R für Data Science

Daten importieren, bereinigen, umformen und visualisieren

KAPITEL 7

Daten importieren

Einführung

Mit Daten zu arbeiten, die von den R-Paketen bereitgestellt werden, ist eine großartige Möglichkeit, die Tools der Data Science kennenzulernen. Doch an einem gewissen Punkt möchten Sie das Gelernte auch auf Ihre eigenen Daten anwenden. Dieses Kapitel beschäftigt sich nun mit den Grundlagen, Datendateien in R einzulesen.

Speziell konzentriert es sich darauf, wie man einfache, rechteckige Textdateien einliest. Los geht es mit einem praktischen Hinweis für den Umgang mit Features wie Spaltennamen, Typen und fehlenden Daten. Dann erfahren Sie, wie man mehrere Dateien auf einmal einliest und Daten aus R in eine Datei schreibt. Schließlich lernen Sie, wie Sie Dataframes in R manuell zusammenbauen.

Voraussetzungen

In diesem Kapitel lernen Sie, lineare Dateien mit dem Paket `readr` in R zu laden. Dieses Paket ist Teil des Kern-Tidyverse:

```
library(tidyverse)
```

Daten aus einer Datei lesen

Zu Beginn konzentrieren wir uns auf den gebräuchlichsten Typ einer rechteckigen Datendatei: CSV, was als Abkürzung für *Comma-Separated Values* (kommagetrennte Werte) steht. Das folgende Beispiel zeigt, wie eine einfache CSV-Datei aussieht. Die erste Zeile, häufig auch *Header-Zeile* oder Überschriftenzeile genannt, gibt die Spaltennamen an, und in den folgenden sechs Zeilen sind die Daten enthalten. Die Spalten werden durch Kommata voneinander getrennt.

```
Student ID,Full Name,favourite.food,mealPlan,AGE
1,Sunil Huffmann,Strawberry yoghurt,Lunch only,4
2,Barclay Lynn,French fries,Lunch only,5
3,Jayendra Lyne,N/A,Breakfast and lunch,7
4,Leon Rossini,Anchovies,Lunch only,
5,Chidiegwu Dunkel,Pizza,Breakfast and lunch,five
6,Güvenç Attila,Ice cream,Lunch only,6
```

Tabelle 7-1 stellt dieselben Daten als Tabelle dar.

Tabelle 7-1: Daten aus der Datei students.csv in Form einer Tabelle

Student ID	Full Name	favourite.food	mealPlan	AGE
1	Sunil Huffmann	Strawberry yoghurt	Lunch only	4
2	Barclay Lynn	French fries	Lunch only	5
3	Jayendra Lyne	N/A	Breakfast and lunch	7
4	Leon Rossini	Anchovies	Lunch only	NA
5	Chidiegwu Dunkel	Pizza	Breakfast and lunch	five
6	Güvenç Attila	Ice cream	Lunch only	6

Diese Datei können wir mit `read_csv()` in R einlesen. Das erste Argument ist das wichtigste: der Pfad zur Datei. Man kann sich den Pfad als die Adresse der Datei vorstellen: Die Datei heißt `students.csv` und »wohnt« im Ordner `data`.

```
students <- read_csv("data/students.csv")
#> Rows: 6 Columns: 5
#> ── Column specification ────────────────────────────────────────────
#> Delimiter: ","
#> chr (4): Full Name, favourite.food, mealPlan, AGE
#> dbl (1): Student ID
#>
#> 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.
```

Der obige Code funktioniert, wenn Sie die Datei *students.csv* in einem Ordner *data* in Ihrem Projekt abgelegt haben. Die Datei *students.csv* können Sie von *https://oreil.ly/GDubb* herunterladen oder direkt von der angegebenen URL lesen:

```
students <- read_csv("https://pos.it/r4ds-students-csv")
```

Die Funktion `read_csv()` gibt eine Meldung zurück mit der Anzahl der Datenzeilen und -spalten, dem verwendeten Trennzeichen und den Spaltenspezifikationen (Namen der Spalten, organisiert nach dem Typ der in den Spalten enthaltenen Daten). Außerdem erfahren Sie, wie sich die vollständige Spaltenspezifikation abrufen lässt und wie Sie diese Meldung unterdrücken können. Die Meldung ist integraler Bestandteil von `readr`, und wir kommen im Abschnitt »Spaltentypen steuern« auf Seite 132 darauf zurück.

