arrange()` – Reihen neu Sortieren
- `select()` – Auswahl von Variablen nach deren Name

- `mutate()` – Erstellen von neuen Variablen aus bereits existierenden
- `summarise()` – Viele Werte zu einem einzelnen Wert zusammenfassen
- `group_by()` – Gruppieren von Daten nach bestimmten Variablen in Kombination mit anderen Funktionen

Der vielleicht wichtigste Befehl ist `group_by()`, mit dem die oben genannten Befehle auf einzelne Gruppen innerhalb eines Datensatzes anwendbar sind.

Diese sechs sogennanten " **Verben**" bilden die Grundlage von `dplyr`. Mit ihnen ist es möglich mehrere einfache Operationen miteinander zu verketten, um ein komplexes Ergebnis zu erzielen. Alle Befehle funktionieren auf die gleiche Art und Weise. Jede Operation ist durch die gleiche Struktur gekennzeichnet:

1. Das erste Argument ist ein Dataframe.
2. Die nachfolgenden Argumente beschreiben, was mit dem Dataframe geschehen soll, wobei die Variablennamen (ohne Anführungszeichen) verwendet werden.
3. Das Ergebnis ist ein neuer Dataframe

5.3 Beispiel: Filtern von Beobachtungen mit `filter()`

Um effektiv nach bestimmten Werten zu filtern, müssen für jede Operation die Kriterien, nach denen gefiltert werden soll, definiert werden. Dies geschieht mit Hilfe der bereits eingeführten logischen Operatoren. Im ersten Beispiel sollen alle Beobachtungen, in welchen die Variablen `height` und `mass` größer als 190 bzw. 90 sind gefiltert werden:

Es werden hier nun alle Helden aus dem Datensatz ausgegeben, die größer als 1,90 m und schwerer als 90 Kilogramm sind

```
filter(starwars, height > 190 & mass > 90)
```

```
## # A tibble: 2 x 11
##   name      height  mass hair_~1 skin_~2 eye_c~3  Age sex
##   <chr>     <int> <dbl> <fct>   <fct>   <fct> <dbl> <fct>
## 1 Darth Va~    202    136 none     white    yellow   41.9 male
## 2 Chewbacca    228    112 brown   unknown   blue    200   male
## # ... with 3 more variables: gender <fct>, homeworld <chr>,
## #   species <chr>, and abbreviated variable names
## #   1: hair_color, 2: skin_color, 3: eye_color
```

5.3.1 Filtern von Strings / Factors

Logische Operatoren lassen sich sehr gut auf numerische / bzw. kontinuierliche Variablen anwenden, um diese nach bestimmten Kriterien zu filtern. Wenn mit Strings oder Factors gearbeitet wird, sucht man häufig nach bestimmten **pattern** in den Strings, wie hier bei den Namen.

type	food	site
otter	urchin	bay
Shark	seal	channel
otter	abalone	bay
otter	crab	wharf

Sollen nun alle Beobachtungen mit „Skywalker“ im Namen gefiltert werden, kann die **grep1()** Funktion aus R genutzt werden. Diese prüft, ob eine Zeichenfolge vorhanden ist oder nicht und gibt dann entsprechend TRUE oder FALSE aus, was ein filtern ermöglicht. Dies ist vor allem bei Strings die aus mehr als einem Wort bestehen und durch ein Leerzeichen getrennt sind, sehr praktisch. Bei Strings die nur aus einem Wort bestehen oder Factors, kann auch mit einem einfach == Vergleich gearbeitet werden.

Beispiel:

```
# Hier werden alle Beobachtungen mit der Spezies "Human"
gefiltert.

filter(starwars, species == "Human")
```

```
## # A tibble: 18 x 11
##   name      height  mass hair_~1 skin_~2 eye_c~3   Age sex
##   <chr>     <int> <dbl> <fct>    <fct>    <fct>  <dbl> <fct>
## 1 Luke Sk~     172    77  blond    fair     blue      19 male
```

```

## 2 Darth V~    202 136    none   white   yellow   41.9 male
## 3 Leia Or~   150  49    brown   light   brown    19  fema~
## 4 Owen La~   178 120    brown,~ light   blue     52  male
## 5 Beru Wh~   165  75    brown   light   blue     47  fema~
## 6 Biggs D~   183  84    black   light   brown    24  male
## 7 Obi-Wan~   182  77    auburn~ fair   blue-g~  57  male
## 8 Anakin ~   188  84    blond   fair   blue     41.9 male
## 9 Han Solo   180   80    brown   fair   brown    29  male
## 10 Wedge A~  170   77    brown   fair   hazel    21  male
## 11 Palpati~  170   75    grey    pale   yellow   82  male
## 12 Boba Fe~  183 78.2   black   fair   brown   31.5 male
## 13 Lando C~  177   79    black   dark   brown    31  male
## 14 Lobot    175   79    none    light   blue     37  male
## 15 Mace Wi~  188  84    none    dark   brown    72  male
## 16 Dooku    193   80    white   fair   brown   102  male
## 17 Jango F~  183   79    black   tan    brown    66  male
## 18 Padm  A~  165   45    brown   light   brown    46  fema~
## # ... with 3 more variables: gender <fct>, homeworld <chr>,
## #   species <chr>, and abbreviated variable names
## #   1: hair_color, 2: skin_color, 3: eye_color

```

Analog hierzu kann auch nach bestimmten Faktorstufen gefiltert werden:

```
filter(starwars, sex == "male")
```

```

## # A tibble: 23 x 11
##   name      height  mass hair_~1 skin_~2 eye_c~3   Age sex
##   <chr>     <int> <dbl> <fct>   <fct>   <dbl> <fct>
## 1 Luke Sk~    172    77 blond   fair    blue     19  male
## 2 Darth V~   202 136    none   white   yellow   41.9 male
## 3 Owen La~   178 120    brown,~ light   blue     52  male
## 4 Biggs D~   183  84    black   light   brown    24  male
## 5 Obi-Wan~   182  77    auburn~ fair   blue-g~  57  male
## 6 Anakin ~   188  84    blond   fair   blue     41.9 male
## 7 Chewbac~  228 112 brown  unknown blue    200  male
## 8 Han Solo   180   80    brown   fair   brown    29  male
## 9 Wedge A~   170   77    brown   fair   hazel    21  male
## 10 Palpati~  170   75    grey    pale   yellow   82  male
## # ... with 13 more rows, 3 more variables: gender <fct>,
## #   homeworld <chr>, species <chr>, and abbreviated
## #   variable names 1: hair_color, 2: skin_color,
## #   3: eye_color

```

Um komplexere Strings zu filtern, kann die grepl() Funktion integriert werden. Hier werden alle Beobachtungen gefiltert, welche innerhalb der Variable "name" den String "Skywalker" enthalten. So erhalten wir alle Skywalker Charaktere, die im Datensatz vorhanden sind:

```
filter(starwars, grepl("Skywalker", name))
```

```
## # A tibble: 2 x 11
##   name      height  mass hair_~1 skin_~2 eye_c~3   Age sex
##   <chr>     <int> <dbl> <fct>    <fct>    <fct> <dbl> <fct>
## 1 Luke Sky~    172    77 blond    fair     blue     19 male
## 2 Anakin S~    188    84 blond    fair     blue    41.9 male
## # ... with 3 more variables: gender <fct>, homeworld <chr>,
## #   species <chr>, and abbreviated variable names
## #   1: hair_color, 2: skin_color, 3: eye_color
```

5.4 dplyr: Der Piping Operator %>%

Die vielleicht wichtigste Funktion in **dplyr** ist der sogenannte „piping operator“ `%>%`. Mit diesem können beliebig viele Befehle kombiniert, oder auch „verkettet“ werden, um einen Datensatz umzuformen. Hierbei bleiben die oben vorgestellten Prinzipien gültig:

1. Das erste Argument ist ein Dataframe.
2. Die nachfolgenden Argumente beschreiben, was mit dem Dataframe geschehen soll, wobei die Variablennamen (ohne Anführungszeichen) verwendet werden.
3. Das Ergebnis ist ein neuer Dataframe

In diesem Kapitel werden nur die basis **dplyr**-Funktionen besprochen und wie diese in einer Pipeline integriert werden können. Prinzipiell lassen sich jedoch auch alle anderen R-Befehle in eine „Pipeline“ integrieren, wie zum Beispiel statistische Transformationen. Es spielt keine Rolle, welche Befehle innerhalb einer Pipeline ausgeführt werden, solange die oben genannten Prinzipien eingehalten werden.

Hier ein sehr fortgeschrittenes Beispiel, wie dies aussehen kann. Es wurden in diesem Beispiel Funktionen aus unterschiedlichen Paketen verwendet (z.B. `fisherz()` aus dem `psych` Package), als auch Funktionen von R (`cor()`) um aus einem sehr großen Datensatz mit 1 Millionen Beobachtungen, mittlere Korrelationen zwischen unterschiedlichen Variablen zu berechnen:

```
# df_clean %>% group_by(N, K, Retrievals) %>%
# Hier wird der Datensatz nach den Variablen N, K und Retrievals
# gruppiert.

# summarise(corrA = cor(mu_est_a, mu_real_a),
#            corrC = cor(mu_est_c, mu_real_c)) %>%
# Dann werden die Korrelationen zwischen den Variablen est_a und
# real_a,
# so wie die Korrelationen zwischen den Variablen est_c und
# real_c für alle Gruppenkombinationen berechnet.

# mutate(z_a = fisherz(corrA), z_c = fisherz(corrC)) %>%
# Anschließend werden diese Korrelationen z-transformiert und dann
# die Mittelwerte nach den Variablen N und K berechnet

# filter(Retrievals == 100) %>%
# group_by(N, K) %>%
# summarise(mean_a_100 = mean(z_a),
#           mean_c_100 = mean(z_c),
#           range_corr = range(mean_a_100),
#           range_corr = range(mean_a_100)) %>%
# mutate(meanCorrA_100 = fisherz2r(mean_a_100),
#        meanCorrC_100 = fisherz2r(mean_c_100)) %>%
# select(-c(mean_a_100, mean_c_100))
```

5.5 Beispiel

Um nun verschiedene Operationen zu einer Pipeline zu verketteten, können wir zwischen die einzelnen `dplyr` Befehle den Piping Operator `%>%` schalten.

Soll zum Beispiel der Mittelwert des Alters der Helden aus dem Starwars Datensatz für verschiedene Gruppen (hier Heimatwelten und Spezies) berechnet werden, können hierfür der `group_by()` und der `summarise()` Befehl kombiniert werden. Zunächst Gruppieren wir den Datensatz nach den Variablen `species` und `homeworld` und berechnen dann mit `summarise()` eine neue Variable `mean_Age` für jede Gruppenkombination. Hierbei wird innerhalb von `summarise()` der Mittelwert der `Age` Variablen berechnet und in der neuen Variablen `mean_Age` gespeichert. Hierbei gelten die oben genannten Prinzipien, welche dann mit `%>%` verkettet werden:

1. Das erste Argument ist ein Dataframe: `starwars`

2. Die nachfolgenden Argumente beschreiben, was mit dem Dataframe geschehen soll:

```
group_by(species, homeworld) und summarise(mean_Age = mean(Age))
```

3. Das Ergebnis ist ein neuer Dataframe, der nur noch die Gruppenvariablen `Homeworld` und `Species`, sowie die neu berechnete Variable `mean_Ages` enthält:

```
# Dazu benutzen wir den Piping Operator %>%, um die Befehle zu verketteten:
```

```
starwars %>%
  group_by(species, homeworld) %>%
  summarise(mean_Age = mean(Age))
```

```
## `summarise()` has grouped output by 'species'. You can
## override using the ` `.groups` argument.
```

```
## # A tibble: 21 x 3
## # Groups:   species [11]
##   species homeworld   mean_Age
##   <chr>    <chr>        <dbl>
## 1 Cerean   Cerea        92
## 2 Ewok     Endor         8
## 3 Gungan   Naboo        52
## 4 Human    Alderaan     19
## 5 Human    Bespin        37
## 6 Human    Concord Dawn 66
## 7 Human    Corellia      25
## 8 Human    Haruun Kal   72
## 9 Human    Kamino       31.5
## 10 Human   Naboo        64
## # ... with 11 more rows
```

Natürlichsprachlich dargestellt werden also folgende Operationen ausgeführt:

1. Nehme den Datensatz `starwars` (1. Zuerst der Dataframe):

```
starwars %>%
```

2. Gruppiere diesen nach Spezies und Heimatwelt (1. Verarbeitungsschritt):

```
group_by(species, homeworld) %>%
```

3. Berechne dann für jede dieser Gruppen den Mittelwert für die Variable `Age` (2. Schritt):

```
summarise(meanAge = mean(Age))
```

Da nun der Piping-Operator die einzelnen Elemente miteinander verkettet, ist es nicht mehr notwendig, den Datensatz innerhalb der einzelnen Befehle als Argument anzugeben, so wie im Beispiel von `filter()`. Es muss lediglich der Ausgangsdatensatz am Anfang der Pipeline angeben werden.

5.6 dplyr: Neue Variablen mit `mutate()` berechnen



Der letzte wichtige Befehl in `dplyr` ist `mutate()` bzw. `across()`. Mit `mutate()` bzw. `across()` ist es möglich eine Variable bzw. mehrere Variablen umzuformen, oder neu zu berechnen. Dies wird hier anhand einer z -Transformation erläutert. Dies ermöglicht der Befehl `scale()`, der standardmäßig in R vorhanden ist.

5.7 Beispiel

```

starwars %>%
  select(height, mass) %>%
  mutate(z_height = scale(height),
        z_mass = scale(mass))

## # A tibble: 29 x 4
##   height  mass z_height[,1] z_mass[,1]
##   <int> <dbl>     <dbl>      <dbl>
## 1    172    77     -0.265    -0.0335
## 2    202   136      1.07     2.52
## 3    150    49     -1.24     -1.25
## 4    178   120      0.00153   1.83
## 5    165    75     -0.576    -0.120
## 6    183    84      0.224     0.270
## 7    182    77      0.179    -0.0335
## 8    188    84      0.446     0.270
## 9    228   112      2.22     1.48
## 10   180    80      0.0904   0.0965
## # ... with 19 more rows

starwars %>%
  select(height, mass) %>%
  mutate(across(c(height, mass), list(z = scale)))

## # A tibble: 29 x 4
##   height  mass height_z[,1] mass_z[,1]
##   <int> <dbl>     <dbl>      <dbl>
## 1    172    77     -0.265    -0.0335
## 2    202   136      1.07     2.52
## 3    150    49     -1.24     -1.25
## 4    178   120      0.00153   1.83
## 5    165    75     -0.576    -0.120
## 6    183    84      0.224     0.270
## 7    182    77      0.179    -0.0335
## 8    188    84      0.446     0.270
## 9    228   112      2.22     1.48
## 10   180    80      0.0904   0.0965
## # ... with 19 more rows

```

In diesem Beispiel wurden zunächst nur `height` und `mass` mit dem `select()` Befehl ausgewählt, daher werden auch nur diese beiden Spalten am Ende der

Pipline im Datensatz angezeigt. Dies kann hilfreich sein, wenn man einen Datensatz mit sehr vielen Variablen analysieren muss, von denen nur einige wenige interessant sind. Dies ist z.B. bei Fragebögen der Fall, die unterschiedliche Fächer erfassen.

Der nächste Befehl `mutate()` besteht immer aus einer Operation, die mit einer Spalte im Datensatz durchgeführt wird. Im Beispiel oben werden die Spalten `z_height` und `z_mass` berechnet, die sich jeweils aus `scale(SPALTENAME)` zusammensetzen und die z-Werte der jeweiligen Variablen ausgeben.

Anstatt beide Variablen einzeln zu transformieren, kann den Befehl `scale()` auch direkt auf mehrere Spalten angewendet werden. Dazu kann der `across()` Befehl verwendet werden.

`dplyr::across()`

use within `mutate()` or `summarize()` to apply function(s) to a selection of columns!

