

# **Bio and Business Data Science**

Nicolas Meseth

# Inhalt

<b>Vorwort</b>	<b>4</b>
<b>I Fall: Campusbier</b>	<b>5</b>
Einführung . . . . .	6
Der Datensatz . . . . .	6
<b>1 CSV-Dateien einlesen</b>	<b>7</b>
1.1 Das CSV-Format . . . . .	7
1.1.1 CSV als weit verbreitetes Format . . . . .	7
1.1.2 CSV eignet sich für strukturierte Daten . . . . .	8
1.1.3 Beispiel einer CSV-Datei . . . . .	8
1.2 CSV-Daten laden mit {readr} . . . . .	9
1.2.1 Aus einer lokalen Datei . . . . .	9
1.2.2 Nur bestimmte Spalten laden . . . . .	9
1.2.3 Das Dezimaltrennzeichen bestimmen . . . . .	11
1.2.4 Datentypen beim Laden setzen . . . . .	11
1.2.5 Daten von einem Webserver laden . . . . .	13
Übungsaufgaben . . . . .	14
<b>2 Datensätze erkunden</b>	<b>15</b>
2.1 Sprechende Tibbles . . . . .	15
2.2 Anzahl Spalten und Zeilen ermitteln . . . . .	16
2.3 Datentypen . . . . .	17
2.4 Spaltennamen . . . . .	21
2.5 Stichproben betrachten . . . . .	23
2.6 Häufigkeiten erfassen . . . . .	25
Übungsaufgaben . . . . .	27
<b>3 Spalten auswählen</b>	<b>29</b>
3.1 Angabe von Spaltennamen . . . . .	29
3.2 Ausschließen von Spalten . . . . .	29
3.3 Spalten anhand eines Namensmuster selektieren . . . . .	30
3.4 Anhand des Datentyps einer Spalte . . . . .	32
3.5 Spalten über deren Position auswählen . . . . .	35

3.6 Spalten über eine Menge auswählen . . . . .	37
<b>4 Spalten hinzufügen und verändern</b>	<b>39</b>
<b>Quellen</b>	<b>40</b>

# Vorwort

Die vorliegende Webseite stellt das Skript für meine Lehrveranstaltungen zum Thema Data Science dar. Es ist momentan im Aufbau und es kommen wöchentlich neue Inhalte hinzu. Zu einem späteren Zeitpunkt soll es in Form eines Buches veröffentlicht werden. Bis dahin bitte ich um Geduld.

Es gibt eine PDF-Version, die hier heruntergeladen werden kann: [PDF-Version “Bio and Business Data Science”](#).

Für Hinweise zu Fehlern bin ich dankbar. Am besten per E-Mail unter [n.meseth@hs-osnabrueck.de](mailto:n.meseth@hs-osnabrueck.de) oder auf [Twitter](#).

Viel Spaß beim Lesen

Prof. Dr. Nicolas Meseth

## **Part I**

# **Fall: Campusbier**

## Einführung

Die Campusbier-Fallstudie ist in dem Sinne eine besondere Fallstudie im Rahmen dieses Buches, als dass sie als Einführung in das Basishandwerkszeug mit R dient. Wenn die Zeit für nur eine Fallstudie ausreicht und man Anfänger mit der Datenanalyse mit R ist, dann ist diese Fallstudie die richtige. Sie steht deshalb auch am Anfang des Buches und dient in allen meinen Modulen zur Datenanalyse mit R als Gegenstand der Einführung.

Das Campusbier-Projekt beschäftigt sich mit der Vermarktung des hochschuleigenen Bieres, das 2009 von Studierenden auf der Versuchsbrauanlage der Hochschule Osnabrück entwickelt wurde. Im Jahr 2019 wurde das Bier erstmals in größerer Menge gebraut und in Flaschen abgefüllt, so dass es der Osnabrücker Bevölkerung zugänglich gemacht werden konnte. Zuvor war das Bier nur intern in Fässern erhältlich, die etwa für Veranstaltungen am Campus der Hochschule verwendet wurden. Die erste öffentliche Verkaufsrunde im Mai/Juni 2019 war ein voller Erfolg, was zu einer dauerhaften Verfügbarkeit des Bieres über den Onlie-Shop [campusbier.de](https://campusbier.de) führte. Seitdem sind in vielen Projekten weitere Aktionen und Produkte hinzugekommen, deren Verkäufe sich im vorliegenden Datensatz widerspiegeln.

## Der Datensatz

Der Datensatz besteht aus den Informationen zu den knapp 3000 Bestellungen, die in den Jahren 2019 bis 2022 über den Online-Shop aufgegeben wurden. Neben den Metadaten einer Bestellung, wie der Bestellzeitpunkt, der Kunde oder die Zahlungsart, gibt es auch die Informationen über die gekauften Produkte, also den Warenkorb jeder Bestellung. Diese Daten liegen als CSV-Exporte aus dem E-Commerce-System Shopify vor. Der Datensatz besteht aus den beiden Dateien `orders.csv` und `line_items.csv`. Um dem Datenschutz Rechnung zu tragen, wurden sämtliche persönliche Daten der Kunden entfernt. Jeder Kunde ist nur über eine technische Nummer identifizierbar, die in jeder Bestellung angegeben ist. Haben mehrere Bestellungen die gleiche Kundennummer, so stammen diese alle vom selben Kunden.



Ladet euch am besten jetzt die beiden Dateien in euer Arbeitsverzeichnis herunter. Hier die Links zum Download (rechte Maustaste, "Link speichern unter"):

- [orders.csv](#)
- [line\\_items.csv](#)

# 1 CSV-Dateien einlesen

Im ersten Schritt jeder Datenanalyse müssen wir unserem Computer den Datensatz zur Verfügung stellen. Wir sprechen dabei auch vom *Laden* des Datensatzes. Dabei sagen wir dem Computer, wo die Daten zu finden sind, und dass er sie für den schnelleren Zugriff in seinen Arbeitsspeicher holen soll.

Daten liegen in den meisten Fällen in Form von Dateien vor. In manchen Fällen sind sie auch in einer Datenbank gespeichert. Im Fall einer Datei kann ein Datensatz in unterschiedlichen *Formaten* darin gespeichert werden. Ein gängiges Format ist das CSV-Format, das auf einfachen Textdateien basiert.

## 1.1 Das CSV-Format

### 1.1.1 CSV als weit verbreitetes Format

Textdateien bieten sich für die Speicherung von Daten an, weil sie auf jedem Betriebssystem mit einem einfachen Texteditor betrachtet und bearbeitet werden können. Das ermöglicht das schnelle Teilen von Daten und somit die Zusammenarbeit. Auch für den Datenaustausch zwischen verschiedenen Informationssystemen wird häufig ein textbasiertes Format verwendet, um spezifische Formate der jeweiligen Hersteller, wie etwa proprietäre Datenbanken, zu überbrücken. Deshalb bieten die meisten Informationssysteme Schnittstellen für den Export und Import von Textdateien an. Speziell das CSV-Format ist hier sehr beliebt, aus guten Gründen:

- Die Verwendung von einfachen Textdateien erlaubt die Speicherung und Verarbeitung auf unterschiedlichen Umgebungen wie Windows, macOS oder Linux.
- Das Format ist einfach zu verstehen und auch für Menschen lesbar.
- CSV ist ein offenes Format, d. h. es gibt keine Organisation, die daran die Rechte besitzt und es kann daher von jeder Software verwendet werden. Es gab lange nicht einmal eine offizielle Spezifikation des Formats. Mittlerweile gibt es eine Spezifikation als offizieller [MIME Type](#).

Auch das E-Commerce-System Shopify, aus dem die vorliegenden Verkaufsdaten stammen, bietet eine Möglichkeit zum Exportieren von Textdateien im sogenannten CSV-Format an.

### 1.1.2 CSV eignet sich für strukturierte Daten

CSV steht für *Comma Separated Values* und beschreibt ein Format, um *strukturierte Daten* in einer Textdatei abzuspeichern. Ihr erkennt eine Textdatei im CSV-Format an der Endung `.csv`.