Praktischer Ratschlag

Nachdem Sie Daten eingelesen haben, besteht der erste Schritt in der Regel darin, sie in bestimmter Weise umzuwandeln, um sie für die weitere Analyse leichter verarbeiten zu können. Sehen wir uns in diesem Sinne noch einmal die `students`-Daten an:

```
students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan            AGE
#>          <dbl> <chr>            <chr>              <chr>               <chr>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2            2 Barclay Lynn     French fries       Lunch only          5
#> 3            3 Jayendra Lyne    N/A                Breakfast and lunch 7
#> 4            4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6            6 Güvenç Attila    Ice cream          Lunch only          6
```

In der Spalte `favourite.food` stehen verschiedene Lebensmittel sowie die Zeichenfolge »N/A«, die ein richtiges `NA` sein sollte, das R als *not available* (nicht verfügbar) erkennt. Das ist etwas, das wir mit dem Argument `na` angehen können. Standardmäßig erkennt die Funktion `read_csv()` nur leere Zeichenfolgen (`""`) in diesem Datenset als `NA`-Werte. Wir möchten aber, dass sie auch den String `"N/A"` erkennt:

```
students <- read_csv("data/students.csv", na = c("N/A", ""))

students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan            AGE
#>          <dbl> <chr>            <chr>              <chr>               <chr>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2            2 Barclay Lynn     French fries       Lunch only          5
#> 3            3 Jayendra Lyne    <NA>               Breakfast and lunch 7
#> 4            4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6            6 Güvenç Attila    Ice cream          Lunch only          6
```

Sicherlich haben Sie bemerkt, dass die Spalten `Student ID` und `Full Name` von Backticks umgeben sind. Das liegt daran, dass sie Leerzeichen enthalten und damit die üblichen Regeln von R für Variablennamen verletzen. Um auf diese Variablen zu verweisen, müssen Sie sie in Backticks (`` ` ``) einschließen:

```
students |>
  rename(
    student_id = `Student ID`,
    full_name = `Full Name`
  )
#> # A tibble: 6 × 5
#>   student_id full_name        favourite.food     mealPlan            AGE
#>        <dbl> <chr>            <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
```

```
#> 3          3 Jayendra Lyne    <NA>              Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies         Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza             Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream         Lunch only          6
```

Als alternativer Ansatz bietet sich die Funktion `janitor::clean_names()` an, um mithilfe einer Heuristik alle Namen auf einmal in Snake Case umzuwandeln:[1]

```
students |> janitor::clean_names()
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>        <dbl> <chr>            <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

Nach dem Einlesen der Daten ist es häufig ebenfalls erforderlich, die Variablentypen zu betrachten. Zum Beispiel ist `meal_plan` eine kategoriale Variable mit einem bekannten Satz möglicher Werte, die in R als Faktor dargestellt werden sollte.

```
students |>
  janitor::clean_names() |>
  mutate(meal_plan = factor(meal_plan))
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>        <dbl> <chr>            <chr>              <fct>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

Beachten Sie, dass die Werte in der Variablen `meal_plan` gleich geblieben sind, aber der Variablentyp, der unter dem Variablennamen angegeben ist, hat sich von Zeichen (`<chr>`) in Faktor (`<fct>`) geändert. Kapitel 16 geht näher auf Faktoren ein.