EXAMPLE:

```
df %>%
  group_by(species) %>%
  summarise(
    across(where(is.numeric), mean)
  )
```

species	mass_g	age_yr	range_sqmi
pika	163	2.4	0.40
marmot	1509	3.0	0.87
marmot	2417	5.6	0.62

@allison_horst

Hier muss innerhalb von `mutate()` einfach mit `across(c(SPALTE1, SPALTE2))` ein Vektor der zu transformierenden Spalten übergeben werden, sowie die Funktion(en), welche auf die Spalten angewandt werden soll. Dies muss dann wie folgt definiert werden:

```
mutate(across(c(height, mass), list(z = scale)))
```

Diese Schreibweise hat den Vorteil, dass:

1. In der `list()` mehrere Befehle übergeben werden können
2. Die Originalspalten beibehalten werden
3. Den neuen Spalten direkt ein Suffix zugewiesen werden kann.
Dieses wird automatisch als „_suffix“ an die neue Variable angehängt.

```
starwars %>% mutate(across(c(height, mass), list(z = scale)))
```

```
## # A tibble: 29 x 13
##   name      height  mass hair_~1 skin_~2 eye_c~3   Age sex
##   <chr>     <int> <dbl> <fct>  <fct>  <fct>  <dbl> <fct>
## 1 Luke Sk~    172    77 blond   fair    blue     19 male
## 2 Darth V~    202   136 none    white   yellow   41.9 male
## 3 Leia Or~    150    49 brown   light   brown    19 fema~
## 4 Owen La~    178   120 brown,~ light   blue     52 male
## 5 Beru Wh~    165    75 brown   light   blue     47 fema~
## 6 Biggs D~    183    84 black   light   brown    24 male
## 7 Obi-Wan~    182    77 auburn~ fair    blue-g~  57 male
## 8 Anakin ~    188    84 blond   fair    blue     41.9 male
## 9 Chewbac~    228   112 brown   unknown blue    200 male
## 10 Han Solo   180    80 brown   fair    brown    29 male
## # ... with 19 more rows, 5 more variables: gender <fct>,
## #   homeworld <chr>, species <chr>, height_z <dbl[,1]>,
## #   mass_z <dbl[,1]>, and abbreviated variable names
## #   1: hair_color, 2: skin_color, 3: eye_color
```

Kapitel 6

Deskriptive Statistik

Deskriptive Statistik wird benutzt, um Daten anhand von Kennzahlen zu beschreiben. Diese Kennzahlen werden auch häufig *statistische MaSSe* oder *statistische Parameter* genannt. Vor allem bei umfangreichen Datensätzen können deskriptive Statistiken helfen einen Überblick über die vorhandene *Datenstruktur* zu gewinnen.

Zu den gängigsten (deskriptiven) statistischen MaSSen zählen sogenannten *LagemaSSe*, wie der **Mittelwert** oder der **Median**, welche MaSSe für die zentrale Tendenz einer Häufigkeitsverteilung darstellen (z.B. der Werte in einer Variable).

Andere statistische MaSSe, wie die **Standardabweichung** oder die **Varianz**, erlauben dagegen Aussagen über die Streuung der Werte in einer Variable (so genannte *StreuungsmaSSe*).

Es gibt andere deskriptive MaSSe, die Aussagen über Zusammenhänge zwischen verschiedener Variablen in einem Datensatz geben (sogenannte *ZusammenhangsmaSSe*). Hierzu zählt beispielsweise die **Korrelation**.

Häufig kommen deskriptive Statistiken im Rahmen von einer **Explorativen Datenanalyse** (EDA) zum Einsatz. EDA stellt häufig dein Einstieg in einer meistens komplexeren und (häufig) theoriegeleitete Datenanalyse. EDA wird auch häufig dazu verwendet die Qualität und Struktur der Daten zu untersuchen.

In diesem Kapitel berechnen wir verschiedene statistische MaSSe, um einen Überblick (oder einen ersten Eindruck) der Daten in einem Datensatz zu gewinnen.

6.1 Allgemeine Informationen eines Datensatzes

Für den folgenden Abschnitt benötigen Sie den `iris` Datensatz. Der `iris` Datensatz ist Teil von der Basisinstallation von R und wird automatisch (im Hintergrund) geladen. D.h. Sie können auf den Datensatz zugreifen in dem Sie `iris` in die R-Konsole eingeben.

```
# als erstes speichern wir den iris-Datensatz in einer neuen
# Variable names 'data'
data <- iris
```

Benutzen Sie `head(data)`, um die ersten 5 Zeilen des Datensatzes zu sehen.

```
head(data)
```

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

Eine weitere Funktion, mit der Sie die Struktur eines Datensatzes untersuchen können ist `str()`. `str()` steht für *structure*. Der output dieser Funktion zeigt Basisinformationen des Datensatzes, wie die Anzahl und Beobachtungen (d.h., `obs.` für *observations*) und Variablen (d.h., `variables`). Wie Sie wissen, stehen Beobachtungen in den Zeilen und Variablen in den Spalten eines Datensatzes.

```
# untersuchen wir die Struktur des Datensatzes
str(data)
```

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

`str()` erlaubt erste Einblicke in den Datensatz. Oben sehen Sie einige Informationen, wie beispielsweise Name (z.B., `Sepal.Length`), Variablen-Typ (z.B., `num`

für numerisch), sowie typische Werte, die die verschiedene Variablen im Datensatz annehmen können (z.B., 5.1 4.9 4.7 4.6).

Der Datensatz erhält also 150 Beobachtungen und 5 Variablen. Es handelt sich um die Länge und Breite des Kelch- und des Blütenblattes sowie die Art von 150 Blumen. Länge und Breite des Kelch- und Blütenblattes sind numerische Variablen. Die Art der Blume ist ein Faktor mit 3 Stufen (d.h., **Factor w/ 3 levels**).

6.2 LagemaSSe

6.2.1 Minimum und Maximum

Minimum und Maximum können mit den Funktionen `min()` und `max()` ermittelt werden:

```
# Berechnen wir das Minimum für die Variable Sepal.Length
min(data$Sepal.Length)
```

```
## [1] 4.3
```

```
# Berechnen wir das Maximum für die Variable Sepal.Width
max(data$Sepal.Width)
```

```
## [1] 4.4
```

Alternativ, können Sie `range()` (Spannweite) benutzen, um Minima und Maxima zu berechnen:

```
# Berechnen wir die Spannweite von Sepal.Width und speichern
# wir das Ergebnis in einer neuen Variable namens "width_range"
width_range <- range(data$Sepal.Width)
width_range
```

```
## [1] 2.0 4.4
```

Wie Sie sehen, berechnet `range()` einen Vektor, der das Minimum und das Maximum (in dieser Reihenfolge) enthält. Genau genommen müssen Sie diese Werte von einander abziehen, um die Spannweite von `Sepal.Width` zu bekommen:

```
# max - min
width_range[2] - width_range[1]
```

```
## [1] 2.4
```

Die Spannweite beträgt also 2.4. Dies bedeutet, dass der Unterschied zwischen dem breitesten und dem dünnsten Kelchblatt 2.4 cm beträgt.

6.2.2 Quantile und Perzentile

Andere häufig eingesetzte deskriptive Parameter sind Quantile oder Perzentile (d.h. welche Werte sind < 25%, < 50%, < 75%, oder > 75%)

Ein häufig verwendetes Perzentil ist der **Median**. Der Median ist äquivalent zum 50. Perzentil (d.h. 50%) einer Variable, weil 50% der Werte einer Variable unterhalb/oberhalb dieses Wertes liegen.

Der Median kann folgendermaßen berechnet werden:

$$(1) \quad x = \begin{cases} x_{m+1} & \text{für ungerades } n \\ \frac{1}{2}(x_m + x_{m+1}) & \text{für gerades } n \end{cases} \quad = 2m + 1 \quad = 2m$$

Ein Wert m ist Median einer Stichprobe, wenn mindestens die Hälfte der Stichprobenelemente nicht größer als m und mindestens die Hälfte nicht kleiner als m ist.

Sortiert man die Beobachtungswerte der GröSSe nach, so entspricht der Median bei einer ungeraden Anzahl von Beobachtungen dem Wert der in der Mitte dieser Folge liegenden Beobachtung. Bei einer geraden Anzahl von Beobachtungen gibt es kein einzelnes mittleres Element, sondern zwei. Der Median entspricht dann (vereinfacht ausgedrückt), dem Mittelwert dieser beiden Beobachtungen.

Sie können diese Annahme mit den Funktionen `median()` und `quantile()` überprüfen.

```
# median
median(data$Sepal.Length)
```

```
## [1] 5.8
```

```
# 0.5 steht für 50%
quantile(data$Sepal.Length, 0.5)
```

```
## 50%
## 5.8
```

Sie können `quantile()` auch benutzen, um unterschiedliche Perzentile auf einmal zu berechnen.

```
# mit 0.25, 0.5, und 0.75 können wir die Grenzen für die
# verschiedenen Quartile berechnen
quantile(data$Sepal.Length, c(0.25, 0.5, 0.75))
```

```
## 25% 50% 75%
## 5.1 5.8 6.4
```

Dies bedeutet, dass 25% der Werte sich unterhalb von 5.1, 50% unterhalb von 5.8 und 75% unterhalb 6.4 befinden. Im Umkehrschluss befinden sich 25% der Werte oberhalb 6.4.

6.2.3 Interquartilsabstand (IQA)

Manchmal möchte man wissen wie groß der Abstand zwischen bestimmten Quantilen ist, um Aussagen über Beobachtungen, die innerhalb oder außerhalb eines bestimmten Wertebereichs liegen, zu treffen. Häufig wird hierfür der Interquartilsabstand benutzt, welches als der Abstand zwischen den 75. und 25. Perzentil definiert ist.

```
# IQR steht für interquartile range
IQR(data$Sepal.Length)
```

```
## [1] 1.3
```

Sie können dies überprüfen, indem Sie den IQA selbst berechnen.

$$(2) \quad \text{IQA} = X_{0.75} - X_{0.25}$$

Sie müssen dafür den 3. Quartil (den 75. Perzentil) und den 1. Quartil (den 25. Perzentil) „händig“ berechnen und diese anschließend von einander abziehen.

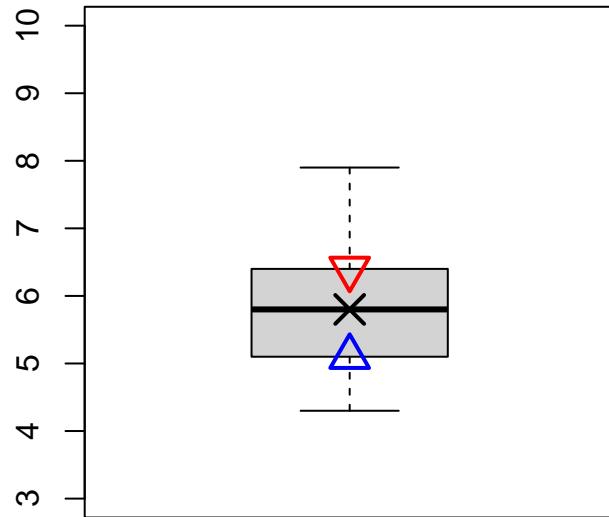
```
as.numeric(quantile(data$Sepal.Length, 0.75) -
            quantile(data$Sepal.Length, 0.25))
```

```
## [1] 1.3
```

Den IQA kann ebenfalls anhand eines Boxplots dargestellt werden. Unten sehen Sie das 75. Perzentil (nach unten zeigenden Dreieck), den Median (ein X) und den 25. Perzentil (nach oben zeigenden Dreieck) dargestellt. Den grau schattierten Bereich um den Median, zeigt der Wertebereich, in dem sich 50% der Werte einer Verteilung befinden.

```
# IQA visualisieren anhand eines Box-Plots

# box-plot
boxplot(data$Sepal.Length, ylim = c(3, 10))
# 0.75 Perzentil
points(quantile(data$Sepal.Length, 0.75),
       col = 'red', cex = 2, pch = 6, lwd = 2)
# 0.50 Perzentil (Median)
points(quantile(data$Sepal.Length, 0.50),
       col = 'black', cex = 2, pch = 4, lwd = 2)
# 0.25 Perzentil
points(quantile(data$Sepal.Length, 0.25),
       col = 'blue', cex = 2, pch = 2, lwd = 2)
```



6.2.4 Mittelwert

Ähnlich zum Median, stellt der Mittelwert ein wichtiges MaSS für die zentralen Tendenzen einer Variable dar. Der Mittelwert (i.S. des arithmetischen Mittels) ist die Summe der gegebenen Werte geteilt durch die Anzahl der Werte.

$$(3) \quad x = \frac{1}{n} \left(\sum_{i=1}^n x_i \right) = \frac{x_1 + x_2 + \dots + x_n}{n}$$

Sie können den Mittelwert also folgendermaSSen berechnen.

```
summe_der_werte <- sum(data$Sepal.Width)
anzahl_der_werte <- length(data$Sepal.Width)

# Mittelwert von Sepal.Width
summe_der_werte / anzahl_der_werte
```

```
## [1] 3.057333
```

Wir können diese Berechnung mit der R-Funktion `mean()` überprüfen.

```
mean(data$Sepal.Width)
```

```
## [1] 3.057333
```

Bitte beachten Sie, dass, sollte eine Variable mindestens einen fehlenden Wert erhalten, sollten Sie `mean(data$Sepal.Width, na.rm = TRUE)`, um den Mittelwert unter Ausschluss der fehlenden Werte zu berechnen. Das Argument `na.rm = TRUE` kann für die meisten in diesem Abschnitt vorgestellten Funktionen verwendet werden, nicht nur für den Mittelwert.

6.3 StreuungsmaSSe

6.3.1 Standardabweichung

Die Standardabweichung ist ein MaSS für die Breite der Streuung der Werte einer Variable rund um deren Mittelwert (arithmetisches Mittel). Vereinfacht gesagt, ist die Standardabweichung die durchschnittliche Entfernung aller gemessenen Ausprägungen eines Merkmals vom Durchschnitt.

Sie kann folgender MaSSen berechnet werden:

$$(4) \quad s_N = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2$$

In R also:

```
sqrt(sum((data$Sepal.Width - mean(data$Sepal.Width))^2) / (length(data$Sepal.Width) - 1))
```

```
## [1] 0.4358663
```

Oder mit Hilfe der R-Funktion `sd()`:

```
sd(data$Sepal.Width)
```

```
## [1] 0.4358663
```

Kapitel 7

Graphiken mit ggplot2



Artwork by Allison Horst

7.1 Exkurs: Warum viridis?

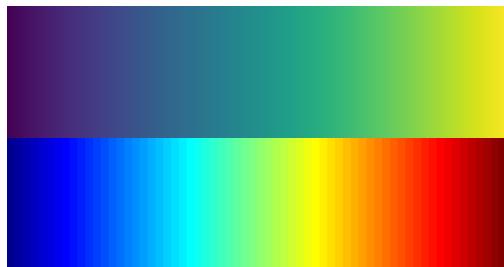
Das **viridis** Paket in R stellt eine Reihe an Colormaps bereit, die folgenden Anspruch an sich stellen:

- gute **Lesbarkeit** und Unterscheidbarkeit in Graphiken
- auch bei **Farbenblindheit** oder **Farbsehschwäche**
- bleibt bei **Grauskala**(-druck) erhalten

Praktisch bedeutet das, die Farbkarten sind:

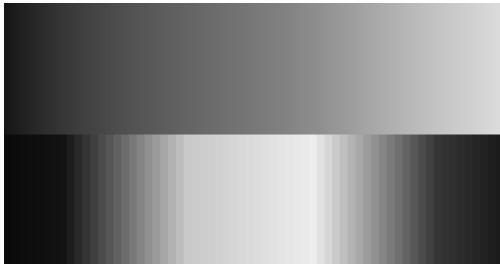
- **farbenfroh**: Sie umspannen weite Teile der Farbpalette, damit Unterschiede einfach zu erkennen sind
- **einheitlich**: Nah beieinander liegende Punkte haben ähnliche Farben und weit auseinander liegende Punkte haben stark unterschiedliche Farben – und das möglichst konsistent über den gesamten Farbraum hinweg
ein fester Abstand wird also an verschiedenen Stellen im Farbraum als perzeptuell gleich wahrgenommen, d.h. keine mal schnellen mal langsamen Farbton- oder Helligkeitsänderungen

Gerade die Einheitlichkeit ist wichtig, sie sorgt dafür, dass keine Gebiete überbetont werden, während Kontrast an anderen Stellen nicht gegeben ist. Hier die **viridis** Farbskala oben und **jet** (langjährig die Standardfarben in Matlab) unten:



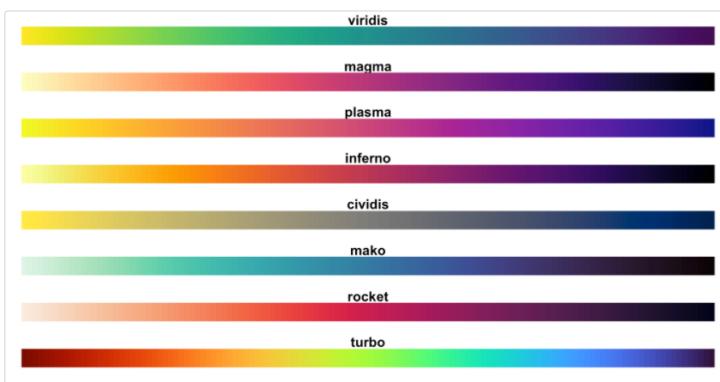
Bei **jet** stechen besonders Gelb und Türkis heraus – in einer Graphik gibt das Artefakte durch die Darstellung, weil unauffällige Daten durch die Farbwahl auf einmal außergewöhnlich wirken. Außerdem werden an anderen Stellen Unterschiede in den Daten dadurch unterschlagen, dass in weiten Teilen des roten und blauen Spektrums in **jet** fast keine Farbstufen enthalten sind. **viridis** versucht dieses Problem zu beheben und wirkt homogen – ohne „Spitzen und Täler“.

