KAPITEL 8

# Datenimport mit readr

## 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 nicht mehr nur lernen, sondern mit eigenen Daten arbeiten. In diesem Kapitel lernen Sie, wie Sie »rechteckige« Textdateien in R einlesen. Wir kratzen hier nur an der Oberfläche des Datenimports, doch viele der beschriebenen Prinzipien lassen sich auch auf andere Datenformate übertragen. Am Ende des Kapitels finden Sie Hinweise auf Pakete, die für andere Datentypen nützlich sind.

### Voraussetzungen

In diesem Kapitel lernen Sie, wie Sie lineare Dateien in R laden, und zwar mit dem Paket **readr**, das zum Kern des Tidyverse gehört.

```
library(tidyverse)
```

## Erste Schritte

Die meisten Funktionen von **readr** haben damit zu tun, lineare Dateien in Dataframes zu überführen:

- `read_csv()` liest kommagetrennte Dateien, `read_csv2()` liest semikolongetrennte Dateien (gebräuchlich in Ländern, in denen das Komma als Dezimaltrennzeichen dient), `read_tsv()` liest tabulatorgetrennte Dateien, und `read_delim()` liest Dateien mit einem beliebigen Trennzeichen.
- `read_fwf()` liest Dateien mit Feldern fester Breite. Felder können Sie entweder durch ihre Breite mit `fwf_widths()` oder ihre Position mit `fwf_positions()` spezifizieren. Die Funktion `read_table()` liest eine gebräuchliche Variante von Dateien mit Feldern fester Breite, in denen die Spalten durch Whitespaces getrennt sind.

- `read_log()` liest Apache-Protokolldateien. (Sehen Sie sich auch das Paket **webreadr** an (*https://github.com/Ironholds/webreadr*), das auf `read_log()` aufbaut und viele weitere hilfreiche Tools bietet.)

Diese Funktionen haben alle eine ähnliche Syntax: Kennen Sie eine, kennen Sie alle. Für den Rest dieses Kapitels konzentrieren wir uns auf `read_csv()`. Zum einen gehört das CSV-Format zu den gebräuchlichsten Formaten der Datenspeicherung, zum anderen können Sie Ihr Wissen auf alle anderen Funktionen in **readr** übertragen, wenn Sie einmal `read_csv()` verinnerlicht haben.

Das erste Argument an `read_csv()` ist das wichtigste – der Pfad zur einzulesenden Datei:

```
heights <- read_csv("data/heights.csv")
#> Parsed with column specification:
#> cols(
#>   earn = col_double(),
#>   height = col_double(),
#>   sex = col_character(),
#>   ed = col_integer(),
#>   age = col_integer(),
#>   race = col_character()
#> )
```

Wenn Sie `read_csv()` aufrufen, gibt die Funktion eine Spaltenspezifikation an, aus der Sie den Namen und den Typ jeder Spalte ablesen können. Dies ist ein wichtiger Bestandteil von **readr**, auf den wir im Abschnitt »Eine Datei parsen« auf Seite 130 zurückkommen.

Sie können auch eine CSV-Datei inline bereitstellen. Das ist zweckmäßig, wenn Sie mit **readr** experimentieren oder reproduzierbare Beispiele erstellen:

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

In beiden Fällen entnimmt `read_csv()` der ersten Datenzeile die Spaltennamen, was eine gebräuchliche Konvention ist. Es gibt zwei Fälle, in denen Sie dieses Verhalten anpassen werden:

- Manchmal stehen am Anfang einer Datei einige Zeilen Metadaten. Mit `skip = n` lassen sich die ersten `n` Zeilen überspringen; oder Sie verwenden `comment = "#"`, um alle Zeilen zu übergehen, die (zum Beispiel) 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
#>   <int> <int> <int>
#> 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
#>   <int> <int> <int>
#> 1     1     2     3
```

- Die Daten enthalten nicht immer Spaltennamen. Mit `col_names = FALSE` können Sie `read_csv()` anweisen, die Spaltenüberschriften nicht aus der ersten Zeile abzuleiten, sondern fortlaufend mit `X1` bis `Xn` zu bezeichnen:

```
read_csv("1,2,3\n4,5,6", col_names = FALSE)
#> # A tibble: 2 × 3
#>      X1    X2    X3
#>   <int> <int> <int>
#> 1     1     2     3
#> 2     4     5     6
```

(Die Zeichenfolge `"\n"` ist eine komfortable Kurzschreibweise, um eine neue Zeile zu erzeugen. Mehr dazu und zu anderen sogenannten Escapezeichen finden Sie in Kapitel 11 im Abschnitt »Grundlagen von Strings« auf Seite 181.)

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