Bevor Sie diese Daten analysieren, werden Sie wahrscheinlich die Spalten `age` und `id` bereinigen wollen. Derzeit ist `age` eine Zeichenvariable, weil eine der Beobachtungen als Zahlwort `five` ausgeschrieben ist statt als Ziffer `5`. Wie sich derartige Probleme korrigieren lassen, besprechen wir ausführlich in Kapitel 20.

```
students <- students |>
  janitor::clean_names() |>
  mutate(
    meal_plan = factor(meal_plan),
    age = parse_number(if_else(age == "five", "5", age))
  )
```

1 Das Paket `janitor` (*https://oreil.ly/-J8GX*) ist nicht im Tidyverse enthalten, bietet aber praktische Funktionen für die Datenbereinigung und funktioniert auch gut in Daten-Pipelines, die `|>` verwenden.

```
students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <fct>                 <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

Neu ist hier die Funktion `if_else()`, die drei Argumente hat. Das erste Argument `test` sollte ein logischer Vektor sein. Das Ergebnis enthält den Wert des zweiten Arguments, `yes`, wenn `test` gleich `TRUE` ist, und den Wert des dritten Arguments, `no`, wenn der Test `FALSE` liefert. Hier sagen wir: Wenn `age` den String `"five"` enthält, mache `"5"` daraus, und wenn nicht, bleibt `age`, wie es war. Mehr über `if_else()` und logische Vektoren lernen Sie in Kapitel 12.

Andere Argumente

Es gibt noch eine Reihe weiterer wichtiger Argumente, die wir erwähnen müssen, und sie lassen sich besser vorführen, wenn wir uns zunächst einen praktischen Trick ansehen: Die Funktion `read_csv()` kann Textzeichenfolgen lesen, die Sie erzeugt und wie eine CSV-Datei formatiert haben:

```
read_csv(
  "a,b,c
  1,2,3
  4,5,6"
)
#> # A tibble: 2 × 3
#>       a     b     c
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

Normalerweise verwendet `read_csv()` die erste Zeile der Daten für die Spaltennamen, was eine gängige Konvention ist. Es ist auch nicht ungewöhnlich, dass am Anfang der Datei einige Zeilen mit Metadaten enthalten sind. Die ersten `n` Zeilen können Sie mit `skip = n` überspringen, oder Sie können mit `comment = "#"` alle Zeilen auslassen, die beispielsweise mit `#` beginnen:

```
read_csv(
  "The first line of metadata
  The second line of metadata
  x,y,z
  1,2,3", skip = 2
)
#> # A tibble: 1 × 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
```

```
read_csv(
  "# A comment I want to skip
  x,y,z
  1,2,3", comment = "#"
)
#> # A tibble: 1 × 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
```

Es kann auch sein, dass die Daten keine Spaltennamen haben. Mit `col_names = FALSE` weisen Sie `read_csv()` an, die erste Zeile nicht als Überschriftenzeile (Header) zu verarbeiten, sondern sie stattdessen sequenziell von `X1` bis `Xn` zu beschriften:

```
read_csv(
  "1,2,3
  4,5,6",
  col_names = FALSE
)
#> # A tibble: 2 × 3
#>      X1    X2    X3
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

Alternativ können Sie `col_names` einen Zeichenvektor übergeben, der für die Spaltennamen verwendet wird:

```
read_csv(
  "1,2,3
  4,5,6",
  col_names = c("x", "y", "z")
)
#> # A tibble: 2 × 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

Wenn Sie diese Argumente kennen, sind Sie in der Lage, die meisten in der Praxis vorkommenden CSV-Dateien einzulesen. (Für die übrigen Varianten müssen Sie Ihre *.csv*-Datei sorgfältig inspizieren und die Dokumentation für die vielen anderen Argumente von `read_csv()` studieren.)