Zusätzlich muss berücksichtigt werden, dass Graphiken auch bei Grauskaladruck noch interpretierbar bleiben. Das macht eine Grauskala aus **viridis** und **jet**:



Bei der oberen `viridis` Skala vermindert sich die Interpretierbarkeit nur minimal, während man bei der unteren Skala nicht mehr zwischen hohen und niedrigen Werten unterscheiden kann. Tatsächlich wirkt es so, als ob die mittleren Bereiche besonders auffällig wären – Bereiche die meistens um die 0 herum liegen!

Im `viridis` Paket gibt es 8 Colormaps, aus denen man wählen kann und die alle diese Kriterien erfüllen:



source: `viridis` Dokumentation

7.2 ggplot2 – Einführung

`ggplot2` ist ein Paket zur graphischen Darstellung in R, das auf dem Buch „The Grammar of Graphics“ von Leland Wilkinson basiert. Wie bei `dplyr` ist die Codestruktur an Grammatik orientiert und somit einfach nachzuvollziehen. Ablauf der Graphikerstellung in `ggplot2` lässt sich wie folgend zusammenfassen:

- Daten bereitstellen
- Wie werden Variablen in der Graphik in Ästhetik (aesthetic mappings) umgewandelt? – `aesthetics()`
 - Was ist die x-Achse, was die y-Achse?

- Wonach wird farbkodiert?
- ...
- Welche graphische Darstellung?
 - Scatterplot – `geom_point()`
 - Histogram – `geom_histogram()`
 - Linienplot – `geom_line()`
 - ...
- Ggf. weitere Details definieren, wie z.B. die Achsenlabel

Um eine Graphik zu „bauen“ kann man also dieser Struktur folgen:

```
ggplot(data = DATA, aes(MAPPINGS)) +
  GEOM_FUNCTION (mapping = aes(MAPPINGS), position =
  POSITION) +
  LABEL_FUNCTION +
  THEME_FUNCTION +
  COORDINATE_FUNCTION +
  SCALE_FUNCTION +
  FACET_FUNCTION
```

Dabei sind DATA, MAPPINGS und GEOM_FUNCTION notwendig, während der Rest nicht notwendig ist, da es gute Standardeinstellungen gibt.

Vorsicht: In `ggplot2` sind Lagen mit `+` verknüpft und nicht mit `%>%!`

source: ggplot2 Cheat Sheet

7.3 Vor der Visualisierung: Die Daten

Das Kapitel arbeitet mit einem Datensatz, der die Superhelden von Marvel und DC miteinander vergleicht. Die Daten finden sich auf GitHub („<https://github.com/cosmoduende/r-marvel-vs-dc>“), sind aber auf zwei Files aufgeteilt, die erst einzeln geladen und dann zusammengeführt werden müssen.

Erst werden die zwei Files von GitHub heruntergeladen und den Variablen `MarvelCharacters` und `MarvelStats` zugewiesen. Dabei werden fehlende Werte als `NA` kodiert:

```
# Lade Daten aus zwei unterschiedlichen Files
MarvelCharacters <-
  read_csv("https://raw.github.com/cosmoduende/r-marvel-vs-dc/main/dataset_shdb/heroesInformation.csv",
           na = c("-", "-99.0"))

## Warning: One or more parsing issues, call `problems()` on your data
## frame for details, e.g.:
##   dat <- vroom(...)
##   problems(dat)

MarvelStats <-
  read_csv("https://raw.github.com/cosmoduende/r-marvel-vs-dc/main/dataset_shdb/charactersStatistics.csv")
```

Dann werden nur die Charaktere aus Marvel und DC ausgewählt und die `MarvelCharacters`-Daten werden mit `dplyr` vorbearbeitet:

```
# Preprocessing der Daten `MarvelCharacters` 
marvelDcInfo <- MarvelCharacters %>%
  rename(Name = name) %>% # `name` -> `Name`
  filter(Publisher == "Marvel Comics" | # nur Marvel oder...
         Publisher == "DC Comics") %>% # ... DC
  filter(!duplicated(Name)) %>% # Duplikate löschen
  # nur bestimmte Spalten (es gibt z.B. auch Augenfarbe)
  select(Name, Gender, Race, Publisher, Weight, Height)
```

Dann werden die beiden Datensätze zusammengefügt:

```
# Zusammenführen der beiden Datensätze
MarvelDC <- inner_join(marvelDcInfo, MarvelStats, by = "Name")

# einige Zeilen sind falsch kodiert und enthalten eine andere
# Skala,
# deswegen werden alle Zeilen mit Intelligence = 1
# ausgeschlossen.
MarvelDC <- MarvelDC %>%
  filter(Intelligence > 1)
```

Damit ist der Datensatz `MarvelDC` entstanden, mit dem wir weiterarbeiten können!

Um `ggplot` Daten zu geben, kann man sie entweder in den Klammern definieren:

```
ggplot(data = MarvelDC)    # "data = " kann auch weggelassen werden
```

Oder man gibt sie via Pipeing an `ggplot` weiter – beides ist äquivalent:

```
MarvelDC %>%
  ggplot()
```

Dieser Code funktioniert allerdings noch nicht, da wir bisher nur die Daten definiert haben, aber noch keine MAPPINGS und auch keine GEOM_FUNCTION – die beiden anderen erforderlichen Angaben.

7.4 Ästhetische Mapping

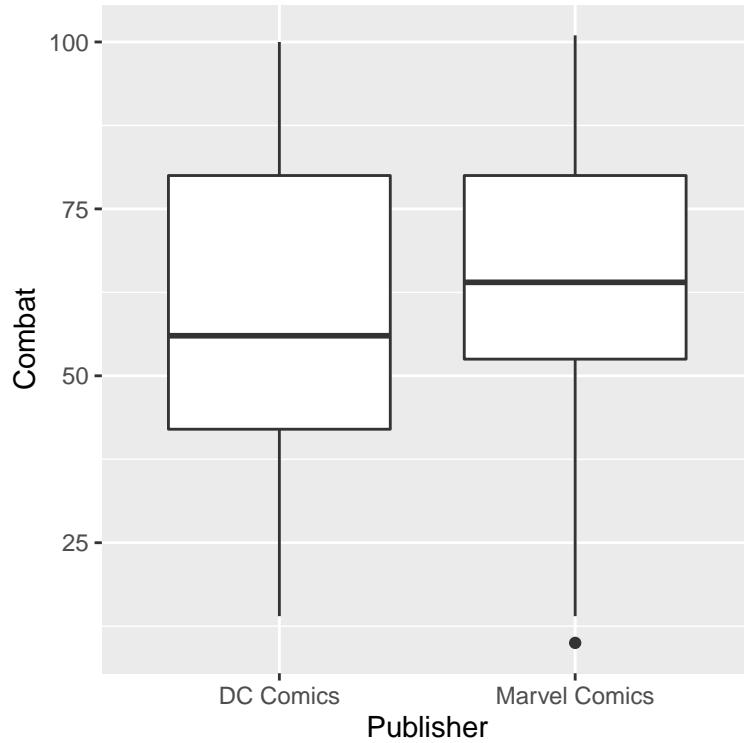
Das ästhetische Mapping (`aesthetics`) kann entweder direkt in `ggplot` definiert werden, dann gilt es für **alle** weiteren Schritte, oder es kann in der jeweiligen graphischen Funktion definiert werden, dann gilt es nur für diese `geom_*` Funktion.

<code>aesthetics</code>	Bedeutung
<code>x</code>	Welche Variable soll auf die x-Achse?
<code>y</code>	Welche Variable soll auf die y-Achse?
<code>shape</code>	Nach den Werten welcher Variable soll die Form vergeben werden?
<code>color</code>	Nach den Werten welcher Variable soll die Farbe vergeben werden?
<code>fill</code>	Nach den Werten welcher Variable soll die Füllfarbe vergeben werden?

`aesthetics` in `ggplot()` werden an alle `geom_*` weitergegeben. `aesthetics` in `geom_*` sind spezifisch für diese graphische Funktion (und können geerbte Werte überschreiben).

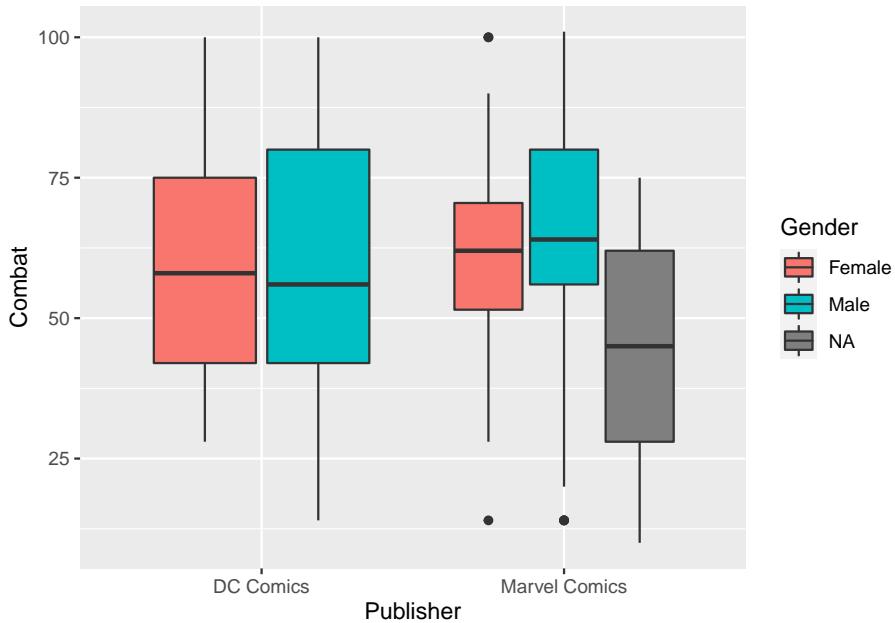
Hier ein Beispiel, das sich wie folgend liest: „Nehme MarvelDC als Daten. Die x-Achse soll `Publisher` sein und die y-Achse `Combat`. Stelle das in einem Boxplot dar.“

```
ggplot(data = MarvelDC, aes(x = Publisher, y = Combat)) +
  geom_boxplot()
```



Um eine farbliche Trennung nach Geschlecht zu haben, kann das Beispiel erweitert werden: „Nehme MarvelDC als Daten. Die x-Achse soll Publisher sein und die y-Achse Combat. Stelle das in einem Boxplot dar. Wähle unterschiedliche Farben für Gender und fülle die Plots damit.“

```
ggplot(data = MarvelDC, aes(x = Publisher, y = Combat)) +  
  geom_boxplot(aes(fill = Gender))
```



7.5 Graphischen Funktionen

Mit den graphischen Funktionen (`geom_*`) gibt man an, welche Art der graphischen Darstellung man will. Beispiele sind:

aesthetics	Anzahl Variablen:	x ist dabei:	y ist dabei:
<code>geom_bar()</code>	1	diskret	-
<code>geom_histogram()</code>	1	kontinuierlich	-
<code>geom_qq()</code>	1	kontinuierlich	-
<code>geom_boxplot()</code>	2	diskret	kontinuierlich
<code>geom_violin()</code>	2	diskret	kontinuierlich
<code>geom_point()</code>	2	kontinuierlich	kontinuierlich
<code>geom_smooth()</code>	2	kontinuierlich	kontinuierlich

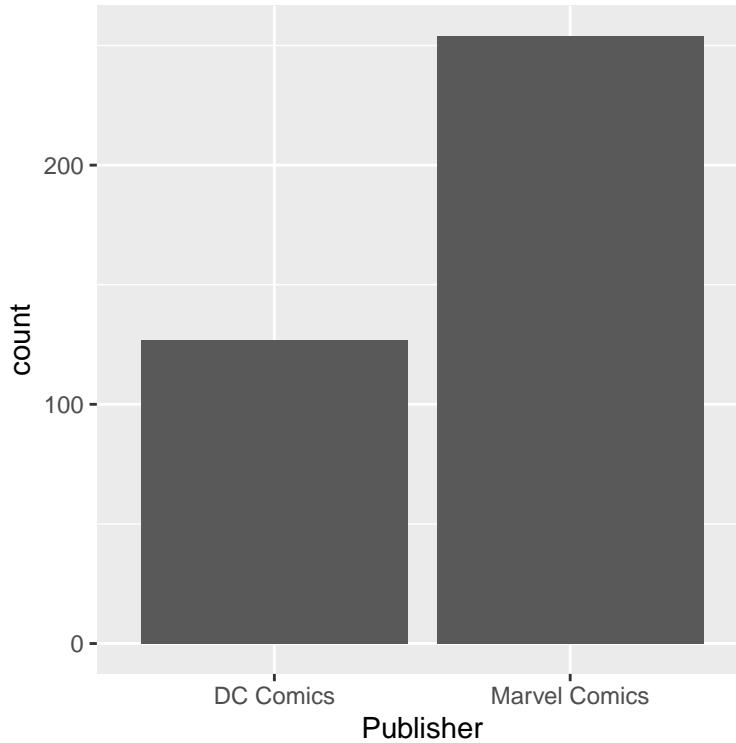
source: ggplot2 Cheat Sheet

7.5.1 Säulen- und Balkendiagramm

`geom_bar` ist eine graphische Funktion, um Häufigkeiten einer diskreten Variable in einem Säulen- oder Balkendiagramm zu visualisieren.

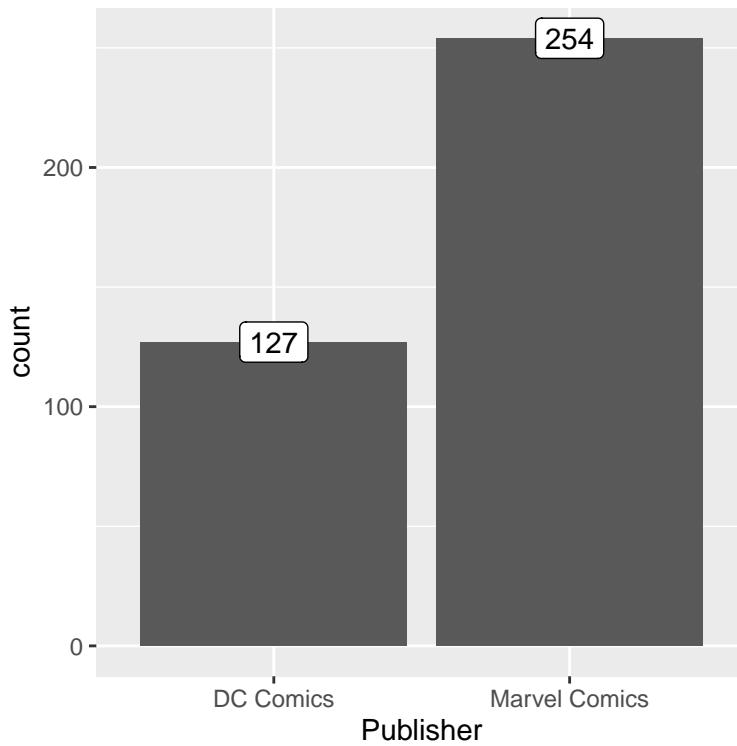
```
ggplot(data = MarvelDC,
       aes(x = Publisher)) +
  geom_bar()
```

*# Nehme Daten MarcelDC
auf der x-Achse soll Publisher
sein
stelle in einem Barplot dar*