Das CSV-Format basiert auf einfachen Textkodierungen, häufig im UTF-8 oder ASCII-Kodierungssystem (letzteres immer seltener wegen der geringen Anzahl verfügbarer Zeichen), die mit fast jedem Werkzeug und Editor gelesen und bearbeitet werden können. Das CSV-Format speichert strukturierte Daten in einer tabellarischen Form, ähnlich wie in Spreadsheets. Die erste Zeile einer CSV-Datei ist üblicherweise der sogenannte Header (Kopfzeile) und beinhaltet die Spaltennamen mit Komma oder Semikolon (Trennzeichen) voneinander getrennt. Jede weitere Zeile stellt eine Beobachtung (Englisch: *observation* oder *case*) oder auch Datensatz (Englisch: *record*) dar. Jeder Datensatz enthält für die im Header definierten Attribute (oder Spalten) die beobachteten Werte, die durch das gleiche Trennzeichen voneinander getrennt sind. Es muss nicht jeder Spaltenwert existieren. Sollte ein Wert für eine Beobachtung nicht vorhanden sein, so wird nach dem Komma nichts eingetragen und es folgen zwei Kommata nacheinander. In R werden fehlende Werte beim Einlesen erkannt und mit `NA` gekennzeichnet.

Die Verwendung des Komma als Trennzeichen in CSV-Dateien ist keineswegs verbindlich, auch wenn es Bestandteil des Namens ist. Generell kann jedes Symbol verwendet werden. Häufige Alternativen sind das Semikolon, Leerzeichen oder ein Tabstop. Letzteres wird oft mit der eigenen Endung `.tsv` für *Tab Separated Values* gespeichert.

### 1.1.3 Beispiel einer CSV-Datei

Der Ausschnitt unten zeigt die ersten vier Zeilen der `orders.csv`. Die erste Zeile enthält die Namen der hier gezeigten vier Spalten (der Datensatz hat mehr Spalten, das ist nur ein Auszug), die mit einem Komma voneinander getrennt sind. Darunter folgen drei beispielhafte Datensätze:

```
id,order_id,name,order_number,app_id,created_at
1130007101519,B1014,1014,580111,2019-05-24T14:59:16+02:00
1130014965839,B1015,1015,580111,2019-05-24T15:09:08+02:00
1130026958927,B1016,1016,580111,2019-05-24T15:22:41+02:00
...
```



## 1.2 CSV-Daten laden mit {readr}

### 1.2.1 Aus einer lokalen Datei

Für das Laden von Datensätzen aus CSV-Dateien bietet das Tidyverse ein Paket namens {readr} an. Dieses wird automatisch mit dem {tidyverse}-Paket mitgeladen. Das Paket bietet für CSV-Dateien, bei denen das Komma als Trennzeichen verwendet wird, die Funktion `read_csv()` an:

```
orders <- read_csv("./data/orders.csv")
```

💡 In einigen Ländern Europas wird häufig ein Semikolon als Trennzeichen und ein Komma als Dezimaltrennzeichen verwendet (wie etwa in Deutschland). Für diesen Fall bietet {readr} die Funktion `read_csv2`. Um es selbst in der Hand zu haben, kann die Funktion `read_delim` verwendet werden und über den Parameter `delim` das Trennzeichen manuell eingestellt werden.

Auch der R-Basisumfang bietet eine ähnliche Funktion für genau diesen Anwendungsfall an. Diese heisst `read.csv()`, man achte hier auf das Detail: Statt eines Unterstrichs `_` wird bei der R-Basisfunktion ein Punkt `.` zwischen den beiden Wörtern `read` und `csv` verwendet. Wenn ihr mit dem Tidyverse und mit Tibbles arbeitet (wie in diesem Buch durchgängig), dann achtet darauf, stets die {readr}-Funktion `read_csv()` zu verwenden, weil nur diese die Daten als Tibble zurückgibt und zudem einige paar nützliche Zusatzfunktionen bietet.

### 1.2.2 Nur bestimmte Spalten laden

Die `read_csv()`-Funktion erlaubt direkt beim Laden der Daten eine Auswahl der Spalten vorzunehmen. Ich empfehle hier immer die Verwendung der `select`-Funktion, die wir im nächsten Abschnitt kennenlernen werden. Dennoch möchte ich kurz demonstrieren, wie das Filtern der Spalten direkt beim Laden funktioniert.

Der folgende Code lädt die Spalten `order_id`, `name`, sowie alle Spalten, deren Name mit "customer" beginnen:

```
orders <- read_csv("./data/orders.csv", col_select = c(order_id, name, starts_with("customer")))
```

Im Ergebnis ist der resultierende Tibble dann sehr viel schmaler und beinhaltet nur die gewünschten Spalten:

```
colnames(orders)
```

```

[1] "order_id"
[2] "name"
[3] "customer_id"
[4] "customer_accepts_marketing"
[5] "customer_accepts_marketing_updated_at"
[6] "customer_marketing_opt_in_level"
[7] "customer_sms_marketing_consent"
[8] "customer_created_at"
[9] "customer_updated_at"
[10] "customer_gender"
[11] "customer_is_hsos"
[12] "customer_state"
[13] "customer_orders_count"
[14] "customer_total_spent"
[15] "customer_last_order_id"
[16] "customer_note"
[17] "customer_verified_email"
[18] "customer_tax_exempt"
[19] "customer_tags"
[20] "customer_last_order_name"

```

Hat man nur eine Filterbedingung, so kann man sich die `c()`-Funktion auch sparen:

```
orders <- read_csv("../data/orders.csv", col_select = starts_with("shipping"))
```

Das Ergebnis:

```
colnames(orders)
```

```

[1] "shipping_address_city"      "shipping_address_zip"
[3] "shipping_address_country"   "shipping_address_latitude"
[5] "shipping_address_longitude"

```

Ist man auf möglichst wenige Zeilen Code aus, so kann die Verwendung des `col_types`-Parameters durchaus Sinn ergeben. Man könnte den gleichen Effekt auch mit einem anschließenden `select` erzielen:

```

orders <- read_csv("../data/orders.csv") %>%
  select(starts_with("shipping"))

```

Mehr zur Auswahl von Spalten mit `{dplyr}` erfahrt ihr in Kapitel 3.

### 1.2.3 Das Dezimaltrennzeichen bestimmen

Wollen wir nur das Dezimaltrennzeichen ändern, weil wir zum Beispiel Daten aus einem deutschen System exportiert haben, das ein Komma verwendet, so können wir das über den `locale`-Parameter erreichen. Im Beispiel unten verwenden wir zusätzlich das Pipe-Symbol `|` als Trennzeichen. Beides können wir über `read_delim` einstellen:

```
# Daten in sales.csv
# year,month,turnover
# 2022|01|2700,85
# 2022|02|2910,10
# 2022|03|1802,37

turnover <- read_delim("sales.csv",
                      delim = "|",
                      locale = locale(decimal_mark = ","))
```

Das Ergebnis:

```
turnover
```

```
# A tibble: 3 x 3
  year month turnover
<dbl> <chr>   <dbl>
1  2022  01      2701.
2  2022  02      2910.
3  2022  03      1802.
```

### 1.2.4 Datentypen beim Laden setzen

Beim Laden mit `read_csv` wird möglicherweise der Datentyp einiger Spalten falsch erkannt. Die Funktion betrachtet die ersten Zeilen und ermittelt darauf basierend den Datentyp. Wird dieser falsch erkannt, so können wir den Datentyp für Spalten explizit angeben. Wie auch das Auswählen von Spalten können wir den Datentyp schon beim Laden mitgeben. Alternativ können wir den Datentyp auch später mit `mutate` noch verändern.

💡 Manchmal hilft es schon, den Parameter `guess_max` auf einen größeren Wert zu setzen. Dieser bestimmt, wie viele Werte die Funktion betrachtet, um daraus den Datentyp einer Spalte abzuleiten. Standardmäßig ist dieser Wert auf 1000 begrenzt.