Andere Dateitypen

Sobald Sie `read_csv()` beherrschen, ist es sehr einfach, die anderen Funktionen von `readr` zu verwenden. Sie müssen lediglich wissen, welche Funktion jeweils infrage kommt:

`read_csv2()`
: Liest Dateien ein, deren Felder durch Semikola (`;`) statt durch Kommata (`,`) getrennt sind. Derartige Dateien sind in Ländern üblich, in denen das Komma als Dezimaltrennzeichen dient.

`read_tsv()`
: Liest Dateien ein, deren Felder durch Tabulatoren getrennt sind.

`read_delim()`
: Liest Dateien mit einem beliebigen Trennzeichen ein, wobei versucht wird, das Trennzeichen automatisch zu erraten, wenn Sie es nicht angeben.

`read_fwf()`
: Liest Dateien mit Feldern fester Breite ein. Die Felder können Sie mit `fwf_widths()` durch ihre Breite oder mit `fwf_positions()` durch ihre Positionen spezifizieren.

`read_table()`
: Liest eine gebräuchliche Variation von Dateien mit fester Breite ein, wobei die Spalten durch Leerzeichen getrennt sind.

`read_log()`
: Liest Protokolldateien im Apache-Stil ein.

Übungen

1. Mit welcher Funktion würden Sie eine Datei einlesen, deren Felder durch `|` getrennt sind?
2. Welche anderen Argumente außer `file`, `skip` und `comment` haben die Funktionen `read_csv()` und `read_tsv()` gemein?
3. Was sind die wichtigsten Argumente der Funktion `read_fwf()`?
4. Manchmal enthalten CSV-Dateien Zeichenfolgen mit Kommata. Um Probleme zu vermeiden, müssen diese Kommata in Anführungszeichen eingeschlossen werden, und zwar in einfache (`'`) oder doppelte (`"`). Standardmäßig geht `read_csv()` davon aus, dass es sich um doppelte Anführungszeichen (`"`) handelt. Welches Argument müssen Sie bei `read_csv()` angeben, um den folgenden Text in einen Dataframe einzulesen?

   ```
   "x,y\n1,'a,b'"
   ```

5. Ermitteln Sie, was bei den folgenden Inline-CSV-Dateien jeweils nicht stimmt. Was passiert, wenn Sie den Code ausführen?

   ```
   read_csv("a,b\n1,2,3\n4,5,6")
   read_csv("a,b,c\n1,2\n1,2,3,4")
   read_csv("a,b\n\"1")
   read_csv("a,b\n1,2\na,b")
   read_csv("a;b\n1;3")
   ```

6. Üben Sie, sich auf nicht syntaktische Namen im folgenden Dataframe zu beziehen, indem Sie
   a. die Variable namens `1` extrahieren,
   b. ein Streudiagramm von `1` gegen `2` erstellen,
   c. eine neue Spalte namens `3` erzeugen, die `2` geteilt durch `1` ist,
   d. die Spalten in `one`, `two` und `three` umbenennen:

```
annoying <- tibble(
`1` = 1:10,
`2` = `1` * 2 + rnorm(length(`1`))
)
```

Spaltentypen steuern

Da eine CSV-Datei keine Informationen über den Typ jeder Variablen enthält (d.h., ob sie einen logischen Wert, eine Zahl, eine Zeichenfolge usw. darstellt), versucht `readr`, den Typ zu erraten. Dieser Abschnitt beschreibt, wie das Erraten funktioniert, wie man einige häufige Probleme löst, die zum Scheitern führen, und wie man bei Bedarf die Spaltentypen selbst bereitstellen kann. Zum Schluss stellen wir noch einige allgemeine Strategien vor, die nützlich sind, wenn `readr` katastrophal versagt und Sie mehr Einblick in die Struktur Ihrer Datei benötigen.