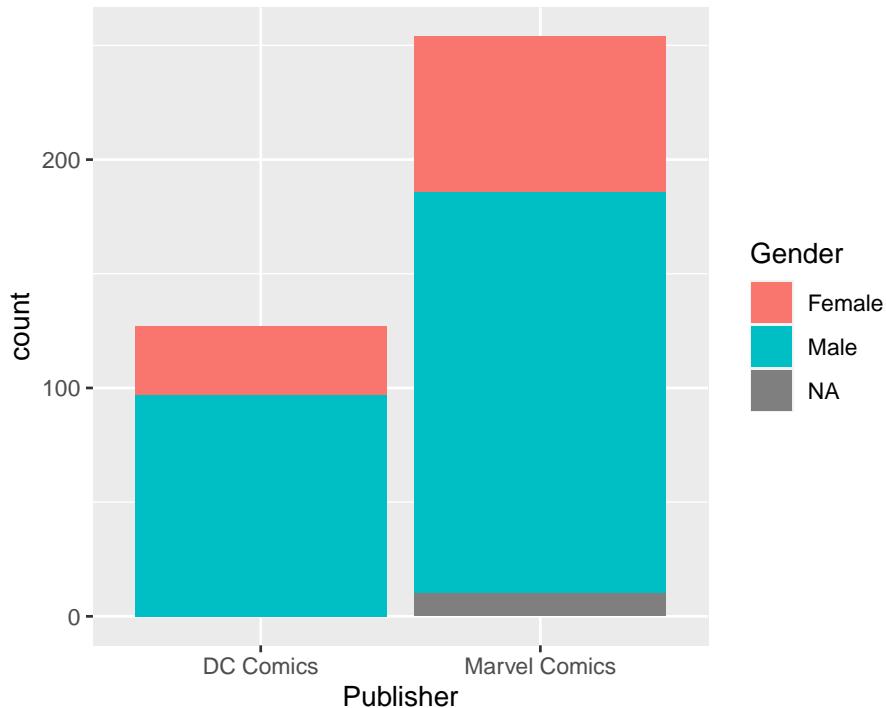
Marvel hat doppelt soviele Superhelden wie DC! Will man wissen, wie viele es genau sind, kann man mithilfe eines Graphik-Labels die Zählung hinzufügen:

```
ggplot(data = MarvelDC,
       aes(x = Publisher)) +
  geom_bar() +
  # man kann Label mithilfe der graphischen Funktion
  # geom_label und der Statistik "zählen" hinzufügen
  geom_label(stat = "count",
             aes(label = ..count..))
```



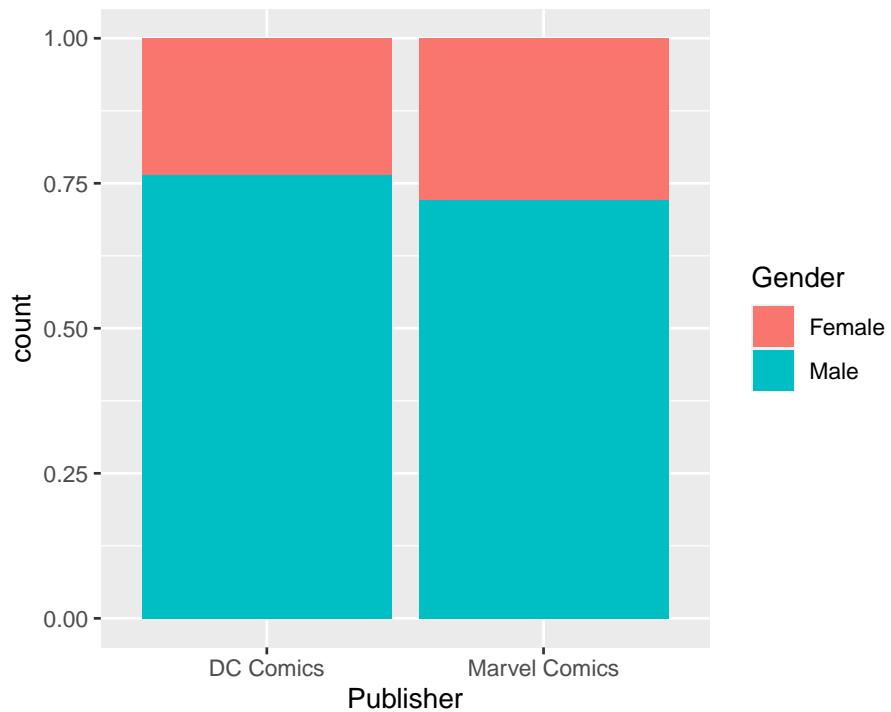
Indem man eine Farbkodierung vergibt, kann man in diesem Säulendiagramm das Geschlechterverhältnis beurteilen. Da hier mit Farben gefüllt wird, ist das Argument `fill`.

```
ggplot(data = MarvelDC,  
       aes(x = Publisher,  
            fill = Gender)) +      # vergabe Füllfarbe nach  
            # Geschlecht  
       geom_bar()
```



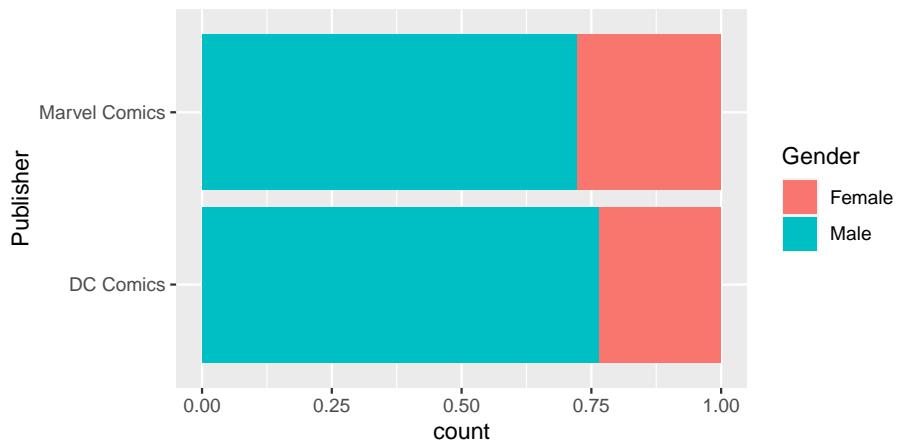
In der Graphik oben wurde der default-Parameter für Position verwendet, weil nichts anderes definiert wurde. Das ist `stack`. Bei `stack` werden die Teile übereinander „gestapelt“ dargestellt. Da weit mehr Charaktere aus Marvel stammen, kann die Position `fill` dafür sorgen, dass man das Geschlechterverhältnis besser beurteilen kann. `fill` erstellt Balken gleicher Länge, sodass eine Beurteilung der relativen Anteile leichter fällt. Erst sollten allerdings die NA Fälle herausgenommen werden, damit es bei Marvel und DC je nur zwei Kategorien gibt.

```
MarvelDC %>%
  filter(!is.na(Gender)) %>%      # Schliee Gender=NA aus
  ggplot(aes(x = Publisher,
             fill = Gender)) +
  geom_bar(position = "fill")      # mache alle Balken gleich lang
```



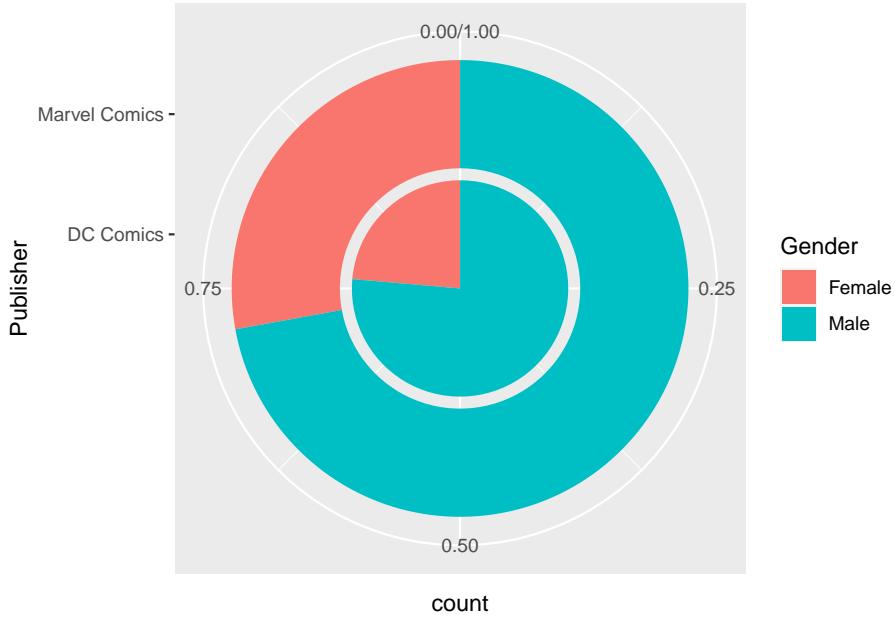
Die x oder y Angabe entscheidet darüber, ob ein Säulen- oder Balkendiagramm entsteht. Oben wurden die Daten als x an die Funktion gegeben und ein Säulendiagramm ist entstanden. Werden die Daten als y angegeben, so entsteht ein Balkendiagramm:

```
MarvelDC %>%
  filter(!is.na(Gender)) %>%
  ggplot(aes(y = Publisher,      # auf der y-Achse soll Publisher
             sein
             fill = Gender)) +
  geom_bar(position = "fill")
```



Dieselbe Information kann auch in Kreisform dargestellt werden, in sogenannten Polarkoordinaten:

```
MarvelDC %>%
  filter(!is.na(Gender)) %>%
  ggplot(aes(y = Publisher,
             fill = Gender)) +
  geom_bar(position = "fill") +
  coord_polar()           # stelle das in Polarkoordinaten
  dar
```



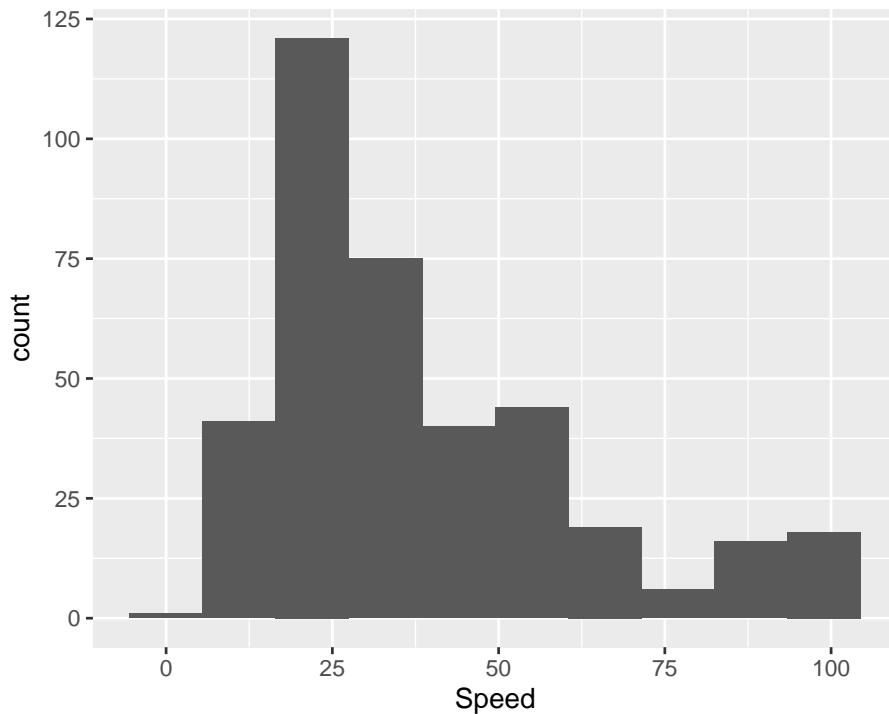
Die Information für Marvel DC wird auf dem äußeren Kreis abgebildet und die für DC Comics auf dem inneren Kreis abgebildet. Hier sieht man, dass bei DC Comics nur weniger als ein Viertel aller Helden weiblich sind, während es bei Marvel etwas mehr als ein Viertel sind.

7.5.2 Histogramm

`geom_histogram` ist eine graphische Funktion um Häufigkeiten einer kontinuierlichen Variable zu visualisieren. Beispielsweise kann man schauen, wie schnell die Superhelden unterwegs sind:

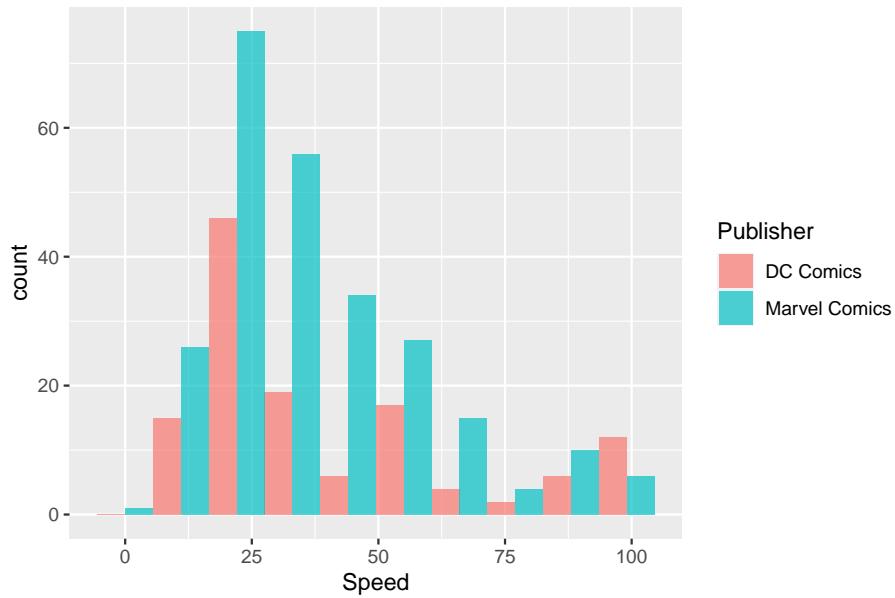
```
ggplot(MarvelDC,
       aes(x = Speed)) +          # auf der x-Achse soll Speed sein
```

```
geom_histogram(  
  dar  
  bins = 10)  
  # Anzahl der Kategorien soll 10  
  sein
```



Unterscheidet sich die Geschwindigkeit der Helden zwischen Marvel und DC?

```
ggplot(MarvelDC,  
  aes(x = Speed,  
      fill = Publisher)) +  
  geom_histogram(bins = 10,  
    alpha = 0.7,          # "Durchsichtigkeit" bei  
    70%  
    position = "dodge") # Position Balken:  
    nebeneinander
```



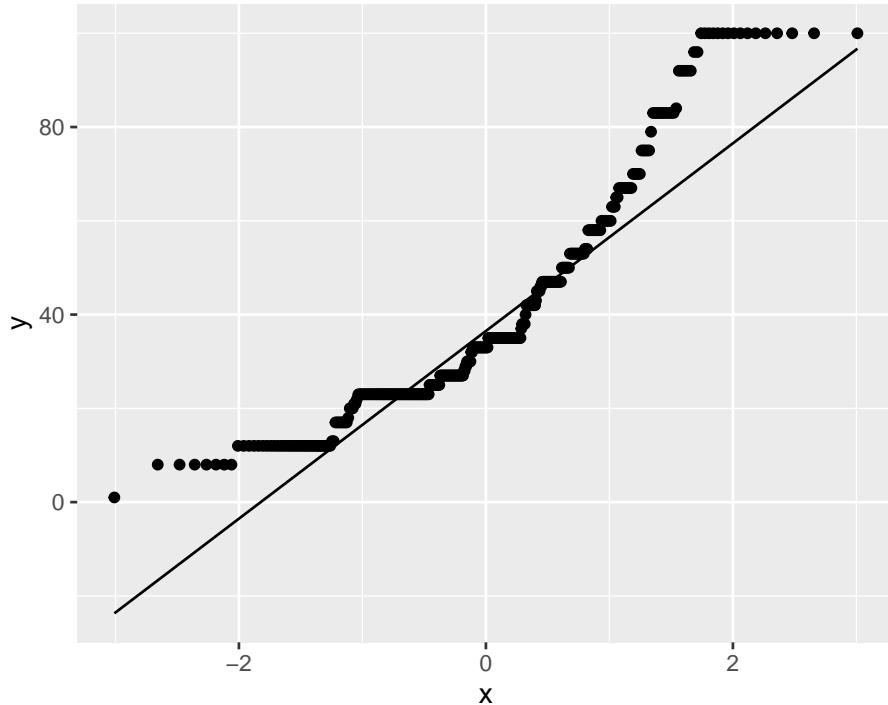
Offensichtlich nicht, die Verteilungen sind sich relativ ähnlich.

Bei `Speed` fällt allerdings direkt auf, dass es nicht sehr normalverteilt aussieht. Ein Histogramm lässt hier einen ersten Eindruck zu, aber eine visuelle Prüfung der Normalverteilung sollte lieber über einen qq-Plot vorgenommen werden.