```
read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
#> # A tibble: 2 × 3
#>       x     y     z
#>   <int> <int> <int>
#> 1     1     2     3
#> 2     4     5     6
```

Eine andere Option, die häufig eine Anpassung verlangt, ist `na`. Damit spezifizieren Sie einen oder mehrere Werte, die zur Darstellung für fehlende Werte in Ihrer Datei dienen:

```
read_csv("a,b,c\n1,2,.", na = ".")
#> # A tibble: 1 × 3
#>       a     b     c
#>   <int> <int> <chr>
#> 1     1     2  <NA>
```

Mehr brauchen Sie gar nicht zu wissen, um bereits etwa 75% der in der Praxis gebräuchlichen CSV-Dateien lesen zu können. Das Gelernte können Sie leicht übertragen, um tabulatorgetrennte Dateien mit `read_tsv()` und Dateien mit Feldern fester Breite mit `read_fwf()` zu lesen. Wenn Sie komplexere Dateien einlesen wollen, müssen Sie mehr darüber lernen, wie **readr** die einzelnen Spalten parst und sie in R-Vektoren überführt.

## Vergleich zu Basis-R

Wenn Sie bereits mit R gearbeitet haben, fragen Sie sich vielleicht, warum wir nicht `read.csv()` verwenden. Es gibt einige gute Gründe, um die **readr**-Funktionen gegenüber den äquivalenten Basisversionen zu bevorzugen:

- In der Regel sind sie wesentlich schneller (etwa zehnmal) als ihre Basisentsprechungen. Langlaufende Aufträge zeigen eine Fortschrittsleiste zur Orientierung an. Wenn Sie auf reine Geschwindigkeit aus sind, können Sie `data.table::fread()` probieren. Die Funktion passt zwar nicht so gut in das Tidyverse, kann aber noch ein ganzes Stück schneller sein.
- Sie produzieren Tibbles, konvertieren Zeichenvektoren nicht in Faktoren, verwenden Zeilennamen oder verschleiern die Spaltennamen. Hier liegen häufige Quellen für Ärgernisse bei den R-Basisfunktionen.
- Sie sind besser reproduzierbar. Da die Basisfunktionen von R bestimmte Verhaltensweisen vom Betriebssystem und den Umgebungsvariablen erben, läuft der Code vielleicht nur bei Ihnen und nicht auf einem anderen Computer.

## Übungen

1. Mit welcher Funktion lesen Sie eine Datei, deren Felder durch »|« getrennt sind?
2. Welche anderen Argumente außer `file`, `skip` und `comment` haben die Funktionen `read_csv()` und `read_tsv()` gemeinsam?
3. Welches sind die wichtigsten Argumente an `read_fwf()`?
4. Manchmal enthalten Strings in einer CSV-Datei Kommas. Damit die Kommas keine Probleme hervorrufen, müssen die Strings in einfache (') oder doppelte (") Anführungszeichen eingeschlossen werden. Per Konvention nimmt `read_csv()` ein doppeltes Anführungszeichen (") an. Wenn Sie dies ändern möchten, müssen Sie stattdessen `read_delim()` verwenden. Welche Argumente müssen Sie angeben, um den folgenden Text in einen Dataframe zu lesen?

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

5. Ermitteln Sie, was in jeder der folgenden Inline-CSV-Dateien falsch ist. 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")
   ```

# Einen Vektor parsen

Bevor wir zu den Details kommen, wie **readr** Dateien von der Festplatte liest, müssen wir einen kleinen Abstecher machen und über die `parse_*()`-Funktio-

nen sprechen. Diese Funktionen übernehmen einen Zeichenvektor und geben einen spezialisierten Vektor zurück, beispielsweise einen logischen, ganzzahligen oder Datumswert:

```
str(parse_logical(c("TRUE", "FALSE", "NA")))
#>  logi [1:3] TRUE FALSE NA
str(parse_integer(c("1", "2", "3")))
#>  int [1:3] 1 2 3
str(parse_date(c("2010-01-01", "1979-10-14")))
#>  Date[1:2], format: "2010-01-01" "1979-10-14"
```

Diese Funktionen sind an sich schon nützlich, aber sie sind es auch als wichtige Bausteine für **readr**. Nachdem Sie in diesem Abschnitt gelernt haben, wie die individuellen Parser arbeiten, kommen wir wieder zurück und sehen uns im nächsten Abschnitt an, wie sie zusammenspielen, um eine ganze Datei zu parsen.