Typen erraten

Das Paket `readr` verwendet eine Heuristik, um die Spaltentypen herauszufinden. Für jede Spalte holt es die Werte von 1.000 Zeilen[2] in gleichmäßigen Abständen von der ersten bis zur letzten Zeile und ignoriert dabei fehlende Werte. Anschließend arbeitet es die folgenden Fragen ab:

- Enthält sie nur `F`, `T`, `FALSE` oder `TRUE` (ohne Beachtung der Groß-/Kleinschreibung)? Wenn ja, handelt es sich um einen logischen Wert.
- Enthält sie nur Zahlen (z. B. `1`, `-4.5`, `5e6`, `Inf`)? Wenn ja, handelt es sich um eine Zahl.
- Entspricht sie dem Standard ISO8601? Wenn ja, handelt es sich um ein Datum oder um ein Datum mit Uhrzeit. (Auf Datums-/Zeitwerte kommen wir ausführlich in Kapitel 17 im Abschnitt »Datums-/Zeitwerte erzeugen« auf Seite 326 zurück.)
- Andernfalls muss es sich um eine Zeichenfolge handeln.

Dieses Verhalten können Sie mit diesem einfachen Beispiel nachvollziehen:

```
read_csv("
  logical,numeric,date,string
```

2 Den Standardwert 1.000 können Sie mit dem Argument `guess_max` überschreiben.

```
  TRUE,1,2021-01-15,abc
  false,4.5,2021-02-15,def
  T,Inf,2021-02-16,ghi
")
#> # A tibble: 3 × 4
#>   logical numeric date       string
#>   <lgl>     <dbl> <date>     <chr>
#> 1 TRUE        1   2021-01-15 abc
#> 2 FALSE       4.5 2021-02-15 def
#> 3 TRUE      Inf   2021-02-16 ghi
```

Diese Heuristik funktioniert gut, wenn das Datenset sauber ist, doch in der Praxis werden Sie auf eine Reihe von seltsamen und ungewöhnlichen Fehlern stoßen.

Fehlende Werte, Spaltentypen und Probleme

Eine Spaltenerkennung scheitert vor allem dann, wenn eine Spalte unerwartete Werte enthält. Dann bekommen Sie eine Zeichenspalte anstelle eines spezifischeren Typs. Eine der häufigsten Ursachen dafür ist ein fehlender Wert, der mit etwas anderem als dem von `readr` erwarteten `NA` erfasst wurde.

Nehmen Sie diese einfache einspaltige CSV-Datei als Beispiel:

```
simple_csv <- "
  x
  10
  .
  20
  30"
```

Wenn wir die Datei ohne zusätzliche Argumente einlesen, wird `x` zu einer Zeichenspalte:

```
read_csv(simple_csv)
#> # A tibble: 4 × 1
#>   x
#>   <chr>
#> 1 10
#> 2 .
#> 3 20
#> 4 30
```

In diesem kleinen Datenset können Sie den fehlenden Wert – durch einen Punkt (`.`) dargestellt – leicht erkennen. Doch wie sieht es aus bei Tausenden von Zeilen mit nur wenigen fehlenden Werten, die durch Punkte dargestellt werden? Man könnte `readr` mitteilen, dass `x` eine numerische Spalte ist, und dann sehen, wo das Einlesen versagt. Hierfür weisen Sie dem Argument `col_types` eine benannte Liste zu, in der die Namen den Spaltennamen in der CSV-Datei entsprechen:

```
df <- read_csv(
  simple_csv,
  col_types = list(x = col_double())
)
```

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

Jetzt meldet `read_csv()`, dass es ein Problem gibt, und sagt uns, dass wir mit `problems()` mehr herausfinden können:

```
problems(df)
#> # A tibble: 1 × 5
#>     row   col expected actual file
#>   <int> <int> <chr>    <chr>  <chr>
#> 1     3     1 a double .      /private/tmp/RtmpAYlSop/file392d445cf269
```