7.5.3 QQ-Plot

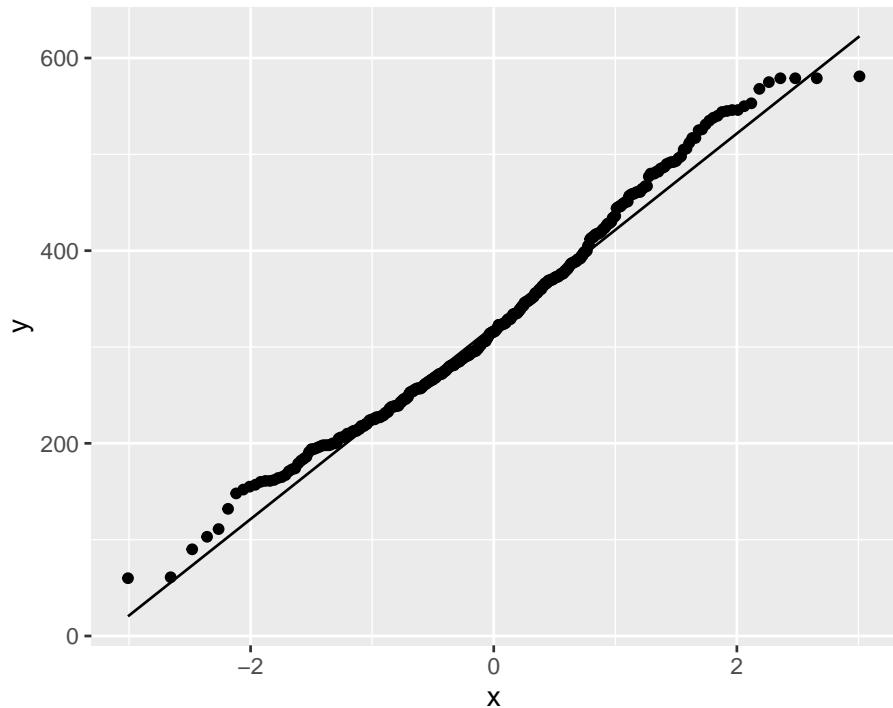
Bei einem qq-Plot kann und muss man Punkte und Vergleichslinie getrennt definieren:

```
ggplot(MarvelDC,
       aes(sample = Speed)) +
  geom_qq() + # erstelle Quantile von: Speed
  geom_qq_line() # erstelle QQ-Plot Punkte
                  # erstelle QQ-Plot Vergleichslinie
```



Bei Speed zeigen sich deutliche Abweichungen von einer Normalverteilung. Das gilt auch für alle anderen Eigenschaften (hier nicht gezeigt). Nur der Summenscore, Total, zeigt im qq-Plot eine Verteilung nahe der Normalverteilung:

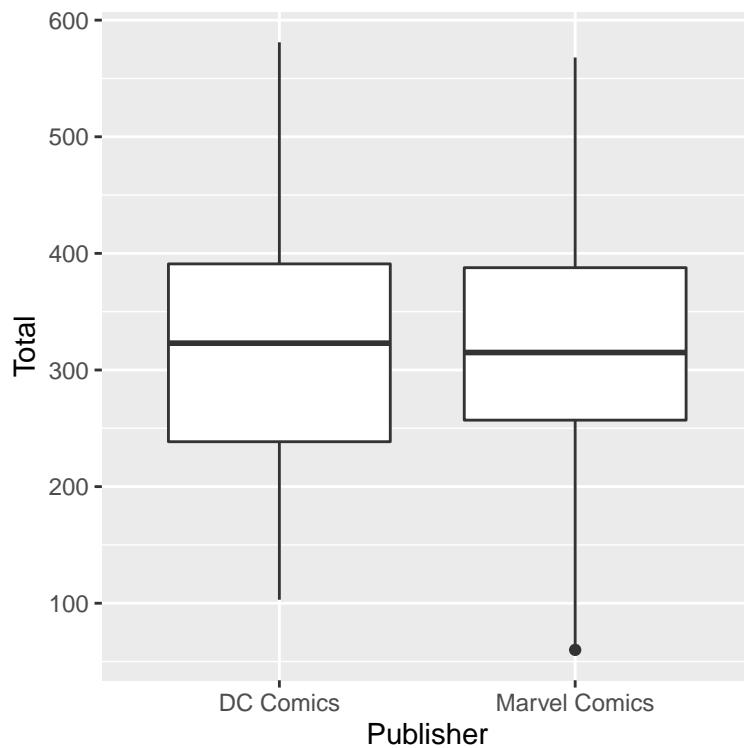
```
ggplot(MarvelDC,  
       aes(sample = Total)) +    # erstelle Quantile von: Total  
       geom_qq() +  
       geom_qq_line()
```



7.5.4 Boxplot

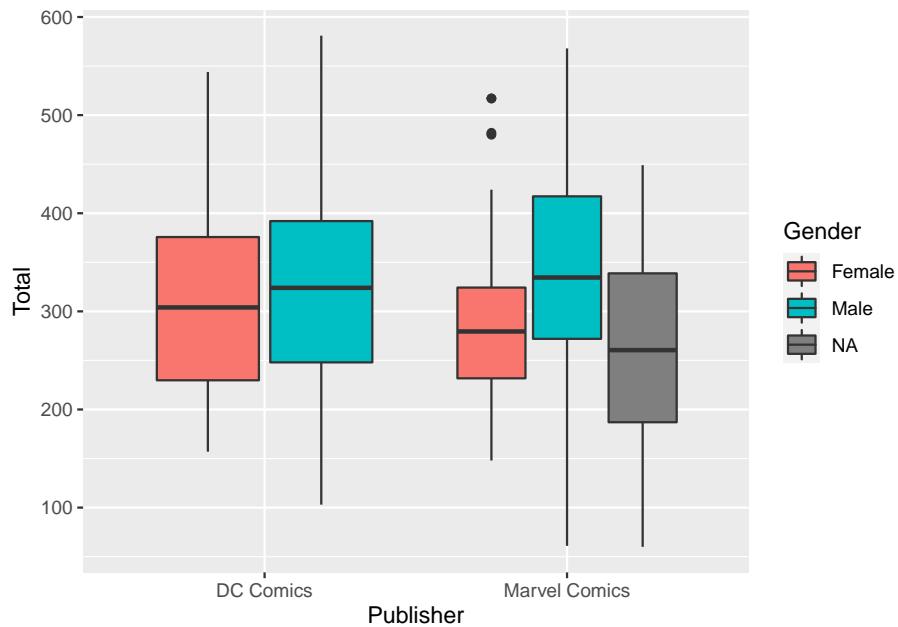
`geom_boxplot` ist eine graphische Funktion, um kontinuierliche Verteilungen getrennt nach einer diskreten Gruppierung darzustellen. Hat Marvel oder DC die stärkeren Superhelden? – Dafür betrachten wir den `Total` Wert jedes Helden.

```
ggplot(MarvelDC,  
       aes(x = Publisher, y = Total)) +  
  geom_boxplot()
```



Auf den ersten Blick mithilfe eines Boxplots gibt es keine Unterschiede zwischen Marvel Comics und DC Comics.

```
ggplot(MarvelDC,
       aes(x = Publisher, y = Total,
            fill= Gender)) +
  geom_boxplot()
```

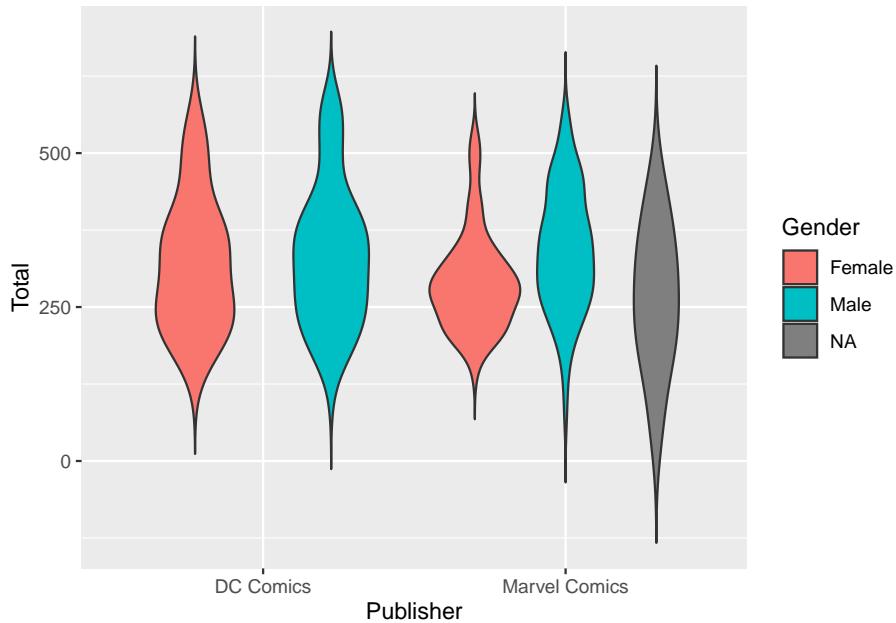


Betrachtet man **Total** aufgespalten nach Geschlecht, so scheint Marvel etwas stärkere Männer und schwächere Frauen zu haben – allerdings sind die Unterschiede so gering, dass es sich wahrscheinlich um zufällige Schwankungen und nicht um belastbare Effekte handelt! Keins der Universen scheint also dem anderen überlegen zu sein.

7.5.5 Violinen-Plot

`geom_violin` ist ebenfalls eine graphische Funktion um kontinuierliche Verteilungen getrennt nach einer diskreten Gruppierung darzustellen – und lässt etwas mehr Aufschluss auf die Daten zu (z.B. wie die grobe Verteilung der Werte ist).

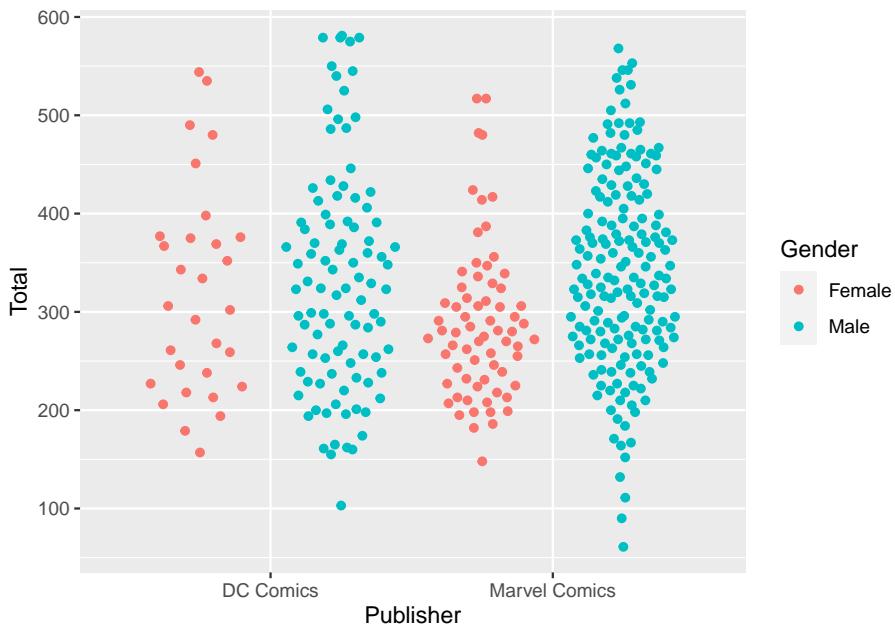
```
ggplot(MarvelDC,
       aes(x = Publisher, y = Total,
           fill= Gender)) +
  geom_violin(                  # stelle in einem Violinenplot dar
              trim=FALSE)      # schneide Spitzen nicht ab
```



7.5.6 Beeswarm Plot mit dem Paket `ggbeeswarm`

Das package `ggbeeswarm` bietet eine ähnliche Darstellung durch die graphische Funktion `geom_quasirandom`:

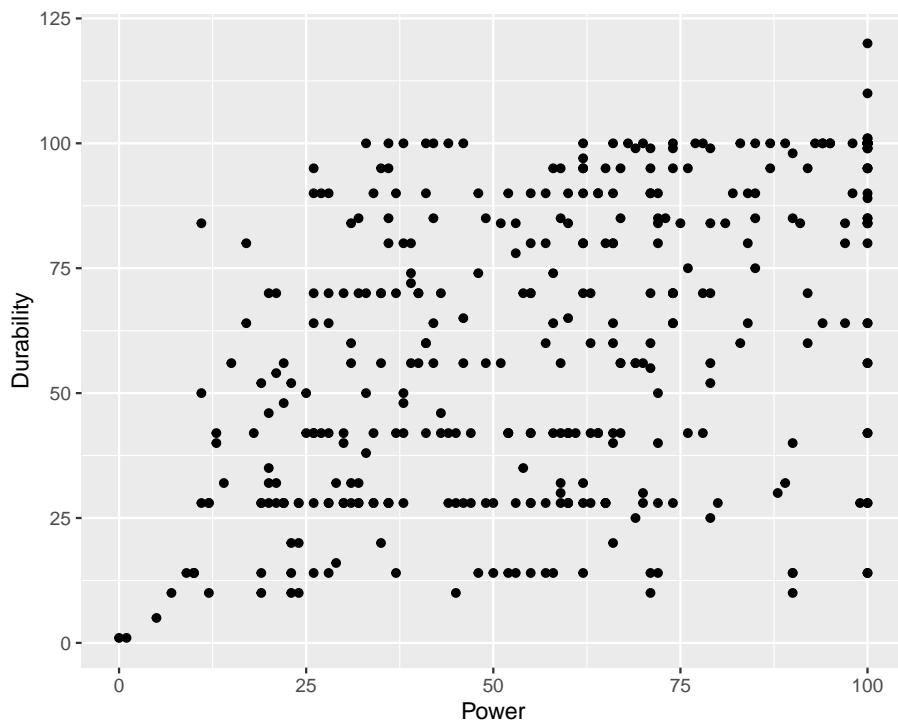
```
library(ggbeeswarm)
MarvelDC %>%
  filter(!is.na(Gender)) %>%
  ggplot(aes(y = Total, x = Publisher,
             color = Gender)) +
  geom_quasirandom(dodge.width = 1)    # stelle als beeswarm dar
```



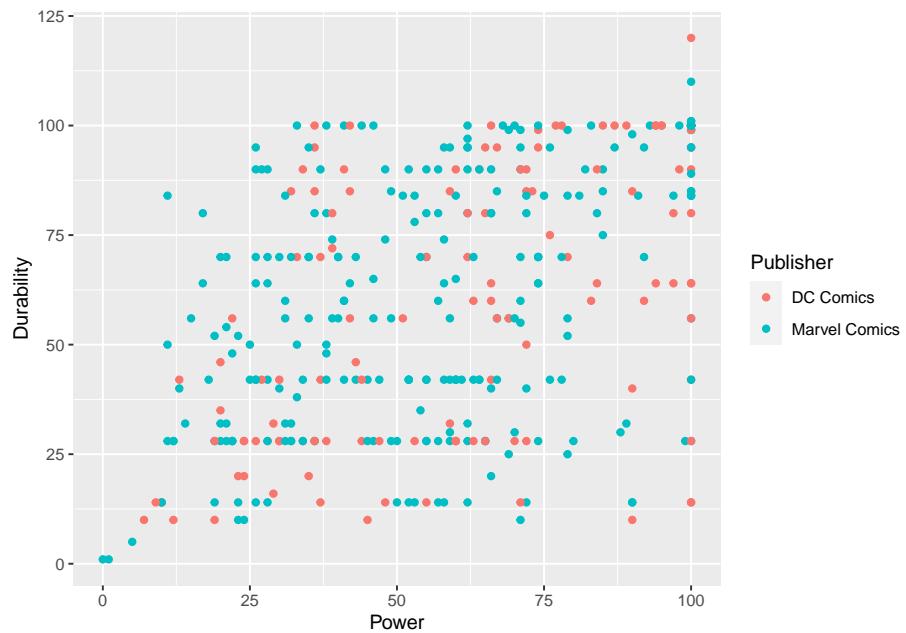
Weder Violinen-Plot noch Beeswarm-Plot deuten darauf hin, dass Marvels Superhelden stärker und cooler sind, als die von DC. Ohne gegenteilige Hinweise bleibt es also bei der Annahme, dass beide Universen gleich gut sind.