In unserem Datensatz `orders.csv` sind tatsächlich einige Datentypen falsch erkannt worden. Etwa die Spalte `order_id`, die als Datentyp `double` erkannt wird. Auch wenn die Werte allesamt als Dezimalzahlen betrachtet werden können, handelt es sich bei einer Bestellnummer nicht um eine Zahl in dem Sinne, dass man damit rechnen möchte. Es ist vielmehr eine Zeichenfolge, die per Konvention nur aus Zahlen besteht (aber nicht müsste). Der Datentyp `character` wäre somit angebrachter. Das können wir wie folgt ändern:

```
orders <- read_csv("./data/orders.csv",
                  col_types = list("order_id" = col_character()))

# Die Spalte order_id ist jetzt vom Typ <chr>
orders %>%
  select(order_id)

# A tibble: 2,874 x 1
#   order_id
#   <chr>
1 1130007101519
2 1130014965839
3 1130026958927
4 1130030563407
5 1130038853711
6 1130045964367
7 1130050519119
8 1130060283983
9 1130102194255
10 1130106880079
# ... with 2,864 more rows
```

Das Gleiche können wir mit beliebig vielen Spalten gleichzeitig durchführen. Denn auch die beiden Spalten mit Postleitzahlen wurden fälschlicherweise als `double` erkannt:

```
orders <- read_csv("./data/orders.csv",
                  col_types = list(
                    "order_id" = col_character(),
                    "app_id" = col_character(),
                    "billing_address_zip" = col_character(),
                    "shipping_address_zip" = col_character()
                  )
)
```

```
# Die Spalte order_id ist jetzt vom Typ <chr>
orders %>%
  select(order_id, app_id, billing_address_zip, shipping_address_zip)

# A tibble: 2,874 x 4
  order_id      app_id billing_address_zip shipping_address_zip
  <chr>         <chr>    <chr>                <chr>
1 1130007101519 580111 49610                <NA>
2 1130014965839 580111 49088                49088
3 1130026958927 580111 49080                <NA>
4 1130030563407 580111 45888                <NA>
5 1130038853711 580111 49356                <NA>
6 1130045964367 580111 49477                <NA>
7 1130050519119 580111 49090                49090
8 1130060283983 580111 48356                48356
9 1130102194255 580111 49090                49090
10 1130106880079 580111 49090                <NA>
# ... with 2,864 more rows
```

## 1.2.5 Daten von einem Webserver laden

Die CSV-Datei muss nicht lokal auf dem eigenen Rechner vorliegen, sondern kann mit `read_csv` über das HTTP-Protokoll direkt von einem Webserver im Internet abgerufen werden. Dabei wird die URL anstelle des lokalen Dateinamens der Funktion übergeben. Der Code unten lädt die tagesaktuelle Version des Covid-19-Datensatzes, der auf den Servern von [Our World in Data](https://covid.ourworldindata.org/) gehostet wird:

```
covid <- read_csv("https://covid.ourworldindata.org/data/owid-covid-data.csv")

Rows: 221029 Columns: 67
-- Column specification -----
Delimiter: ","
chr   (4): iso_code, continent, location, tests_units
dbl   (62): total_cases, new_cases, new_cases_smoothed, total_deaths, new_dea...
date  (1): date

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

## Übungsaufgaben

1. Ladet den tagesaktuellen Datensatz für die Covid-Daten, lasst aber nur die Spalten `date`, `location` und `new_cases_smoothed_per_million` im Ergebnis!
2. Korrigiert alle falsch als `<dbl>` erkannten ID-Spalten im Campusbier-Verkaufsdatensatz direkt beim Laden der Daten und wandelt sich in `<chr>` um.

## 2 Datensätze erkunden

### 2.1 Sprechende Tibbles

Alleine beim Aufruf seines Namens gibt ein Tibble in der Konsole viele seiner Informationen preis:

```
orders
```

```
# A tibble: 2,874 x 68
  order_id name order~1 app_id created_at updated_at test
  <dbl> <chr> <dbl> <dbl> <dtm> <dtm> <lgl>
1 1.13e12 B1014 1014 580111 2019-05-24 12:59:16 2019-06-19 13:23:26 FALSE
2 1.13e12 B1015 1015 580111 2019-05-24 13:09:08 2019-06-21 14:40:07 FALSE
3 1.13e12 B1016 1016 580111 2019-05-24 13:22:41 2019-06-21 12:35:23 FALSE
4 1.13e12 B1017 1017 580111 2019-05-24 13:27:43 2019-06-21 14:27:18 FALSE
5 1.13e12 B1018 1018 580111 2019-05-24 13:36:46 2019-06-21 12:11:57 FALSE
6 1.13e12 B1019 1019 580111 2019-05-24 13:44:41 2019-06-21 14:37:21 FALSE
7 1.13e12 B1020 1020 580111 2019-05-24 13:49:21 2019-06-21 12:25:16 FALSE
8 1.13e12 B1021 1021 580111 2019-05-24 13:59:57 2019-06-21 11:49:47 FALSE
9 1.13e12 B1022 1022 580111 2019-05-24 14:43:53 2019-06-19 14:12:38 FALSE
10 1.13e12 B1023 1023 580111 2019-05-24 14:48:16 2019-06-21 15:54:24 FALSE
# ... with 2,864 more rows, 61 more variables: current_subtotal_price <dbl>,
# current_total_price <dbl>, current_total_discounts <dbl>,
# current_total_duties_set <dbl>, total_discounts <dbl>,
# total_line_items_price <dbl>, total_outstanding <dbl>, total_price <dbl>,
# total_tax <dbl>, total_tip_received <dbl>, taxes_included <lgl>,
# discount_codes <chr>, financial_status <chr>, fulfillment_status <chr>,
# source_name <chr>, landing_site <chr>, landing_site_ref <chr>, ...
```

Neben den ersten paar Zeilen als Vorschau gibt ein Tibble auch die Gesamtzahl an Zeilen und Spalten aus. Hier sind es 2874 Zeilen und 68 Spalten. Darunter folgt eine mit Kommata getrennte Auflistung der Spaltennamen und ihren Datentypen. Diese Liste wird aber nach wenigen Zeilen abgebrochen, um die Konsole nicht mit Text zu überladen.

Versucht das einmal: Ladet die CSV-Datei statt mit `read_csv` mit der Funktion `read.csv` aus dem Basis-R. Gebt jetzt den Namen des Dataframes in die Konsole ein und drückt Enter. Was ist der Unterschied bei der Ausgabe? Was gefällt euch besser?

## 2.2 Anzahl Spalten und Zeilen ermitteln

Ein Tibble gibt es uns freiwillig Auskunft über seine Dimensionierung, wenn wir seinen Namen aufrufen. Wir können diese Werte auch dediziert ermitteln, falls wir mit diesen Information etwa weitere Berechnungen in unserem Skript durchführen wollen.

Mit `ncol` erhalten wir die Anzahl Spalten:

```
orders %>%  
  ncol()
```

```
[1] 68
```

Mit `nrow` entsprechend die Anzahl Zeilen:

```
orders %>%  
  nrow()
```

```
[1] 2874
```

Wir könnten mit beiden Werten die Anzahl Zellen errechnen:

```
cols <- ncol(orders)  
rows <- nrow(orders)  
cells <- cols * rows  
cells
```

```
[1] 195432
```

Mit der Funktion `dim` erhalten wir einen Vektor mit beiden Informationen:

```
orders %>%  
  dim()
```



```
[1] 2874    68
```

Der erste Wert steht für die Anzahl Zeilen, der zweite für die Spalten.

Wenn wir als Ergebnis statt eines Vektors lieber ein Tibble hätten, können wir uns für die Anzahl Zeilen mit `count` helfen:

```
orders %>%
  count()

# A tibble: 1 x 1
      n
  <int>
1  2874
```

Wir können auch `nrow` und `ncol` als Spalten in einen neuen Dataframe (Tibble) verwenden und darauf basierend eine weitere, berechnete Spalte erstellen:

```
tibble(
  number_cols = ncol(orders),
  number_rows = nrow(orders),
  number_cells = number_cols * number_rows
)