Wir erfahren nun, dass es ein Problem in Zeile 3, Spalte 1 gibt, wo `readr` einen Wert vom Typ `double` erwartet, aber einen Punkt (`.`) vorgefunden hat. Das legt nahe, dass dieses Datenset fehlende Werte mit einem Punkt kennzeichnet. Also setzen wir `na = "."`. Die automatische Typherleitung ist nun erfolgreich und liefert uns die numerische Spalte, die wir haben wollten:

```
read_csv(simple_csv, na = ".")
#> # A tibble: 4 × 1
#>       x
#>   <dbl>
#> 1    10
#> 2    NA
#> 3    20
#> 4    30
```

Spaltentypen

Im Paket `readr` können Sie aus insgesamt neun Spaltentypen wählen:

- `col_logical()` und `col_double()` lesen logische Werte und Realzahlen. Man benötigt sie nur selten (außer wie oben gezeigt), da `readr` normalerweise solche Typen automatisch erkennt.
- `col_integer()` liest Ganzzahlen. In diesem Buch unterscheiden wir nur selten zwischen Ganzzahlen und Gleitkommazahlen (`double`), da sie funktional äquivalent sind. Allerdings kann das explizite Lesen von Ganzzahlen gelegentlich nützlich sein, da sie gegenüber Gleitkommazahlen nur die Hälfte des Speichers belegen.
- `col_character()` liest Zeichenfolgen. Dies kann nützlich sein, wenn zum Beispiel eine Spalte einen numerischen Bezeichner verkörpert, d.h. eine lange Folge von Ziffern, die ein Objekt identifizieren, aber in mathematischen Operationen nicht sinnvoll sind. Beispiele hierfür sind Telefonnummern, Sozialversicherungsnummern, Kreditkartennummern usw.
- `col_factor()`, `col_date()` und `col_datetime()` erzeugen Faktoren, Datumswerte und Datums-/Zeitwerte. Mehr dazu erfahren Sie, wenn wir in den Kapiteln 16 und 17 auf diese Datentypen zu sprechen kommen.

- `col_number()` ist ein toleranter numerischer Parser, der nicht numerische Komponenten ignoriert und besonders nützlich ist für Währungen. Mehr dazu lesen Sie in Kapitel 13.
- `col_skip()` überspringt eine Spalte, die auch nicht in das Ergebnis aufgenommen wird. Dies kann zum Beispiel Zeit sparen, wenn Sie sehr große CSV-Dateien einlesen müssen, aber nur einige der Spalten benötigen.

Es ist auch möglich, die Standardspalte zu überschreiben, indem Sie von `list()` zu `cols()` wechseln und `.default` angeben:

```
another_csv <- "
x,y,z
1,2,3"

read_csv(
  another_csv,
  col_types = cols(.default = col_character())
)
#> # A tibble: 1 × 3
#>   x     y     z
#>   <chr> <chr> <chr>
#> 1 1     2     3
```

Eine andere nützliche Hilfsfunktion ist `cols_only()`, die nur die angegebene(n) Spalte(n) einliest:

```
read_csv(
  another_csv,
  col_types = cols_only(x = col_character())
)
#> # A tibble: 1 × 1
#>   x
#>   <chr>
#> 1 1
```

Daten aus mehreren Dateien einlesen

Manchmal sind Ihre Daten auf mehrere Dateien verteilt, anstatt in einer einzigen Datei enthalten zu sein. Ein Beispiel hierfür sind monatliche Umsatzdaten, wobei die Daten für jeden Monat in einer eigenen Datei liegen: *01-sales.csv* für Januar, *02-sales.csv* für Februar und *03-sales.csv* für März. Mit `read_csv()` können Sie diese Daten auf einmal einlesen und sie in einem einzelnen Dataframe übereinanderstapeln.

```
sales_files <- c("data/01-sales.csv", "data/02-sales.csv", "data/03-sales.csv")
read_csv(sales_files, id = "file")
#> # A tibble: 19 × 6
#>   file              month    year brand  item     n
#>   <chr>             <chr>   <dbl> <dbl> <dbl> <dbl>
#> 1 data/01-sales.csv January  2019     1  1234     3
#> 2 data/01-sales.csv January  2019     1  8721     9
```

```
#> 3 data/01-sales.csv January  2019     1  1822     2
#> 4 data/01-sales.csv January  2019     2  3333     1
#> 5 data/01-sales.csv January  2019     2  2156     9
#> 6 data/01-sales.csv January  2019     2  3987     6
#> # … with 13 more rows
```