7.5.7 Scatterplot

Ein Scatterplot mit `geom_point` stellt die Daten als Punkte im Raum zweier kontinuierlicher Variablen dar. Beispielsweise kann man `Power` und `Durability` aller Helden gegeneinander darstellen:



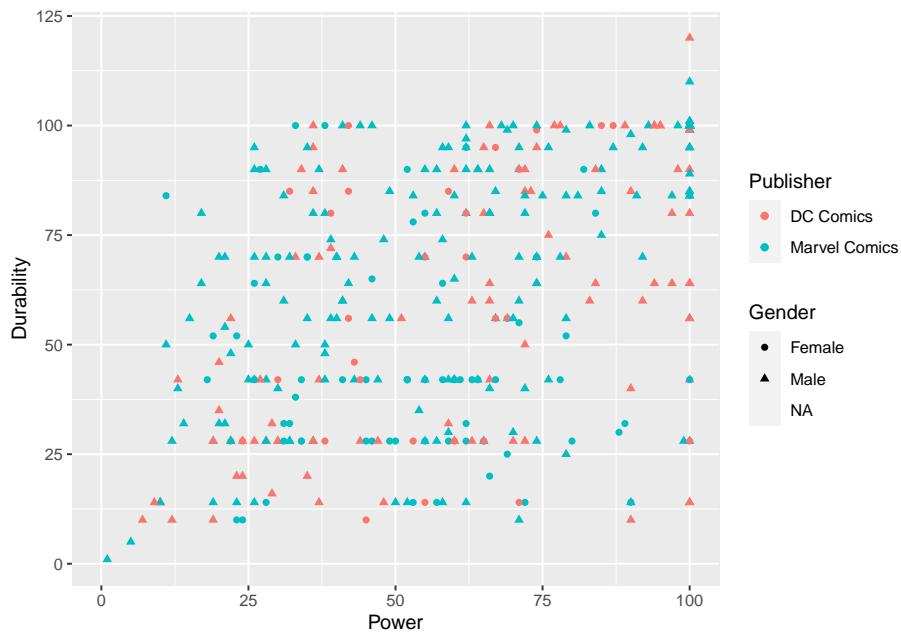
Auch diese Graphik kann einfach nach Publisherfarblich kodiert werden. Hier sollen die Punkte Farbe haben, aber es wird nichts mit Farbe gefüllt, deswegen ist das Argument hier `color`:



Die Daten beider Publisher sind weit verteilt, es ist also keineswegs so, dass DCs Superhelden weniger Power haben, als die von Marvel.

Will man zusätzlich nach Geschlecht unterscheiden, so geht das beispielsweise über die Form der Punkte mithilfe des Arguments `shape`:

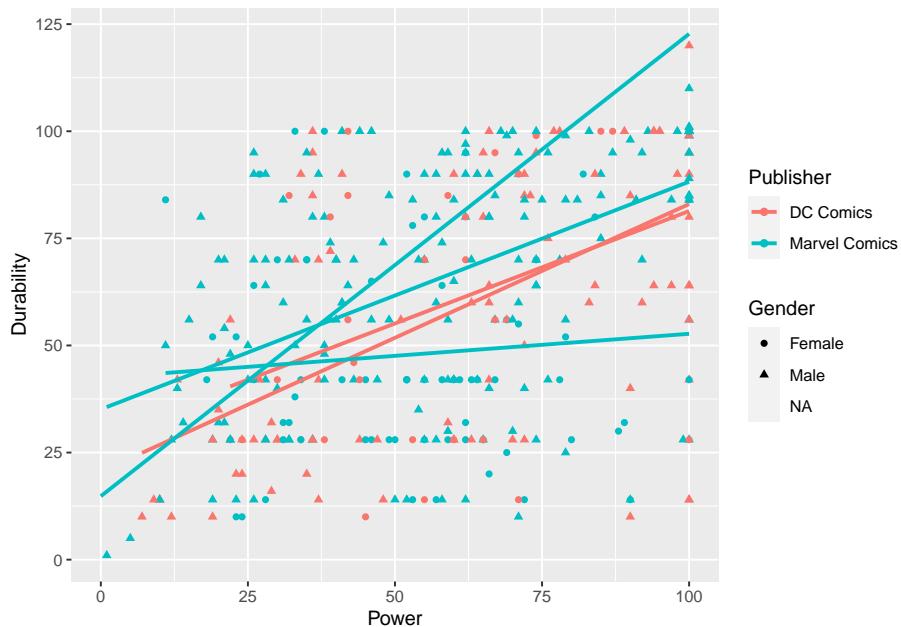
```
ggplot(data = MarvelDC,
       aes(x = Power, y = Durability,
            color = Publisher,
            shape = Gender)) +      # getrennte Formen für Gender
       geom_point()
```



7.5.8 (Linearer) Fit

Die **Power** und **Durability** Daten können mit einer Funktion gefittet werden. So kann man visuell prüfen, ob es bei Marvel und DC jeweils einen anderen Zusammenhang zwischen Power und Durability gibt. Dafür wird die graphische Funktion `geom_smooth` verwendet. Um ein **lineares** Modell zu erhalten, muss das Argument `method = "lm"` für „linear model“ gesetzt werden. Um `y` durch `x` vorherzusagen, braucht es die Formel `y ~ x`:

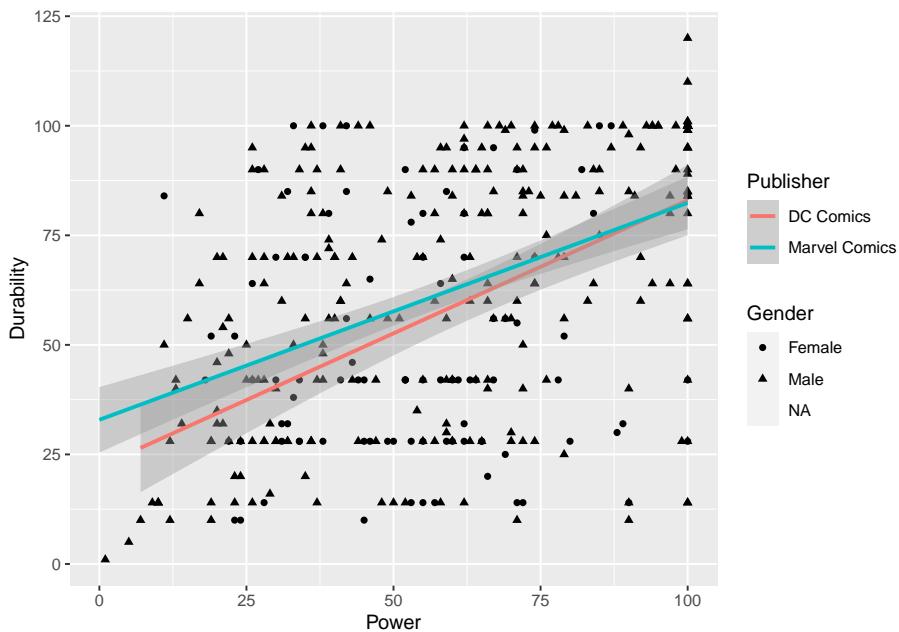
```
ggplot(data = MarvelDC,
       aes(x = Power, y = Durability,
           color = Publisher,
           shape = Gender)) +
  geom_point() +
  geom_smooth(method = "lm",          # lineares Modell
              formula = 'y ~ x',    # y durch x vorhersagen
              se = FALSE)           # keine Fehlerbalken
```



Das sind aber 5 Fits – je für Geschlecht und Publisher. Nicht nur für die beiden Publisher wie geplant.

Das liegt daran, dass sowohl Farbe als auch Form innerhalb von `ggplot` bei den allgemeinen `aesthetics` definiert sind. Um nur einen linearen Fit für jeden Publisher zu erhalten, nicht aber nach Geschlecht aufgespalten, muss man die Form-Kennung für Geschlecht nur an `geom_point` geben. Auf diese Weise kann sie nicht mehr an `geom_smooth` weitergegeben werden. Gibt man die Farb-Kennung für Publisher nur an die Linien, also nur an `geom_smooth`, so erhält man nur eine farbliche Trennung bei den Fit-Linien, nicht aber bei den zugrunde liegenden Punkten. Mit `se=TRUE` werden die Fehlerintervalle um die linearen Fit-Funktionen herum dargestellt. Der folgende Code liefert einen linearen Fit der Power und Durability der Superhelden farblich getrennt nach Marvel und DC:

```
ggplot(data = MarvelDC,
       aes(x = Power, y = Durability)) +
  geom_point(aes(shape = Gender)) +      # Form gilt nur für Punkte
  geom_smooth(aes(color = Publisher),    # Farbe gilt nur für Linien
              method = "lm",
              formula = 'y ~ x',
              se = TRUE)
```



Auch hier zeigen sich weiterhin keine Unterschiede zwischen den Publishern. Es muss wohl endgültig davon ausgegangen werden, dass Marvel nicht das alleinige Recht auf coole Superhelden hat. Unsere Daten geben keine Hinweise auf die Überlegenheit des einen oder anderen Superhelden-Universums.

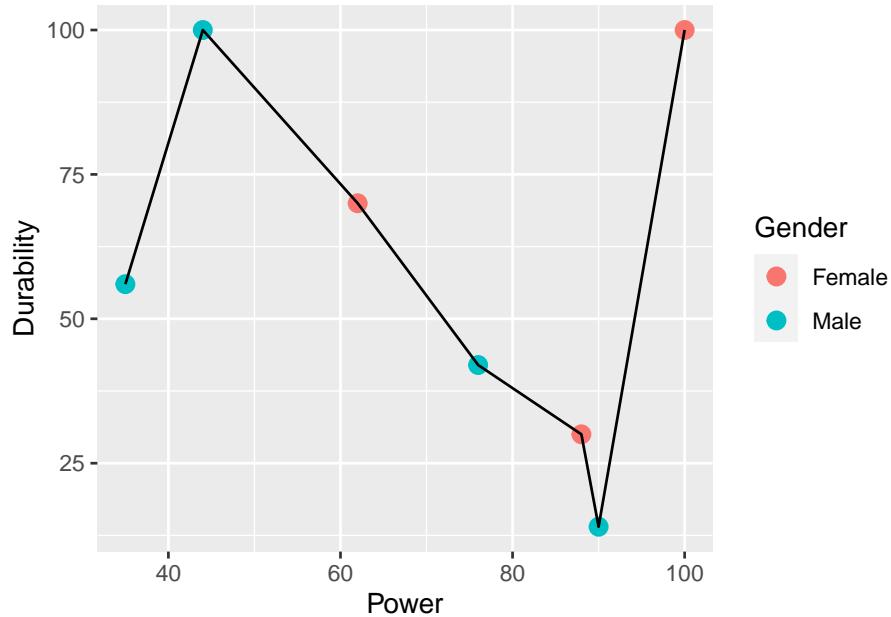
7.5.9 Linienplot

Linienplots lassen sich am besten an einer überschaubaren Anzahl von Punkten veranschaulichen, z.B. eignet sich eine Auswahl der X-Men. Die Datenpunkte werden farblich nach Geschlecht getrennt – dann werden alle Punkte durch eine Linie verbunden:

```
# Erstelle Vektor der X-Men Charaktere:
XMen <- c("Wolverine", "Jean Grey", "Storm", "Beast",
         "Cyclops", "Professor X", "Raven")

MarvelDC %>%
  filter(Name %in% XMen) %>% # Verwende nur X-Men
  ggplot(aes(x = Power,
             y = Durability)) +
  geom_point( # stelle in Scatterplot dar
    aes(color = Gender), # Farbe nach Gender (Punkte)
```

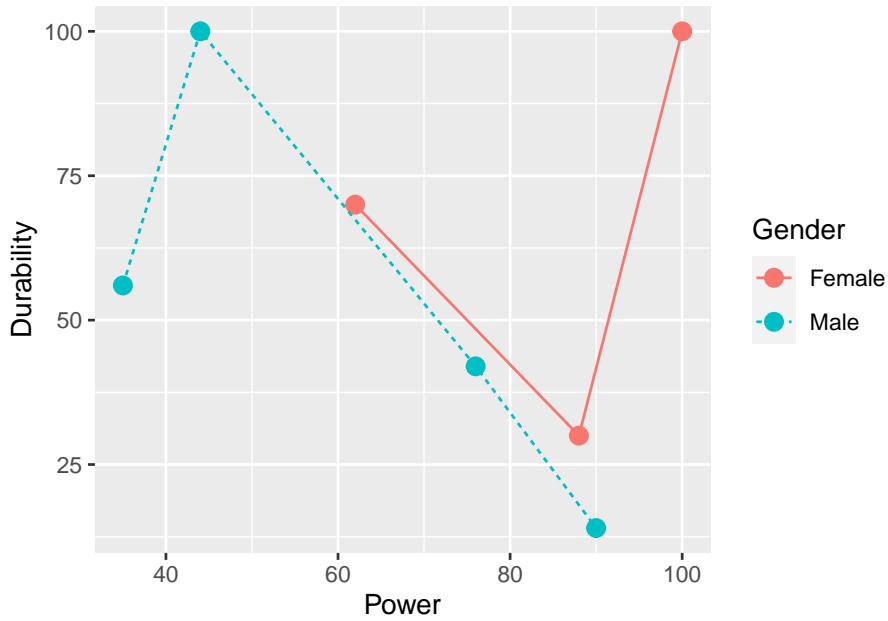
```
size = 3) +      # Punkte sollen Größe 3 haben
geom_line()       # verbinde Punkte mit Linie
```



Da Farbe innerhalb von `geom_point` definiert ist, enthalten die allgemeinen `aesthetics` keine Gruppierung. Deswegen verbindet die Linie einfach alle Datenpunkte. Informativer ist es natürlich, getrennte Linien für die Geschlechter zu haben. Dazu muss die Farbdefinition in den allgemeinen `aesthetics` erfolgen und nicht innerhalb von `geom_point`:

```
XMen <- c("Wolverine", "Jean Grey", "Storm", "Beast",
        "Cyclops", "Professor X", "Raven")

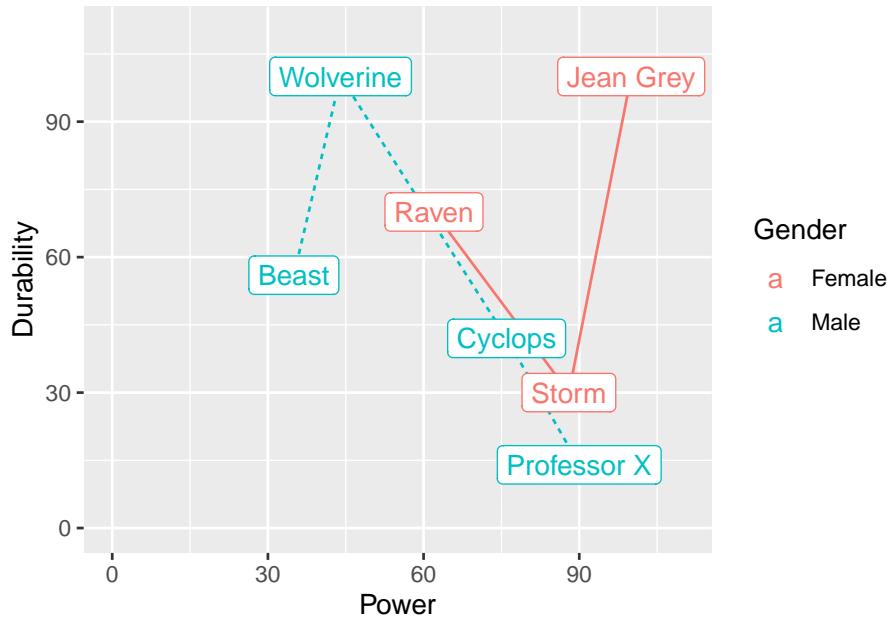
MarvelDC %>%
  filter(Name %in% XMen) %>%
  ggplot(aes(x = Power,
             y = Durability,
             color = Gender)) +
    # Farbe nach Gender (Punkte &
    # Linie)
  geom_point(size = 3) +
  geom_line(aes(linetype = Gender)) # zusätzlich Linentyp nach
                                   # Gender
```



Will man wissen, welcher Charakter welche Werte aufweist, fügt `geom_label` Textlabel zu dem Plot hinzu. Außerdem können die Achsen so skaliert werden, dass der absolute Nullpunkt enthalten ist und beide Achsen symmetrisch laufen:

```
XMen <- c("Wolverine", "Jean Grey", "Storm", "Beast",
         "Cyclops", "Professor X", "Raven")

MarvelDC %>%
  filter(Name %in% XMen) %>%
  ggplot(aes(x = Power,
             y = Durability,
             color = Gender)) +
  geom_point(size = 3) +
  geom_line(aes(linetype = Gender)) +
  geom_label(aes(label = Name,           # Füge Label hinzu mit:
              "Name"
              color = Gender)) + # färbe Labels nach
                        # Geschlecht
  xlim(c(0, 110)) +          # x-Achse soll laufen von
                            # 0-110
  ylim(c(0, 110))           # y-Achse soll laufen von
                            # 0-110
```



7.6 Sonstige Funktionalität

7.6.1 Labels

Label sind die wahrscheinlich wichtigste zusätzliche Funktionalität, die fast immer benötigt wird, um einen Plot allgemein verständlich zu machen. Mithilfe von `labs()` können Titel, Untertitel, Achsenbeschriftungen und eine Caption zu einem Plot hinzugefügt werden. Z.B. für den Plot der X-Men:

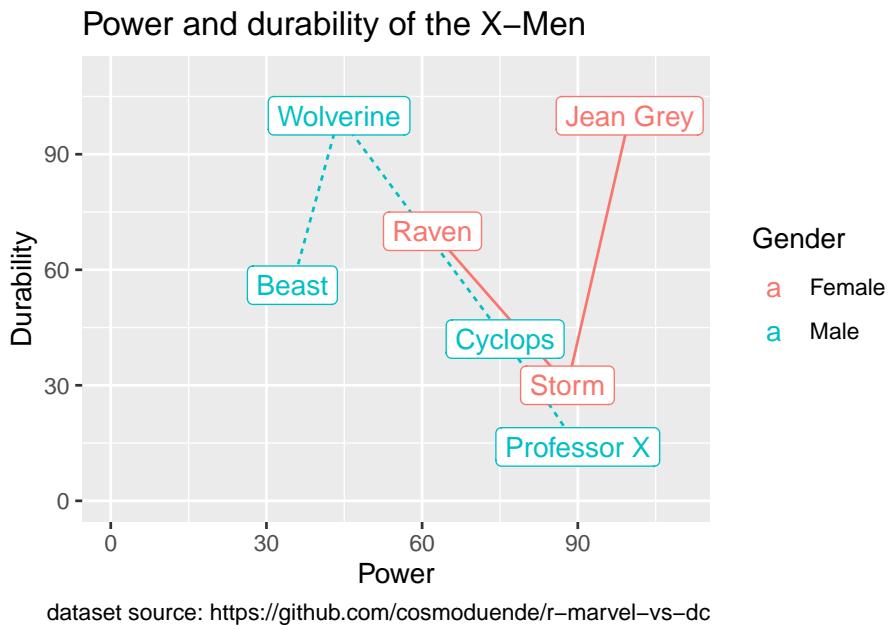
```
XMen <- c("Wolverine", "Jean Grey", "Storm", "Beast",
        "Cyclops", "Professor X", "Raven")

MarvelDC %>%
  filter(Name %in% XMen) %>%
  ggplot(aes(x = Power,
             y = Durability,
             color = Gender)) +
  geom_point(size = 3) +
  geom_line(aes(linetype = Gender)) +
  geom_label(aes(label = Name,
                 color = Gender)) +
```

```

xlim(c(0, 110)) +
ylim(c(0, 110)) +
# Füge Beschriftungen hinzu: Titel, x- & y-Achsen, Caption
labs(title = "Power and durability of the X-Men",
x = "Power",
y = "Durability",
caption = "dataset source:
https://github.com/cosmoduende/r-marvel-vs-dc" )

```

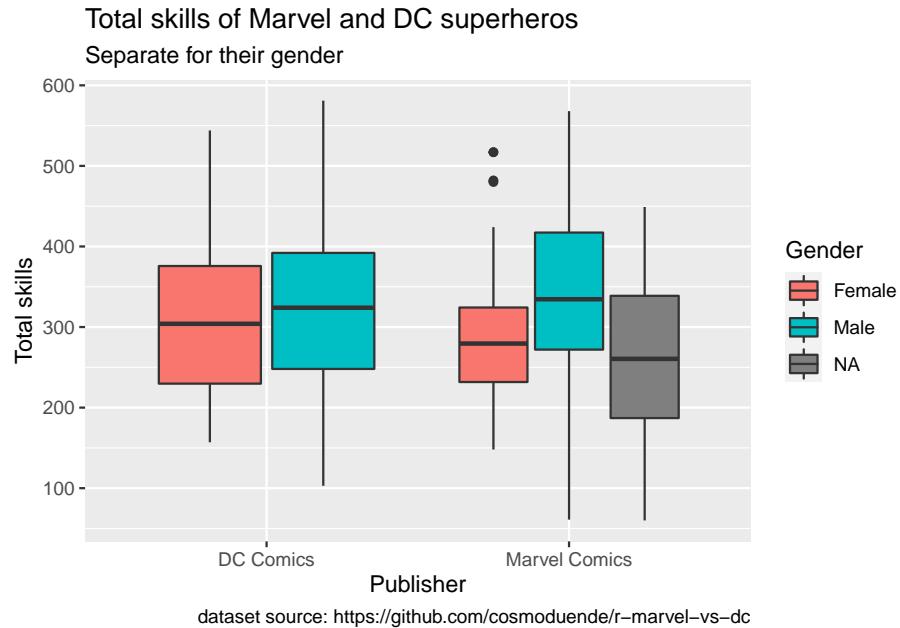


Ein weiteres Beispiel:

```

ggplot(MarvelDC,
aes(x = Publisher, y = Total,
fill= Gender)) +
geom_boxplot() +
labs(title = "Total skills of Marvel and DC superheros",
subtitle = "Separate for their gender",
x = "Publisher",
y = "Total skills",
caption = "dataset source:
https://github.com/cosmoduende/r-marvel-vs-dc" )

```



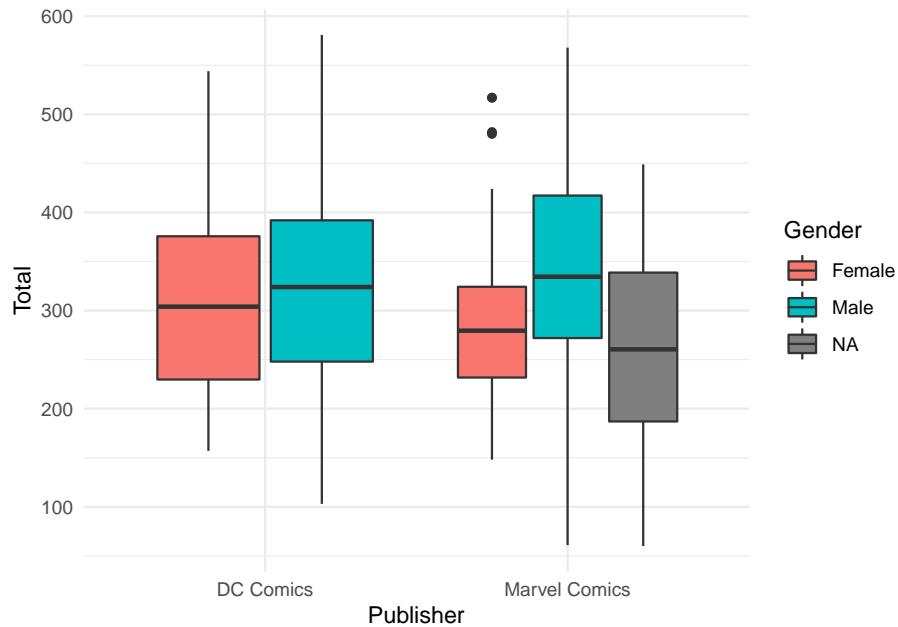
7.6.2 Thema

Es gibt verschiedene **themes** die das grundlegende Graphiklayout bestimmen.

- `theme_bw()` – weiSSer Hintergrund mit grid Linien
- `theme_gray()` – grauer Hintergrund mit hellen grid Linien (default)
- `theme_dark()` – dunkler Hintergrund mit grid Linien
- `theme_minimal()` – minimalistisches theme
- `theme_void()` – leeres theme

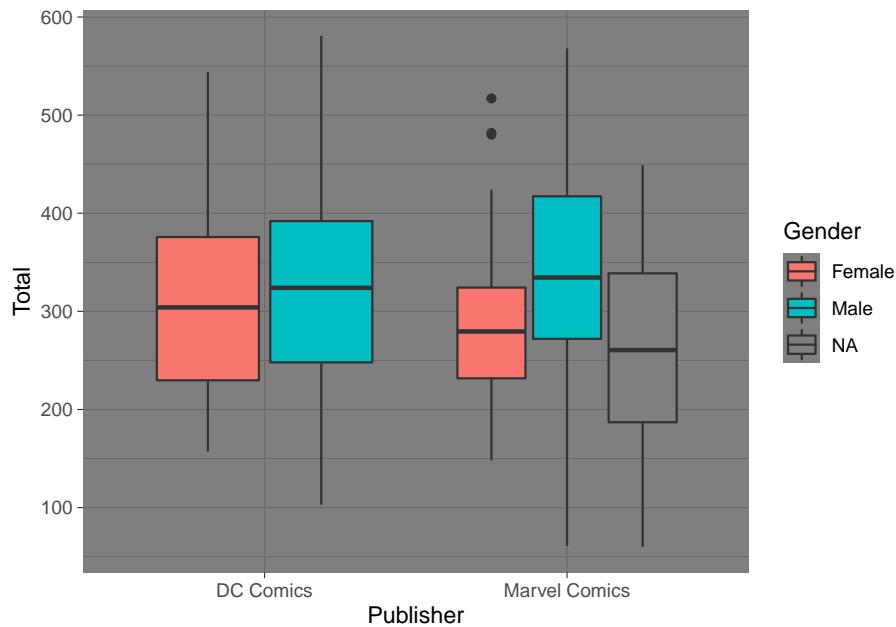
Minimalistisches Thema:

```
ggplot(MarvelDC,
       aes(x = Publisher, y = Total,
            fill= Gender)) +
  geom_boxplot() +
  theme_minimal()      # verwende Thema: Minimal
```



Dunkles Thema:

```
ggplot(MarvelDC, aes(x = Publisher, y = Total, fill= Gender)) +  
  geom_boxplot() +  
  theme_dark()          # verwende Thema: Dunkel
```



7.6.3 Position

Mit `position` kann die Positionierung graphischer Elemente zueinander einge stellt werden. Positionsarten sind z.B.:

- `identity` – Werte sollen an ihrer wahren Position (d.h. so wie sie in der Funktion eingegeben wurden) angezeigt werden. Tipp: Bei dieser Einstel lung kann man die angezeigten Werte machmal nicht so gut erkennen durch (z.B. weil sie aufeinander liegen). Manchmal kann es hilfreich sein, den `alpha` Parameter zusätzliche zu verwenden, um die angezeigten Punk ten (oder Balken, etc.) etwas zu entzerren.
- `stack` – Elemente aufeinander schichten (default)
- `dodge` – Elemente nebeneinander zeigen
- `jitter` – Zufälliges Rauschen auf die (x,y) Position jedes Elements addie ren, um den Plot zu entzerren

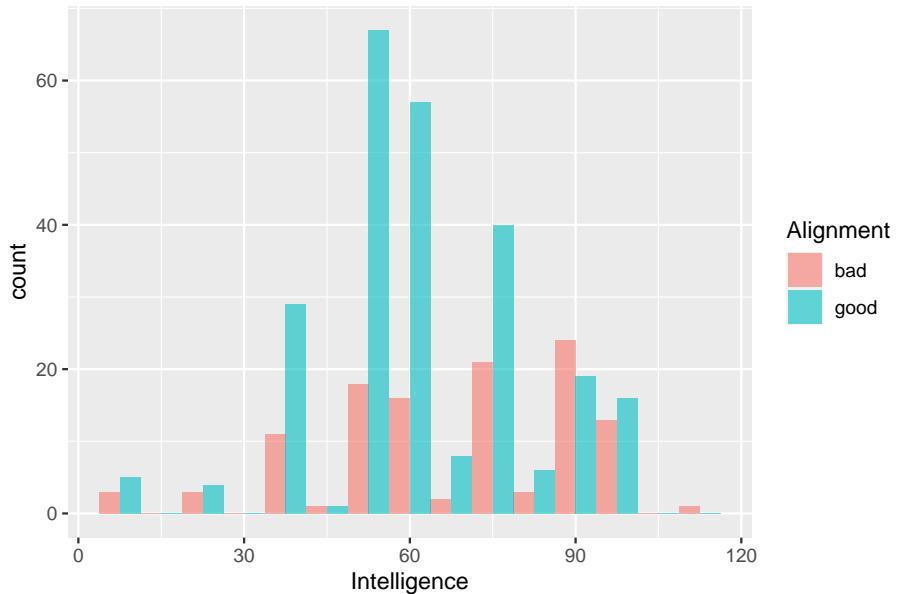
`dodge` = nebeneinander:

```
MarvelDC %>%
  filter(!is.na(Alignment), # verwende nur
         Alignment...)
```

```

    Alignment != "neutral") %>% # ... `good` und `bad`
ggplot(aes(x = Intelligence,
           fill = Alignment)) +
  geom_histogram(bins = 15,
                 alpha = 0.6,
                 position = "dodge") # Position: nebeneinander

```

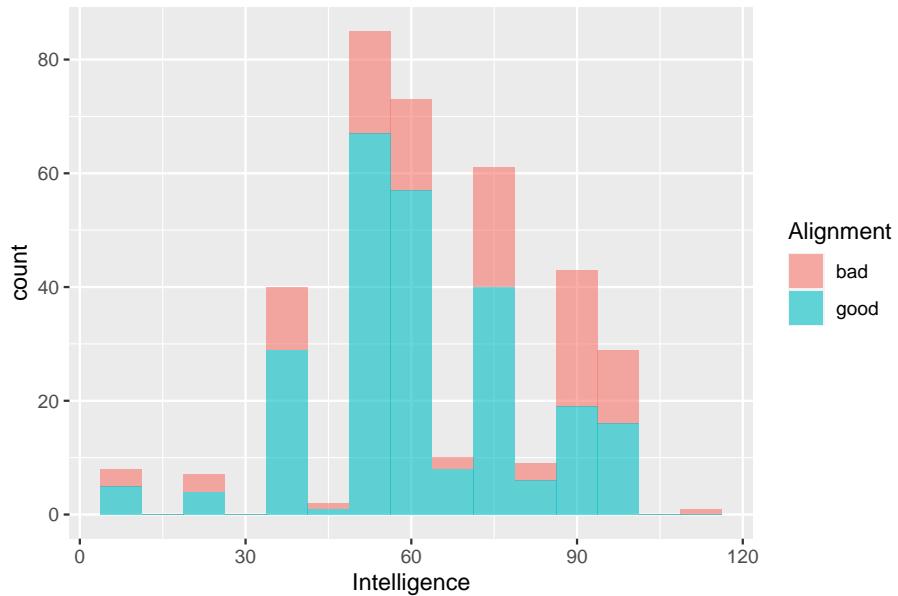


stack = aufeinander:

```

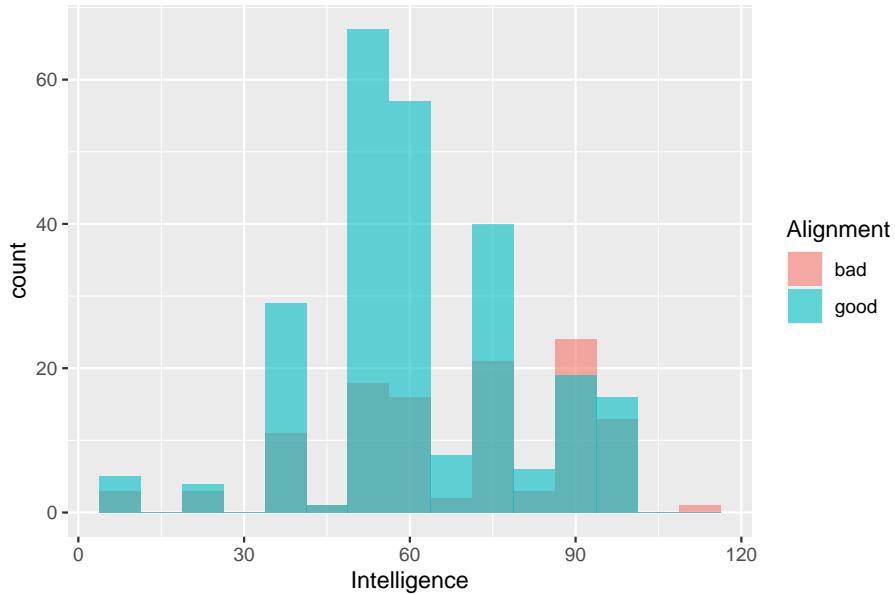
MarvelDC %>%
  filter(!is.na(Alignment),
         Alignment != "neutral") %>%
  ggplot(aes(x = Intelligence,
             fill = Alignment)) +
  geom_histogram(bins = 15,
                 alpha = 0.6,
                 position = "stack") # Position: aufeinander

```



identity = hintereinander:

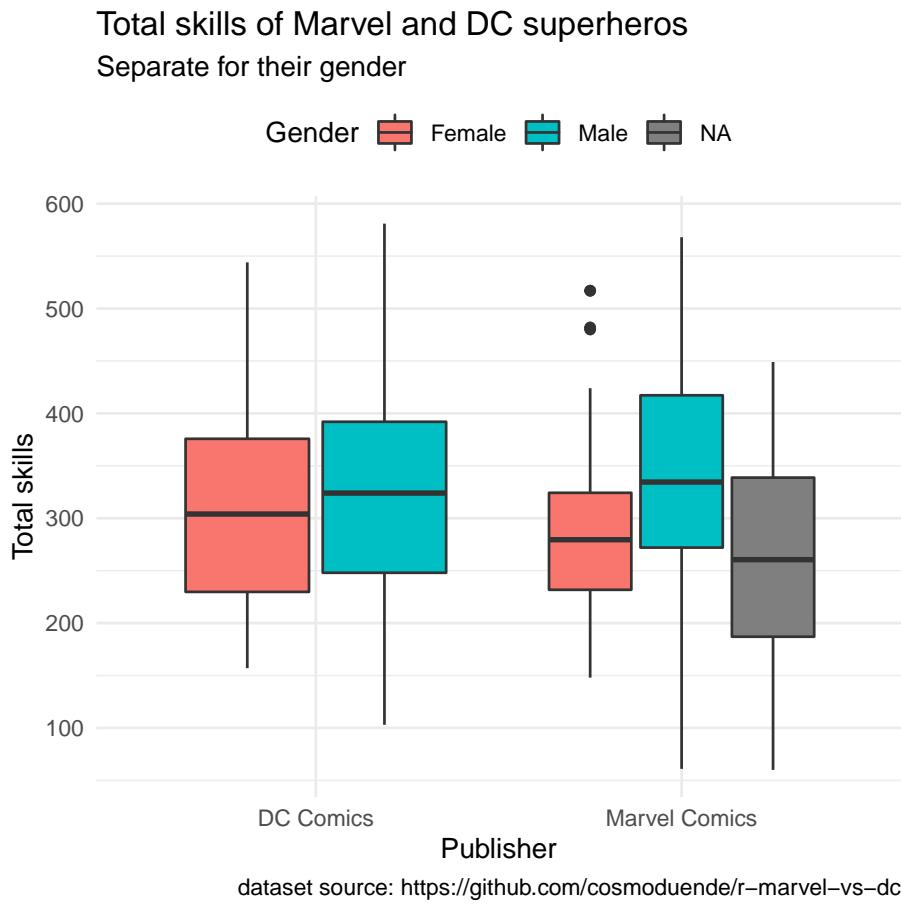
```
MarvelDC %>%
  filter(!is.na(Alignment),
         Alignment != "neutral") %>%
  ggplot(aes(x = Intelligence,
             fill = Alignment)) +
  geom_histogram(bins = 15,
                 alpha = 0.6,
                 position = "identity") # Position:
                           hintereinander
```



7.6.4 Legenden

Die Legende wird von ggplot automatisch erstellt und kann einfach an die Positionen `bottom`, `right`, `left` und `top` gelegt werden:

```
ggplot(MarvelDC, aes(x = Publisher, y = Total,
                      fill= Gender)) +
  geom_boxplot() +
  theme_minimal() +
  labs(title = "Total skills of Marvel and DC superheros",
       subtitle = "Separate for their gender",
       x = "Publisher",
       y = "Total skills",
       caption = "dataset source:
                  https://github.com/cosmoduende/r-marvel-vs-dc") +
  theme(legend.position="top") # positioniere Legende oben
```



7.6.5 Koordinatensysteme

Das wichtigste Koordinatensystem ist zweifellos das kartesische. In R kann man aber z.B. auch Polarkoordinaten verwenden, oder das kartesische System „drehen“:

- `coord_flip` – dreht ein kartesisches Koordinatensystem um (x wird zu y und umgekehrt) - empfohlen ist allerdings, das über die Definition von x und y direkt zu lösen.
- `coord_polar` – Polarkoordinaten

Polarkoordinaten wurden bereits beim Balkendiagramm gezeigt.

7.6.6 Farbskalen

`scale` bestimmt, auf welche Farbskala die Daten abgebildet werden. Am Beispiel von `viridis`:

- `scale_color_viridis_d()` – diskretes `viridis` Spektrum für das Argument `color`
- `scale_color_viridis_c()` – kontinuierliches `viridis` Spektrum für das Argument `color`
- `scale_fill_viridis_d()` – diskretes `viridis` Spektrum für das Argument `fill`

Über `option` kann im Paket `viridis` auf eine andere der enthaltenen Skalen zugegriffen werden:

- `scale_fill_viridis_d(option=inferno)` – diskretes `inferno` Spektrum für das Argument `fill`

Neben dem `viridis` Package gibt es auch den ColorBrewer in R, der unter anderem folgende Farbkarten kennt (am Beispiel von `fill` Skalen):

- `scale_fill_brewer(palette="Set1")` andere mögliche Einstellungen für `palette`: "Set2", "Set3", "Accent", "Dark2", "Pastel1", "Pastel2"

Wenn man allerdings schon ein Farbspektrum selbst definiert, ist es empfehlenswert ein *colorblind friendly* Schema wie alle Skalen aus dem `viridis` Package zu benutzen.

Im folgenden Beispiel werden verschiedene Einstellungen zusammengeführt, um eine ansprechende und informative Graphik zu erhalten. Erst wird eine Liste der besten Superhelden der X-Men, der Avengers und der Guardians of the Galaxy erstellt. Dann veranschaulicht ein Balkendiagramm welcher Art sie angehören. Also ob es Menschen sind oder Mutanten oder Androiden... Dann wird die Graphik informativ beschriftet, so dass sie auch ohne den zugehörigen Code verständlich ist, und als letztes wird ein diskretes `viridis` Spektrum als die Farbskala bestimmt, die R verwenden soll:

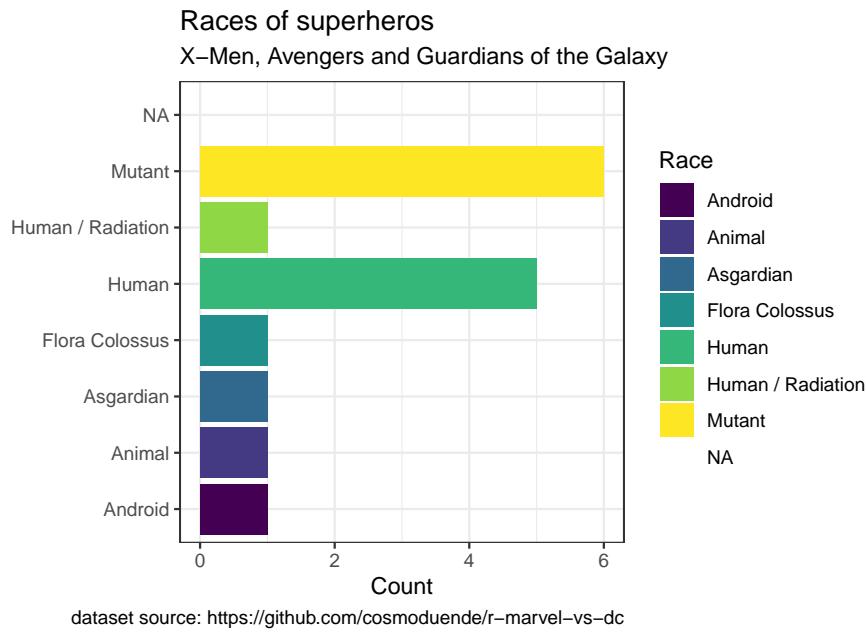
```
# Charaktere der X-Men, der Guardians of the Galaxy und der
# Avengers
XMen <- c("Wolverine", "Jean Grey", "Storm", "Beast",
          "Cyclops", "Professor X", "Raven")
GoG <- c("Groot", "Quill", "Rocket Raccoon")
```

```
Avengers <- c("Iron Man", "Captain America", "Black Widow",
            "Hulk", "Hawkeye", "Thor", "Spiderman", "Vision")

# ein Vektor mit den coolsten Superhelden
BestHeros <- c(XMen, GoG, Avengers)

MarvelDC %>%
  filter(Name %in% BestHeros) %>%      # Nur Charaktere aus
  'BestHeros'
  ggplot(aes(y = Race,                      # y soll 'Race' sein
             fill= Race)) +           # vergabe Füllfarbe nach
  'Race'
  geom_bar() +                         # stelle als Balken dar
  theme_bw() +                         # in scharz-weiem Thema

# Beschrifte
  labs(title = "Races of superheroes",
       subtitle = "X-Men, Avengers and Guardians of the Galaxy",
       x = "Count",
       y = "",
       caption = "dataset source:
https://github.com/cosmoduende/r-marvel-vs-dc" ) +
  scale_fill_viridis_d()                 # verwende Farben aus
  'viridis'
```



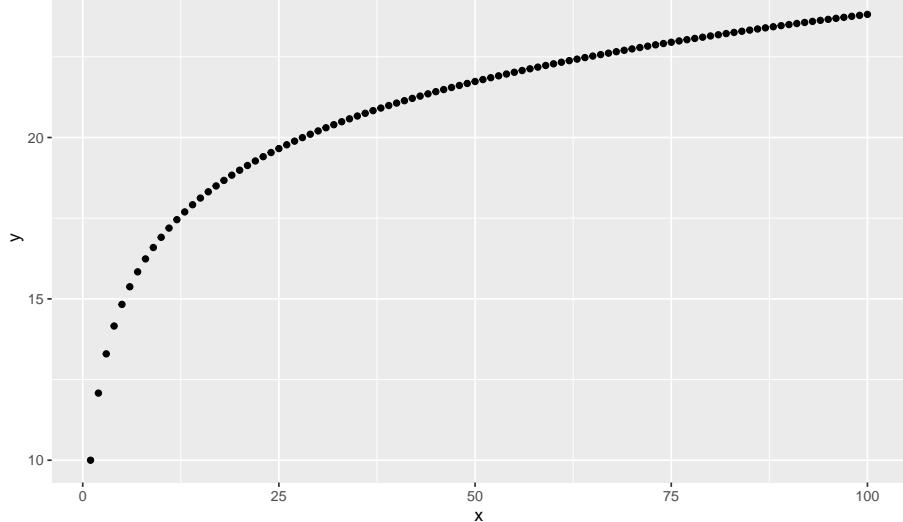
7.6.7 Variablenkalen

Mit `ggplot2` können Variablen einfach auf unterschiedliche Skalen übertragen werden. Diese Funktionalität kann mit der x-Achse oder mit der y-Achse verwendet werden, z.B.:

- `scale_x_log10()()`
- `scale_x_sqrt()()`
- `scale_x_reverse()()`

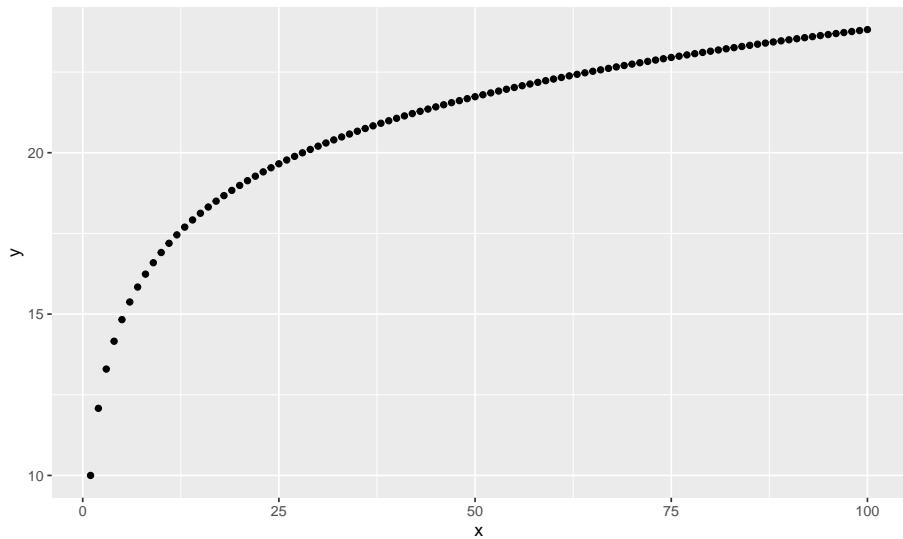
Beispielsweise können Sie in der folgenden Abbildung den Zusammenhang zweier Variablen betrachten. Dabei wird ein logarithmischer Zusammenhang dargestellt.

```
ggplot(data = data.frame(x = seq(1:100), y = 10 + 3 *  
  log(seq(1:100))),  
       aes(x = x, y = y)) +  
   geom_point()
```



Wenn Sie die x-Achse mittels `scale_x_log10()` umwandeln, könnten Sie sehen, dass der dargestellte Zusammenhang linear aussieht (d.H. der Zusammenhang ist linear in der Logarithmus-Skala).

```
ggplot(data = data.frame(x = seq(1:100), y = 10 + 3 *  
  log(seq(1:100))),  
       aes(x = x, y = y)) +  
   geom_point()
```



```
scale_x_log10()

## <ScaleContinuousPosition>
##   Range:
##   Limits:    0 --    1
```

7.7 Facetten

Facettierung bedeutet, dass ein Plot in mehrere Plots aufgespalten wird, abhängig von den Werten einer oder mehrerer diskreter Variablen. Der einfachste Weg geht über `facet_wrap()`. Beispielsweise kann man das Balkendiagramm von oben, welches die Art der Superhelden zeigt, um die größten Gegner dieser Helden erweitern. Erst definiert man in einem Vektor, wer die größten Helden und Bösewichte sind. Dann facettiert man das Balkendiagramm nach `Alignment`, es werden also unterschiedliche Grids für die Guten und die Bösen erstellt:

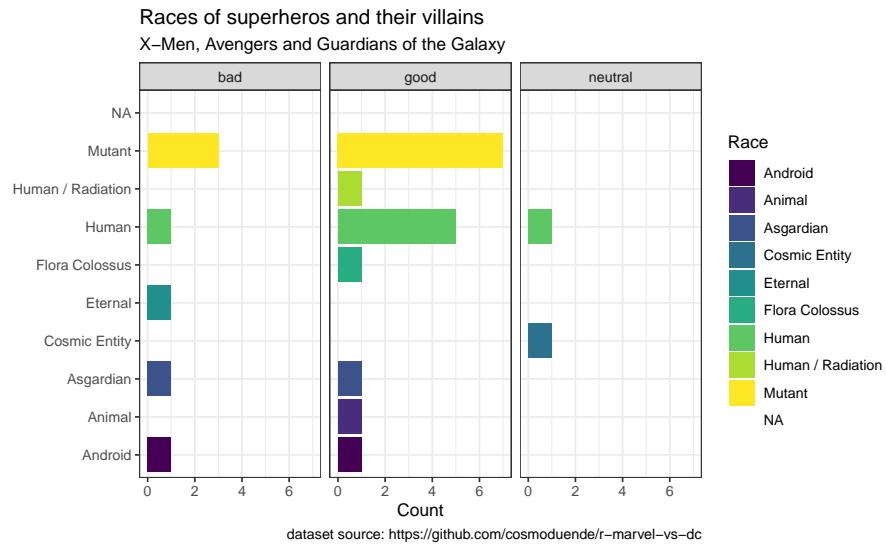
```
# Gegner der X-Men, der Guardians of the Galaxy und der Avengers
VXMen <- c("Magneto", "Gambit", "Apocalypse", "Mystique")
VGoG <- c("Ronin", "Thanos")
VAvengers <- c("Loki", "Ultron", "Doctor Doom", "Galactus")

# ein Vektor mit den coolsten Superhelden und ihren größten
# Gegnern
HerosVillains <- c(BestHeros, VXMen, VGoG, VAvengers)

MarvelDC %>%
  filter(Name %in% HerosVillains) %>% # Nur wenn in
  'HerosVillains'
  ggplot(aes(y = Race, # y soll 'Race' sein
             fill = Race)) + # vergabe Füllfarbe nach
  'Race'
  geom_bar() + # stelle als Balken dar
  theme_bw() + # in scharz-weiem Thema

# Beschrifte
  labs(title = "Races of superheros and their villains",
       subtitle = "X-Men, Avengers and Guardians of the Galaxy",
       x = "Count",
       y = "",
       caption = "dataset source:
https://github.com/cosmoduende/r-marvel-vs-dc" ) +
  scale_fill_viridis_d() + # verwende Farben aus
  viridis
```

```
facet_wrap(~ Alignment)          # trenne Grids nach
'Alignment'
```



Dieser Plot enthält viele der sonstigen Funktionalitäten von `ggplot2`. Auch wenn es anfangs wie unnötige Arbeit erscheint: gerade die richtige Beschriftung eines Plots ist unerlässlich! Farbskala und Thema hingegen sind nett zu haben, aber nicht unbedingt ab der ersten Version eines Plots erforderlich.