# A tibble: 1 x 3
  number_cols number_rows number_cells
    <int>         <int>         <int>
1         68        2874        195432
```

## 2.3 Datentypen

Die schnellste und, wie ich finde, einfachste Möglichkeit, einen Überblick über die Datentypen in einem Tibble zu erhalten, ist die `glimpse`-Funktion:

```
orders %>%
  glimpse()
```

Rows: 2,874

Columns: 68

\$ order_id	<dbl> 1.130007e+12, 1.130015e+12, 1.13~
\$ name	<chr> "B1014", "B1015", "B1016", "B101~
\$ order_number	<dbl> 1014, 1015, 1016, 1017, 1018, 10~
\$ app_id	<dbl> 580111, 580111, 580111, 580111, ~
\$ created_at	<dtm> 2019-05-24 12:59:16, 2019-05-24~
\$ updated_at	<dtm> 2019-06-19 13:23:26, 2019-06-21~
\$ test	<lgl> FALSE, FALSE, FALSE, FALSE, FALS~
\$ current_subtotal_price	<dbl> 94.66, 32.22, 30.22, 32.22, 30.2~
\$ current_total_price	<dbl> 94.66, 32.22, 30.22, 32.22, 30.2~
\$ current_total_discounts	<dbl> 2, 0, 2, 0, 2, 2, 2, 0, 0, 0, 0, ~
\$ current_total_duties_set	<dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
\$ total_discounts	<dbl> 2, 0, 2, 0, 2, 2, 2, 0, 0, 0, 0, ~
\$ total_line_items_price	<dbl> 96.66, 32.22, 32.22, 32.22, 32.2~
\$ total_outstanding	<dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
\$ total_price	<dbl> 94.66, 32.22, 30.22, 32.22, 30.2~
\$ total_tax	<dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
\$ total_tip_received	<dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
\$ taxes_included	<lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TR~
\$ discount_codes	<chr> "DCBPXGJB1JGA", NA, "KYOD5MNEZBM~
\$ financial_status	<chr> "paid", "paid", "paid", "paid", ~
\$ fulfillment_status	<chr> "fulfilled", "fulfilled", "fulfi~
\$ source_name	<chr> "web", "web", "web", "web", "web~
\$ landing_site	<chr> "/password", "/wallets/checkouts~
\$ landing_site_ref	<chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
\$ location_id	<dbl> NA, NA, NA, NA, NA, NA, NA, NA, ~
\$ note	<chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
\$ tags	<chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
\$ processed_at	<dtm> 2019-05-24 12:59:15, 2019-05-24~
\$ processing_method	<chr> "direct", "express", "express", ~
\$ payment_details_gateway	<chr> "shopify_payments", "paypal", "p~
\$ payment_details_credit_card_company	<chr> "Mastercard", NA, NA, NA, NA, NA~
\$ customer_id	<dbl> 1.861678e+12, 1.856892e+12, 1.87~
\$ customer_accepts_marketing	<dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
\$ customer_accepts_marketing_updated_at	<dtm> 2019-05-21 13:38:54, 2019-05-24~
\$ customer_marketing_opt_in_level	<chr> "single_opt_in", "single_opt_in"~
\$ customer_sms_marketing_consent	<lgl> NA, NA, NA, NA, NA, NA, NA, NA, ~
\$ customer_created_at	<dtm> 2019-05-13 14:41:16, 2019-05-10~
\$ customer_updated_at	<dtm> 2022-02-23 13:50:50, 2022-06-21~
\$ customer_gender	<chr> "f", "m", "m", "m", "f", "m", "m~
\$ customer_is_hsos	<dbl> 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, ~
\$ customer_state	<chr> "disabled", "disabled", "enabled~

```

$ customer_orders_count      <dbl> 7, 10, 3, 1, 2, 1, 1, 1, 3, 6, 1~
$ customer_total_spent      <dbl> 345.22, 338.40, 95.17, 32.22, 66~
$ customer_last_order_id    <dbl> 4.640512e+12, 5.053779e+12, 1.95~
$ customer_note             <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ customer_verified_email   <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,~
$ customer_tax_exempt       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ customer_tags             <chr> "password page, prospect, umfrag~
$ customer_last_order_name  <chr> "41005353622", "41005379922", "1~
$ campaign_tag              <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ shipping_address_city     <chr> NA, "Osnabrück", NA, NA, NA, NA,~
$ shipping_address_zip      <dbl> NA, 49088, NA, NA, NA, NA, 49090~
$ shipping_address_country  <chr> NA, "Germany", NA, NA, NA, NA, "~
$ shipping_address_latitude <dbl> NA, 52.29756, NA, NA, NA, NA, 52~
$ shipping_address_longitude <dbl> NA, 8.058480, NA, NA, NA, NA, 8.~
$ billing_address_city      <chr> "Quakenbrück", "Osnabrück", "Osn~
$ billing_address_zip       <dbl> 49610, 49088, 49080, 45888, 4935~
$ billing_address_country   <chr> "Germany", "Germany", "Germany",~
$ billing_address_company   <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ billing_address_latitude  <dbl> 52.67033, 52.29756, 52.26469, 51~
$ billing_address_longitude <dbl> 7.959575, 8.058480, 8.035603, 7.~
$ client_details_browser_ip <chr> "131.173.97.189", "131.173.96.36~
$ client_details_browser_height <dbl> 665, 1010, 657, 600, 560, 553, 5~
$ client_details_browser_width <dbl> 1263, 2048, 1349, 360, 360, 375,~
$ client_details_user_agent <chr> "Mozilla/5.0 (Windows NT 10.0; W~
$ cancel_reason             <chr> NA, NA, NA, NA, NA, NA, NA, NA, ~
$ cancelled_at              <dtm> NA, NA, NA, NA, NA, NA, NA, NA,~
$ closed_at                 <dtm> 2019-06-19 13:23:26, 2019-06-21~

```

Neben dem Namen der Spalten erhalten wir den Datentyp sowie eine Vorschau der ersten in jeder Spalte enthaltenen Werte. Auch die Anzahl Zeilen und Spalten nennt uns `glimpse`.

Etwas technischer geht es mit der `spec`-Funktion. Diese liefert statt des Datentyp-Kürzels auch gleich die genaue Spezifikation als R-Funktion zurück:

```
spec(orders)
```

```

cols(
  order_id = col_double(),
  name = col_character(),
  order_number = col_double(),
  app_id = col_double(),
  created_at = col_datetime(format = ""),

```

```

updated_at = col_datetime(format = ""),
test = col_logical(),
current_subtotal_price = col_double(),
current_total_price = col_double(),
current_total_discounts = col_double(),
current_total_duties_set = col_double(),
total_discounts = col_double(),
total_line_items_price = col_double(),
total_outstanding = col_double(),
total_price = col_double(),
total_tax = col_double(),
total_tip_received = col_double(),
taxes_included = col_logical(),
discount_codes = col_character(),
financial_status = col_character(),
fulfillment_status = col_character(),
source_name = col_character(),
landing_site = col_character(),
landing_site_ref = col_character(),
location_id = col_double(),
note = col_character(),
tags = col_character(),
processed_at = col_datetime(format = ""),
processing_method = col_character(),
payment_details_gateway = col_character(),
payment_details_credit_card_company = col_character(),
customer_id = col_double(),
customer_accepts_marketing = col_double(),
customer_accepts_marketing_updated_at = col_datetime(format = ""),
customer_marketing_opt_in_level = col_character(),
customer_sms_marketing_consent = col_logical(),
customer_created_at = col_datetime(format = ""),
customer_updated_at = col_datetime(format = ""),
customer_gender = col_character(),
customer_is_hsos = col_double(),
customer_state = col_character(),
customer_orders_count = col_double(),
customer_total_spent = col_double(),
customer_last_order_id = col_double(),
customer_note = col_character(),
customer_verified_email = col_double(),
customer_tax_exempt = col_double(),
customer_tags = col_character(),

```