Auch hier funktioniert der obige Code nur dann, wenn Sie die CSV-Dateien in einem Ordner *data* in Ihrem Projekt abgelegt haben. Diese drei Dateien können Sie von *https://oreil.ly/jVd8o*, *https://oreil.ly/RYsgM* und *https://oreil.ly/4uZOm* herunterladen oder sie wie folgt direkt einlesen:

```
sales_files <- c(
  "https://pos.it/r4ds-01-sales",
  "https://pos.it/r4ds-02-sales",
  "https://pos.it/r4ds-03-sales"
)
read_csv(sales_files, id = "file")
```

Das Argument `id` fügt eine neue Spalte namens `file` in den resultierenden Dataframe ein. In ihr ist die Datei angegeben, aus der die Daten stammen. Das ist vor allem dann hilfreich, wenn die einzulesenden Dateien keine Identifizierungsspalte haben, die es erlauben würde, die Beobachtungen zu ihren ursprünglichen Quellen zurückzuverfolgen.

Wenn Sie viele Dateien einlesen wollen, kann es recht umständlich sein, ihre Namen als Liste zu schreiben. Stattdessen können Sie die Basisfunktion `list.files()` verwenden, die Ihnen die gewünschten Dateien anhand eines Musters in den Dateinamen zusammensucht. Mehr über derartige Muster lernen Sie in Kapitel 15.

```
sales_files <- list.files("data", pattern = "sales\\.csv$", full.names = TRUE)
sales_files
#> [1] "data/01-sales.csv" "data/02-sales.csv" "data/03-sales.csv"
```

In eine Datei schreiben

Das Paket `readr` bringt ebenfalls zwei nützliche Funktionen mit, um Daten auf einen Datenträger zu schreiben: `write_csv()` und `write_tsv()`. Die wichtigsten Argumente dieser Funktionen sind `x` (der zu speichernde Dataframe) und `file` (der Ort, an dem die Datei zu speichern ist). Außerdem können Sie mit `na` festlegen, wie fehlende Werte gespeichert werden sollen, und mit `append`, ob Sie den Dataframe an eine vorhandene Datei anfügen wollen.

```
write_csv(students, "students.csv")
```

Lesen wir nun diese CSV-Datei erneut ein. Beachten Sie, dass die Informationen über die Variablentypen, die Sie eben eingerichtet haben, verloren gehen, wenn Sie den Dataframe als CSV speichern. Somit stehen Sie wieder am Anfang und müssen eine reine Textdatei einlesen:

```
students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <fct>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
write_csv(students, "students-2.csv")
read_csv("students-2.csv")
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <chr>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

Deshalb sind CSV-Dateien nicht so recht geeignet, um Zwischenergebnisse zu speichern – Sie müssen die Spaltenspezifikation jedes Mal erneut durchführen, wenn Sie den Dataframe laden. Hierzu gibt es zwei Alternativen:

- `write_rds()` und `read_rds()` sind einheitliche Wrapper um die Basisfunktionen `readRDS()` und `saveRDS()`. Diese Funktionen speichern die Daten in dem R-eigenen Binärformat namens RDS. Wenn Sie also das Objekt zurückladen, dann laden Sie *genau das gleiche* R-Objekt zurück, das Sie gespeichert haben.