```

customer_last_order_name = col_character(),
campaign_tag = col_character(),
shipping_address_city = col_character(),
shipping_address_zip = col_double(),
shipping_address_country = col_character(),
shipping_address_latitude = col_double(),
shipping_address_longitude = col_double(),
billing_address_city = col_character(),
billing_address_zip = col_double(),
billing_address_country = col_character(),
billing_address_company = col_character(),
billing_address_latitude = col_double(),
billing_address_longitude = col_double(),
client_details_browser_ip = col_character(),
client_details_browser_height = col_double(),
client_details_browser_width = col_double(),
client_details_user_agent = col_character(),
cancel_reason = col_character(),
cancelled_at = col_datetime(format = ""),
closed_at = col_datetime(format = "")
)

```

Fällt uns auf, dass ein Datentyp falsch erkannt wurde, so können wir entweder unseren Ladevorgang anpassen und etwa den Datentyp direkt beim Lesen korrigieren. Alternativ verändern wir die Spalte mit `mutate`, was wir in Kapitel 4 lernen werden.

## 2.4 Spaltennamen

Ist man noch nicht lange mit einem Datensatz vertraut, ist eine Funktion für die Auflistung der Spaltennamen hilfreich:

```

orders %>%
  colnames()

```

```

[1] "order_id"
[2] "name"
[3] "order_number"
[4] "app_id"
[5] "created_at"
[6] "updated_at"

```

[7] "test"  
[8] "current\_subtotal\_price"  
[9] "current\_total\_price"  
[10] "current\_total\_discounts"  
[11] "current\_total\_duties\_set"  
[12] "total\_discounts"  
[13] "total\_line\_items\_price"  
[14] "total\_outstanding"  
[15] "total\_price"  
[16] "total\_tax"  
[17] "total\_tip\_received"  
[18] "taxes\_included"  
[19] "discount\_codes"  
[20] "financial\_status"  
[21] "fulfillment\_status"  
[22] "source\_name"  
[23] "landing\_site"  
[24] "landing\_site\_ref"  
[25] "location\_id"  
[26] "note"  
[27] "tags"  
[28] "processed\_at"  
[29] "processing\_method"  
[30] "payment\_details\_gateway"  
[31] "payment\_details\_credit\_card\_company"  
[32] "customer\_id"  
[33] "customer\_accepts\_marketing"  
[34] "customer\_accepts\_marketing\_updated\_at"  
[35] "customer\_marketing\_opt\_in\_level"  
[36] "customer\_sms\_marketing\_consent"  
[37] "customer\_created\_at"  
[38] "customer\_updated\_at"  
[39] "customer\_gender"  
[40] "customer\_is\_hsos"  
[41] "customer\_state"  
[42] "customer\_orders\_count"  
[43] "customer\_total\_spent"  
[44] "customer\_last\_order\_id"  
[45] "customer\_note"  
[46] "customer\_verified\_email"  
[47] "customer\_tax\_exempt"  
[48] "customer\_tags"  
[49] "customer\_last\_order\_name"

```
[50] "campaign_tag"
[51] "shipping_address_city"
[52] "shipping_address_zip"
[53] "shipping_address_country"
[54] "shipping_address_latitude"
[55] "shipping_address_longitude"
[56] "billing_address_city"
[57] "billing_address_zip"
[58] "billing_address_country"
[59] "billing_address_company"
[60] "billing_address_latitude"
[61] "billing_address_longitude"
[62] "client_details_browser_ip"
[63] "client_details_browser_height"
[64] "client_details_browser_width"
[65] "client_details_user_agent"
[66] "cancel_reason"
[67] "cancelled_at"
[68] "closed_at"
```

Diese Information liefert einem auch `glimpse`, wie oben gezeigt. Mit `colnames` bekommen wir die Spaltennamen als Vektor, was nützlich sein kann, um damit weiter arbeiten zu können.

## 2.5 Stichproben betrachten

Für einen ersten Eindruck der Daten reicht es oft schon, sich ein paar Zeilen anzeigen zu lassen. Das geht mit unterschiedlichen Funktionen:

```
# Zeigt standardmäßig die ersten 6 Zeilen an
orders %>%
  head()

# Zeigt die ersten 20 Zeilen an
orders %>%
  head(20)

# Wenn man mehr anzeigen möchte, kann print verwendet werden
orders %>%
  print(n = 100)

# Das Ganze geht auch von unten
```

```
orders %>%
  tail()
```

Möchte man nicht die oberen oder unteren Zeilen anzeigen, sondern eine zufällige Auswahl, so hilft `sample_n`:

```
orders %>%
  sample_n(size = 5)
```

```
# A tibble: 5 x 68
  order_id name order~1 app_id created_at updated_at test
  <dbl> <chr> <dbl> <dbl> <dtm> <dtm> <lgl>
1 1.14e12 B1121 1121 580111 2019-05-28 12:56:14 2019-06-21 12:58:16 FALSE
2 1.88e12 1681~ 1284 580111 2019-11-15 10:49:39 2019-12-13 13:45:02 FALSE
3 4.21e12 4100~ 2862 580111 2021-10-19 06:26:29 2021-11-09 12:47:25 FALSE
4 5.16e12 4100~ 3921 580111 2022-09-22 12:50:44 2022-09-22 12:50:47 FALSE
5 1.13e12 B1020 1020 580111 2019-05-24 13:49:21 2019-06-21 12:25:16 FALSE
# ... with 61 more variables: current_subtotal_price <dbl>,
# current_total_price <dbl>, current_total_discounts <dbl>,
# current_total_duties_set <dbl>, total_discounts <dbl>,
# total_line_items_price <dbl>, total_outstanding <dbl>, total_price <dbl>,
# total_tax <dbl>, total_tip_received <dbl>, taxes_included <lgl>,
# discount_codes <chr>, financial_status <chr>, fulfillment_status <chr>,
# source_name <chr>, landing_site <chr>, landing_site_ref <chr>, ...
```

Um einen bestimmten prozentualen Anteil an den gesamten Zeilen zufällig zu ermitteln, gibt es `sample_frac`:

```
# Gibt zufällig gewählte Zeilen zurück, die 1% entsprechen
orders %>%
  sample_frac(size = 0.01)
```

```
# A tibble: 29 x 68
  order_id name order~1 app_id created_at updated_at test
  <dbl> <chr> <dbl> <dbl> <dtm> <dtm> <lgl>
1 4.10e12 4100~ 2643 580111 2021-09-01 20:34:00 2021-09-14 12:00:55 FALSE
2 4.70e12 4100~ 3642 580111 2022-03-16 09:33:53 2022-03-18 14:15:58 FALSE
3 4.17e12 4100~ 2704 580111 2021-09-28 15:28:17 2021-10-07 14:57:05 FALSE
4 4.64e12 4100~ 3535 580111 2022-01-14 14:11:29 2022-01-14 15:08:22 FALSE
5 3.12e12 4100~ 2328 580111 2020-12-09 12:11:46 2020-12-20 15:20:52 FALSE
```



```

6   4.60e12 4100~    3332 580111 2021-12-10 07:31:59 2021-12-16 17:06:13 FALSE
7   1.88e12 1681~    1358 580111 2019-11-16 10:46:52 2019-12-13 15:37:20 FALSE
8   3.13e12 4100~    2353 580111 2020-12-11 17:16:20 2020-12-20 15:16:53 FALSE
9   3.12e12 4100~    2299 580111 2020-12-08 10:53:00 2020-12-15 10:23:14 FALSE
10  2.27e12 4100~    1640 580111 2020-05-13 13:28:09 2020-05-14 14:52:41 FALSE
# ... with 19 more rows, 61 more variables: current_subtotal_price <dbl>,
#   current_total_price <dbl>, current_total_discounts <dbl>,
#   current_total_duties_set <dbl>, total_discounts <dbl>,
#   total_line_items_price <dbl>, total_outstanding <dbl>, total_price <dbl>,
#   total_tax <dbl>, total_tip_received <dbl>, taxes_included <lgl>,
#   discount_codes <chr>, financial_status <chr>, fulfillment_status <chr>,
#   source_name <chr>, landing_site <chr>, landing_site_ref <chr>, ...

```

## 2.6 Häufigkeiten erfassen

Wir haben oben bereits `count` für das Zählen der Zeilen kennengelernt. Die Funktion eignet sich auch für das Zählen von Zeilen gruppiert nach einem Merkmal in den Daten:

```

orders %>%
  count(payment_details_gateway)

```

```

# A tibble: 4 x 2
  payment_details_gateway      n
  <chr>                  <int>
1 manual                  194
2 paypal                  1790
3 shopify_payments        889
4 <NA>                     1

```

Das geht auch sortiert nach Häufigkeit:

```

orders %>%
  count(payment_details_gateway, sort = TRUE)

```

```

# A tibble: 4 x 2
  payment_details_gateway      n
  <chr>                  <int>
1 paypal                  1790
2 shopify_payments        889

```

3 manual	194
4 <NA>	1

Mit dem bereits bekannten {janitor}-Paket erhalten wir eine Funktion, um für nominal skalierte Merkmale schnell die Häufigkeiten, sowohl absolut als auch prozentual, zu ermitteln:

```
library(janitor)

orders %>%
  tabyl(payment_details_gateway)
```

payment_details_gateway	n	percent	valid_percent
manual	194	0.0675017397	0.06752523
paypal	1790	0.6228253305	0.62304212
shopify_payments	889	0.3093249826	0.30943265
<NA>	1	0.0003479471	NA

Hier müssen wir das Sortieren manuell mit `arrange` vornehmen:

```
orders %>%
  tabyl(payment_details_gateway) %>%
  arrange(-n)
```

payment_details_gateway	n	percent	valid_percent
paypal	1790	0.6228253305	0.62304212
shopify_payments	889	0.3093249826	0.30943265
manual	194	0.0675017397	0.06752523
<NA>	1	0.0003479471	NA

Wenn wir eine zweite Variable hinzufügen, so erstellt `tabyl` eine Kreuztabelle mit den absoluten Häufigkeiten der jeweiligen Kombinationen:

```
orders %>%
  tabyl(payment_details_gateway, payment_details_credit_card_company)
```

payment_details_gateway	American Express	Mastercard	Visa	NA_
manual	0	0	0	194
paypal	0	1	3	1786
shopify_payments	14	372	303	200
<NA>	0	0	0	1

Wir können so erkennen, dass für PayPal-Zahlungen nur sehr selten ein Kreditkartenanbieter hinterlegt ist. Bei den Shopify-Payments hingegen ist das in den meisten Bestellungen der Fall. Dabei liegt Mastercard knapp vor Visa, American Express ist eher die Ausnahme.

💡 Es wäre doch interessant zu wissen, warum genau eine Bestellung keine Angabe zur Zahlungsart besitzt. Prüft doch mal nach, welche das ist und versucht die Frage zu beantworten.

## Übungsaufgaben

Speichert die Lösungen für die folgenden Aufgaben in einem neuen R-Skript. Vergesst nicht, die Datei zwischendurch zu speichern.

### 1. Dimensionen ermitteln

- Ermittelt die Anzahl Spalten und Zeilen des Datensatzes. Ihr werdet herausfinden, dass es dafür verschiedene Möglichkeiten gibt.
- Findet auch eine Möglichkeit, mit der ihr beide Größen auf jeweils einer Variable speichern könnt. Das kann zum Beispiel wichtig sein, wenn ihr mit den Größen im weiteren Skriptverlauf weiter arbeiten wollt.

### 2. Nützliche Funktionen

- Recherchiert und testet die folgenden Funktionen. Notiert euch in eigenen Worten, was jede Funktion zurückgibt:
  - `head()`
  - `tail()`
  - `print()`
  - `sample_n()`
  - `glimpse()`
  - `summary()`
  - `colnames()`
- Welche Parameter könnt ihr den Funktionen jeweils übergeben und wozu dienen sie?
- Wozu würdet ihr die Funktionen verwenden?

### 3. Datentypen ermitteln

- Findet einen Weg, um den Datentyp für jede Spalte auf der Konsole auszugeben!
- Welche Datentypen sind vertreten?
- Sind eurer Meinung nach alle Spalten korrekt erkannt worden?

#### 4. Das Paket `{skimr}`

- Installiert das Paket `{skimr}` und betrachtet nun die Funktion `skim()`.
- Was ermittelt die Funktion, wenn wir ihr einen Datensatz übergeben?
- Gebt mithilfe der Funktion nur den prozentualen Anteil fehlender Werte (NA) für jede Spalte auf der Konsole aus!

## 3 Spalten auswählen

### 3.1 Angabe von Spaltennamen

Die einfachste und intuitive Methode, um die Spalten eines Datensatzes einzuschränken, ist die Nennung der Spaltennamen. Dabei werden mehrere Spaltennamen mit einem Komma getrennt.

```
orders %>%  
  select(billing_address_zip, billing_address_city)
```

```
# A tibble: 2,874 x 2  
  billing_address_zip billing_address_city  
  <chr>              <chr>  
1 49610              Quakenbrück  
2 49088              Osnabrück  
3 49080              Osnabrück  
4 45888              Gelsenkirchen  
5 49356              Diepholz  
6 49477              Ibbenbüren  
7 49090              Osnabrück  
8 48356              Nordwalde  
9 49090              Osnabrück  
10 49090             Osnabrück  
# ... with 2,864 more rows
```

### 3.2 Ausschließen von Spalten

Statt eine Positivliste zu definieren, können wir auch Spalten über ihren Namen ausschließen, indem wir ein Minuszeichen voranstellen. Das folgende Beispiel wählt alle Spalten des Datensatzes, außer die beiden genannten:

```
orders %>%  
  select(-order_id, -order_number)
```

### 3.3 Spalten anhand eines Namensmuster selektieren

```
orders %>%  
  select(starts_with("shipping_"))
```

```
# A tibble: 2,874 x 5  
  shipping_address_city shipping_address_zip shipping_address~1 shipp~2 shipp~3  
  <chr>                <chr>                <chr>                <dbl>  <dbl>  
1 <NA>                <NA>                <NA>                NA      NA  
2 Osnabrück          49088              Germany            52.3    8.06  
3 <NA>                <NA>                <NA>                NA      NA  
4 <NA>                <NA>                <NA>                NA      NA  
5 <NA>                <NA>                <NA>                NA      NA  
6 <NA>                <NA>                <NA>                NA      NA  
7 Osnabrück          49090              Germany            52.3    8.00  
8 Nordwalde          48356              Germany            52.1    7.51  
9 Osnabrück          49090              Germany            52.3    8.06  
10 <NA>                <NA>                <NA>                NA      NA  
# ... with 2,864 more rows, and abbreviated variable names  
#   1: shipping_address_country, 2: shipping_address_latitude,  
#   3: shipping_address_longitude
```

```
orders %>%  
  select(ends_with("_at"))
```

```
# A tibble: 2,874 x 8  
  created_at      updated_at      processed_at  
  <dtm>          <dtm>          <dtm>  
1 2019-05-24 12:59:16 2019-06-19 13:23:26 2019-05-24 12:59:15  
2 2019-05-24 13:09:08 2019-06-21 14:40:07 2019-05-24 13:09:07  
3 2019-05-24 13:22:41 2019-06-21 12:35:23 2019-05-24 13:22:40  
4 2019-05-24 13:27:43 2019-06-21 14:27:18 2019-05-24 13:27:42  
5 2019-05-24 13:36:46 2019-06-21 12:11:57 2019-05-24 13:36:45  
6 2019-05-24 13:44:41 2019-06-21 14:37:21 2019-05-24 13:44:41  
7 2019-05-24 13:49:21 2019-06-21 12:25:16 2019-05-24 13:49:20  
8 2019-05-24 13:59:57 2019-06-21 11:49:47 2019-05-24 13:59:57  
9 2019-05-24 14:43:53 2019-06-19 14:12:38 2019-05-24 14:43:53  
10 2019-05-24 14:48:16 2019-06-21 15:54:24 2019-05-24 14:48:16  
# ... with 2,864 more rows, and 5 more variables:  
#   customer_accepts_marketing_updated_at <dtm>, customer_created_at <dtm>,
```

```
# customer_updated_at <dtm>, cancelled_at <dtm>, closed_at <dtm>
```

```
orders %>%
  select(contains("address"))
```

```
# A tibble: 2,874 x 11
```

	shipping_ad~1	shipp~2	shipp~3	shipp~4	shipp~5	billi~6	billi~7	billi~8	billi~9
	<chr>	<chr>	<chr>	<dbl>	<dbl>	<chr>	<chr>	<chr>	<chr>
1	<NA>	<NA>	<NA>	NA	NA	Quaken~	49610	Germany	<NA>
2	Osnabrück	49088	Germany	52.3	8.06	Osnabr~	49088	Germany	<NA>
3	<NA>	<NA>	<NA>	NA	NA	Osnabr~	49080	Germany	<NA>
4	<NA>	<NA>	<NA>	NA	NA	Gelsen~	45888	Germany	<NA>
5	<NA>	<NA>	<NA>	NA	NA	Diepho~	49356	Germany	<NA>
6	<NA>	<NA>	<NA>	NA	NA	Ibbenb~	49477	Germany	<NA>
7	Osnabrück	49090	Germany	52.3	8.00	Osnabr~	49090	Germany	<NA>
8	Nordwalde	48356	Germany	52.1	7.51	Nordwa~	48356	Germany	<NA>
9	Osnabrück	49090	Germany	52.3	8.06	Osnabr~	49090	Germany	<NA>
10	<NA>	<NA>	<NA>	NA	NA	Osnabr~	49090	Germany	<NA>

```
# ... with 2,864 more rows, 2 more variables: billing_address_latitude <dbl>,
```

```
# billing_address_longitude <dbl>, and abbreviated variable names
```

```
# 1: shipping_address_city, 2: shipping_address_zip,
```

```
# 3: shipping_address_country, 4: shipping_address_latitude,
```

```
# 5: shipping_address_longitude, 6: billing_address_city,
```

```
# 7: billing_address_zip, 8: billing_address_country,
```

```
# 9: billing_address_company
```

```
orders %>%
  select(matches("price"))
```

```
# A tibble: 2,874 x 4
```

	current_subtotal_price	current_total_price	total_line_items_price	total_price
	<dbl>	<dbl>	<dbl>	<dbl>
1	94.7	94.7	96.7	94.7
2	32.2	32.2	32.2	32.2
3	30.2	30.2	32.2	30.2
4	32.2	32.2	32.2	32.2
5	30.2	30.2	32.2	30.2
6	30.2	30.2	32.2	30.2
7	30.2	30.2	32.2	30.2
8	32.2	32.2	32.2	32.2

```

9          96.7          96.7          96.7          96.7
10         32.2          32.2          32.2          32.2
# ... with 2,864 more rows

```

### 3.4 Anhand des Datentyps einer Spalte

```

orders %>%
  select(where(is.logical))

```

```

# A tibble: 2,874 x 3
  test  taxes_included customer_sms_marketing_consent
<lgl> <lgl>          <lgl>
1 FALSE TRUE         NA
2 FALSE TRUE         NA
3 FALSE TRUE         NA
4 FALSE TRUE         NA
5 FALSE TRUE         NA
6 FALSE TRUE         NA
7 FALSE TRUE         NA
8 FALSE TRUE         NA
9 FALSE TRUE         NA
10 FALSE TRUE        NA
# ... with 2,864 more rows

```

```

# Verschiedene Optionen kombiniert
orders %>%
  select(where(is.numeric), -contains(c("discounts", "current_")))

```

```

# A tibble: 2,874 x 21
  order_number total~1 total~2 total~3 total~4 total~5 locat~6 custo~7 custo~8
      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1         1014      96.7        0      94.7        0        0      NA 1.86e12        1
2         1015      32.2        0      32.2        0        0      NA 1.86e12        1
3         1016      32.2        0      30.2        0        0      NA 1.87e12        1
4         1017      32.2        0      32.2        0        0      NA 1.87e12        1
5         1018      32.2        0      30.2        0        0      NA 1.87e12        1
6         1019      32.2        0      30.2        0        0      NA 1.87e12        1
7         1020      32.2        0      30.2        0        0      NA 1.87e12        1
8         1021      32.2        0      32.2        0        0      NA 1.88e12        1

```



```

 9      1022      96.7      0  96.7      0      0      NA 1.88e12      1
10      1023      32.2      0  32.2      0      0      NA 1.88e12      1
# ... with 2,864 more rows, 12 more variables: customer_is_hsos <dbl>,
#   customer_orders_count <dbl>, customer_total_spent <dbl>,
#   customer_last_order_id <dbl>, customer_verified_email <dbl>,
#   customer_tax_exempt <dbl>, shipping_address_latitude <dbl>,
#   shipping_address_longitude <dbl>, billing_address_latitude <dbl>,
#   billing_address_longitude <dbl>, client_details_browser_height <dbl>,
#   client_details_browser_width <dbl>, and abbreviated variable names ...

```

```
# Verschiedene Optionen kombiniert
```

```

orders %>%
  select(where(is.character),
         -ends_with("id"),
         -starts_with("shipping"),
         -starts_with("billing"),
         -starts_with("customer")
        )

```

```
# A tibble: 2,874 x 16
```

	name	discount_~1	finan~2	fulfi~3	sourc~4	landi~5	landi~6	note	tags	proce~7
	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>
1	B1014	DCBPXGJB1J~	paid	fulfil~	web	/passw~	<NA>	<NA>	<NA>	direct
2	B1015	<NA>	paid	fulfil~	web	/walle~	<NA>	<NA>	<NA>	express
3	B1016	KYOD5MNEZB~	paid	fulfil~	web	/	<NA>	<NA>	<NA>	express
4	B1017	<NA>	paid	fulfil~	web	/walle~	<NA>	<NA>	<NA>	express
5	B1018	DCBPXGJB1J~	paid	fulfil~	web	<NA>	<NA>	<NA>	<NA>	express
6	B1019	DCBPXGJB1J~	paid	fulfil~	web	<NA>	<NA>	<NA>	<NA>	express
7	B1020	DCBPXGJB1J~	paid	fulfil~	web	<NA>	<NA>	<NA>	<NA>	express
8	B1021	<NA>	paid	fulfil~	web	/	<NA>	<NA>	<NA>	express
9	B1022	<NA>	paid	fulfil~	web	/walle~	<NA>	<NA>	<NA>	express
10	B1023	<NA>	paid	fulfil~	web	<NA>	<NA>	<NA>	<NA>	express

```

# ... with 2,864 more rows, 6 more variables: payment_details_gateway <chr>,
#   payment_details_credit_card_company <chr>, campaign_tag <chr>,
#   client_details_browser_ip <chr>, client_details_user_agent <chr>,
#   cancel_reason <chr>, and abbreviated variable names 1: discount_codes,
#   2: financial_status, 3: fulfillment_status, 4: source_name,
#   5: landing_site, 6: landing_site_ref, 7: processing_method

```

```

orders %>%
  select(where(lubridate::is.Date))

```

```
# A tibble: 2,874 x 0
```

```
# Eigene Funktion, um Datums- und Datum + Zeitspalten zu ermitteln
is.DateTime <- function(x) {
  inherits(x, c("Date", "POSIXct"))
}

orders %>%
  select(where(is.DateTime))
```

```
# A tibble: 2,874 x 8
```

```
  created_at      updated_at      processed_at
  <dtm>          <dtm>          <dtm>
1 2019-05-24 12:59:16 2019-06-19 13:23:26 2019-05-24 12:59:15
2 2019-05-24 13:09:08 2019-06-21 14:40:07 2019-05-24 13:09:07
3 2019-05-24 13:22:41 2019-06-21 12:35:23 2019-05-24 13:22:40
4 2019-05-24 13:27:43 2019-06-21 14:27:18 2019-05-24 13:27:42
5 2019-05-24 13:36:46 2019-06-21 12:11:57 2019-05-24 13:36:45
6 2019-05-24 13:44:41 2019-06-21 14:37:21 2019-05-24 13:44:41
7 2019-05-24 13:49:21 2019-06-21 12:25:16 2019-05-24 13:49:20
8 2019-05-24 13:59:57 2019-06-21 11:49:47 2019-05-24 13:59:57
9 2019-05-24 14:43:53 2019-06-19 14:12:38 2019-05-24 14:43:53
10 2019-05-24 14:48:16 2019-06-21 15:54:24 2019-05-24 14:48:16
# ... with 2,864 more rows, and 5 more variables:
#   customer_accepts_marketing_updated_at <dtm>, customer_created_at <dtm>,
#   customer_updated_at <dtm>, cancelled_at <dtm>, closed_at <dtm>
```