```
write_rds(students, "students.rds")
read_rds("students.rds")
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <fct>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

- Das Paket `arrow` erlaubt es, Dateien im Datendateiformat Parquet zu lesen und zu schreiben. Dieses schnelle, binäre Dateiformat lässt sich über Programmiersprachen hinweg einsetzen. Auf das Paket `arrow` kommen wir ausführlich in Kapitel 22 zurück.

```
library(arrow)
write_parquet(students, "students.parquet")
read_parquet("students.parquet")
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <fct>               <dbl>
```

```
#> 1         1 Sunil Huffmann   Strawberry yoghurt Lunch only            4
#> 2         2 Barclay Lynn     French fries       Lunch only            5
#> 3         3 Jayendra Lyne    NA                 Breakfast and lunch   7
#> 4         4 Leon Rossini     Anchovies          Lunch only           NA
#> 5         5 Chidiegwu Dunkel Pizza              Breakfast and lunch   5
#> 6         6 Güvenç Attila    Ice cream          Lunch only            6
```

Parquet ist in der Regel viel schneller als RDS und lässt sich auch außerhalb von R einsetzen. Allerdings erfordert es das Paket `arrow`.

Dateneingabe

Manchmal müssen Sie ein Tibble »manuell« zusammenstellen, indem Sie ein wenig Dateieingabe in Ihrem R-Skript praktizieren. Hierbei helfen Ihnen zwei nützliche Funktionen, die sich darin unterscheiden, ob das Tibble spalten- oder zeilenorientiert ist. Die Funktion `tibble()` arbeitet spaltenorientiert:

```
tibble(
x = c(1, 2, 5),
y = c("h", "m", "g"),
z = c(0.08, 0.83, 0.60)
)
#> # A tibble: 3 × 3
#>       x y         z
#>   <dbl> <chr> <dbl>
#> 1     1 h      0.08
#> 2     2 m      0.83
#> 3     5 g      0.6
```

Wenn die Daten spaltenorientiert angeordnet sind, lässt sich schwerer erkennen, welche Beziehungen zwischen den Zeilen bestehen. Eine Alternative ist also `tribble()`, kurz für *transposed tibble*, mit dem Sie Ihre Daten zeilenweise anordnen können. Die Funktion `tribble()` ist auf die Dateneingabe im Code angepasst: Spaltenüberschriften beginnen mit einer Tilde (~), und die Einträge werden durch Kommata getrennt. Dadurch ist es möglich, kleine Datenmengen in einer gut lesbaren Form anzuordnen:

```
tribble(
 ~x, ~y, ~z,
 1, "h", 0.08,
 2, "m", 0.83,
 5, "g", 0.60
)
#> # A tibble: 3 × 3
#>   x         y     z
#>   <chr> <dbl> <dbl>
#> 1         1 h  0.08
#> 2         2 m  0.83
#> 3         5 g  0.6
```

Zusammenfassung

In diesem Kapitel haben Sie gelernt, wie Sie CSV-Dateien mit `read_csv()` laden und Ihre eigene Dateneingabe mit `tibble()` und `tribble()` realisieren. Es wurde gezeigt, wie CSV-Dateien funktionieren, auf welche Probleme Sie möglicherweise stoßen und wie Sie diese lösen können. In diesem Buch werden wir noch mehrmals auf den Datenimport zurückkommen: In Kapitel 20 laden Sie Daten aus Excel und Google Sheets, in Kapitel 21 aus Datenbanken, in Kapitel 22 aus Parquet-Dateien, in Kapitel 23 von JSON und in Kapitel 24 von Websites.

Wir sind fast am Ende dieses Abschnitts des Buchs angelangt, aber es gibt noch ein wichtiges letztes Thema zu behandeln: wie man Hilfe bekommt. Im nächsten Kapitel erfahren Sie, wo Sie am besten nach Hilfe suchen können und wie Sie ein Reprex erstellen, um die Chancen auf gute Hilfe zu maximieren, und Sie erhalten einige allgemeine Ratschläge dazu, wie Sie in der Welt von R auf dem Laufenden bleiben können.