```
# Weitere Datentypen, die in diesem Datensatz nicht auftreten
orders %>%
  select(where(is.factor))

orders %>%
  select(where(is.list))
```

## 3.5 Spalten über deren Position auswählen

```
# Die letzte Spalte im Datensatz
orders %>%
  select(last_col())
```

```
# A tibble: 2,874 x 1
  closed_at
  <dtm>
1 2019-06-19 13:23:26
2 2019-06-21 14:40:07
3 2019-06-21 12:35:22
4 2019-06-21 14:27:18
5 2019-06-21 12:11:57
6 2019-06-21 14:37:21
7 2019-06-21 12:25:16
8 2019-06-21 11:49:46
9 2019-06-19 14:12:38
10 2019-06-21 15:54:24
# ... with 2,864 more rows
```

```
# Die vorletzte Spalte im Datensatz
orders %>%
  select(last_col(2))
```

```
# A tibble: 2,874 x 1
  cancel_reason
  <chr>
1 <NA>
2 <NA>
3 <NA>
4 <NA>
5 <NA>
6 <NA>
7 <NA>
8 <NA>
9 <NA>
10 <NA>
# ... with 2,864 more rows
```

```
# Die erste Spalte im Datensatz
orders %>%
  select(1)
```

```
# A tibble: 2,874 x 1
  order_id
  <chr>
1 1130007101519
2 1130014965839
3 1130026958927
4 1130030563407
5 1130038853711
6 1130045964367
7 1130050519119
8 1130060283983
9 1130102194255
10 1130106880079
# ... with 2,864 more rows
```

```
orders %>%
  select(2:5)
```

```
# A tibble: 2,874 x 4
  name order_number app_id created_at
  <chr>      <dbl> <chr>  <dtm>
1 B1014      1014 580111 2019-05-24 12:59:16
2 B1015      1015 580111 2019-05-24 13:09:08
3 B1016      1016 580111 2019-05-24 13:22:41
4 B1017      1017 580111 2019-05-24 13:27:43
5 B1018      1018 580111 2019-05-24 13:36:46
6 B1019      1019 580111 2019-05-24 13:44:41
7 B1020      1020 580111 2019-05-24 13:49:21
8 B1021      1021 580111 2019-05-24 13:59:57
9 B1022      1022 580111 2019-05-24 14:43:53
10 B1023      1023 580111 2019-05-24 14:48:16
# ... with 2,864 more rows
```

```
# Alle Spalten bis auf die letzten 10
orders %>%
  select(1:last_col(10))
```

```
# A tibble: 2,874 x 58
  order_id name order~1 app_id created_at updated_at test
  <chr>    <chr>    <dbl> <chr>    <dtm>          <dtm>          <lgl>
1 113000710~ B1014    1014 580111 2019-05-24 12:59:16 2019-06-19 13:23:26 FALSE
2 113001496~ B1015    1015 580111 2019-05-24 13:09:08 2019-06-21 14:40:07 FALSE
3 113002695~ B1016    1016 580111 2019-05-24 13:22:41 2019-06-21 12:35:23 FALSE
4 113003056~ B1017    1017 580111 2019-05-24 13:27:43 2019-06-21 14:27:18 FALSE
5 113003885~ B1018    1018 580111 2019-05-24 13:36:46 2019-06-21 12:11:57 FALSE
6 113004596~ B1019    1019 580111 2019-05-24 13:44:41 2019-06-21 14:37:21 FALSE
7 113005051~ B1020    1020 580111 2019-05-24 13:49:21 2019-06-21 12:25:16 FALSE
8 113006028~ B1021    1021 580111 2019-05-24 13:59:57 2019-06-21 11:49:47 FALSE
9 113010219~ B1022    1022 580111 2019-05-24 14:43:53 2019-06-19 14:12:38 FALSE
10 113010688~ B1023    1023 580111 2019-05-24 14:48:16 2019-06-21 15:54:24 FALSE
# ... with 2,864 more rows, 51 more variables: current_subtotal_price <dbl>,
#   current_total_price <dbl>, current_total_discounts <dbl>,
#   current_total_duties_set <dbl>, total_discounts <dbl>,
#   total_line_items_price <dbl>, total_outstanding <dbl>, total_price <dbl>,
#   total_tax <dbl>, total_tip_received <dbl>, taxes_included <lgl>,
#   discount_codes <chr>, financial_status <chr>, fulfillment_status <chr>,
#   source_name <chr>, landing_site <chr>, landing_site_ref <chr>, ...
```

### 3.6 Spalten über eine Menge auswählen

```
# Liste von Spalten via all_of: Nur gültige Spaltennamen erlaubt
cols <- c("created_at", "processed_at", "closed_at")
```

```
orders %>%
  select(all_of(cols))
```

```
# A tibble: 2,874 x 3
  created_at processed_at closed_at
  <dtm>          <dtm>          <dtm>
1 2019-05-24 12:59:16 2019-05-24 12:59:15 2019-06-19 13:23:26
2 2019-05-24 13:09:08 2019-05-24 13:09:07 2019-06-21 14:40:07
3 2019-05-24 13:22:41 2019-05-24 13:22:40 2019-06-21 12:35:22
4 2019-05-24 13:27:43 2019-05-24 13:27:42 2019-06-21 14:27:18
5 2019-05-24 13:36:46 2019-05-24 13:36:45 2019-06-21 12:11:57
6 2019-05-24 13:44:41 2019-05-24 13:44:41 2019-06-21 14:37:21
7 2019-05-24 13:49:21 2019-05-24 13:49:20 2019-06-21 12:25:16
8 2019-05-24 13:59:57 2019-05-24 13:59:57 2019-06-21 11:49:46
```

```

9 2019-05-24 14:43:53 2019-05-24 14:43:53 2019-06-19 14:12:38
10 2019-05-24 14:48:16 2019-05-24 14:48:16 2019-06-21 15:54:24
# ... with 2,864 more rows

```

```

# Liste von Spalten via any_of: Nicht alle Spalten muss es auch geben.
# Sinnvoll, wenn gleiche Liste auf unterschiedliche Datensätze angewendet wird.
cols <- c("created_at", "processed_at", "closed_at", "shipped_at")

orders %>%
  select(any_of(cols))

```

```

# A tibble: 2,874 x 3
  created_at      processed_at      closed_at
  <dtm>          <dtm>          <dtm>
1 2019-05-24 12:59:16 2019-05-24 12:59:15 2019-06-19 13:23:26
2 2019-05-24 13:09:08 2019-05-24 13:09:07 2019-06-21 14:40:07
3 2019-05-24 13:22:41 2019-05-24 13:22:40 2019-06-21 12:35:22
4 2019-05-24 13:27:43 2019-05-24 13:27:42 2019-06-21 14:27:18
5 2019-05-24 13:36:46 2019-05-24 13:36:45 2019-06-21 12:11:57
6 2019-05-24 13:44:41 2019-05-24 13:44:41 2019-06-21 14:37:21
7 2019-05-24 13:49:21 2019-05-24 13:49:20 2019-06-21 12:25:16
8 2019-05-24 13:59:57 2019-05-24 13:59:57 2019-06-21 11:49:46
9 2019-05-24 14:43:53 2019-05-24 14:43:53 2019-06-19 14:12:38
10 2019-05-24 14:48:16 2019-05-24 14:48:16 2019-06-21 15:54:24
# ... with 2,864 more rows

```

## **4 Spalten hinzufügen und verändern**

## Quellen