Wie alle Funktionen im Tidyverse sind auch die `parse_*()`-Funktionen einheitlich konzipiert: Das erste Argument ist ein zu parsender Vektor, und das Argument `na` spezifiziert, welche Strings als fehlend gelten sollen:

```
parse_integer(c("1", "231", ".", "456"), na = ".")
#> [1]   1 231  NA 456
```

Wenn das Parsing scheitert, erhalten Sie eine Warnung:

```
x <- parse_integer(c("123", "345", "abc", "123.45"))
#> Warning: 2 parsing failures.
#> row col               expected actual
#>   3  -- an integer               abc
#>   4  -- no trailing characters    .45
```

Und die Fehler erscheinen nicht in der Ausgabe:

```
x
#> [1] 123 345  NA  NA
#> attr(,"problems")
#> # A tibble: 2 × 4
#>     row   col               expected actual
#>   <int> <int>                  <chr>  <chr>
#> 1     3    NA             an integer    abc
#> 2     4    NA no trailing characters    .45
```

Wenn es viele Parsing-Fehler gibt, müssen Sie mit `problems()` das vollständige Set abrufen. Diese Funktion gibt ein Tibble zurück, das Sie dann mit **dplyr** manipulieren können:

```
problems(x)
#> # A tibble: 2 × 4
#>     row   col               expected actual
#>   <int> <int>                  <chr>  <chr>
#> 1     3    NA             an integer    abc
#> 2     4    NA no trailing characters    .45
```

Um Parser zu verwenden, muss man vor allem wissen, was verfügbar ist und wie die Parser mit verschiedenen Eingabetypen umgehen. Besonders wichtig sind die folgenden Parser:

- `parse_logical()` und `parse_integer()` parsen logische Werte und Ganzzahlen. Da bei diesen Parsern prinzipiell nichts schiefgehen kann, gehe ich hier nicht weiter auf sie ein.
- `parse_double()` ist ein strenger numerischer Parser und `parse_number()` ein flexibler numerischer Parser. Diese sind komplizierter, als Sie denken, weil in verschiedenen Teilen der Welt Zahlen auf verschiedene Art und Weise geschrieben werden.
- `parse_character()` scheint so einfach zu sein, dass die Funktion eigentlich nicht notwendig ist. Doch durch eine Komplikation wird sie ziemlich wichtig: Zeichencodierungen.
- `parse_factor()` erzeugt Faktoren. R stellt mit einer solchen Datenstruktur kategoriale Variablen mit festen und bekannten Werten dar.
- `parse_datetime()`, `parse_date()` und `parse_time()` ermöglichen es, verschiedene Datums-/Uhrzeit-Spezifikationen zu parsen. Diese Parser sind am komplexesten, weil es so viele verschiedene Formate gibt, um Datum und Uhrzeit darzustellen.

Die folgenden Abschnitte beschreiben diese Parser ausführlicher.

## Zahlen

Anscheinend sollte es ganz einfach sein, eine Zahl zu parsen, doch drei Probleme machen dies zu einer verzwickten Aufgabe:

- Die Menschen schreiben in verschiedenen Teilen der Welt Zahlen auf unterschiedliche Art und Weise. So ist in manchen Ländern der Punkt zwischen dem ganzzahligen und dem gebrochenen Anteil einer Realzahl üblich, in anderen Ländern ein Komma.
- Zahlen sind oftmals von anderen Zeichen umgeben, die einen gewissen Kontext liefern, wie zum Beispiel »$1000« oder »10%«.
- Oftmals enthalten Zahlen »Gruppierungszeichen«, damit sie sich leichter lesen lassen, beispielsweise »1,000,000«, und auch diese Gruppierungszeichen sind in verschiedenen Teilen der Welt unterschiedlich.

Um dem ersten Problem entgegenzuwirken, hat **readr** das Konzept eines »Gebietsschemas« (`locale`). Ein solches Objekt spezifiziert Parsing-Optionen, die sich von Ort zu Ort unterscheiden. Beim Parsen von Zahlen ist die wichtigste Option das Zeichen, das Sie als Dezimaltrennzeichen verwenden. Den Standardwert (einen Punkt) können Sie überschreiben, indem Sie ein neues `locale`-Objekt erstellen und das Argument `decimal_mark` festlegen:

```
parse_double("1.23")
#> [1] 1.23
parse_double("1,23", locale = locale(decimal_mark = ","))
#> [1] 1.23
```

Das Standardgebietsschema von **readr** ist US-orientiert, weil R im Allgemeinen US-orientiert ist (das heißt, die Dokumentation von Basis-R ist in amerikanischem Englisch geschrieben). Alternativ könnte man versuchen, die Standardeinstellungen vom Betriebssystem abzufragen. Das ist ebenfalls nicht ganz einfach, vor allem aber wird der Code dadurch fragil: Selbst wenn er auf Ihrem Computer funktioniert, scheitert er möglicherweise, wenn Sie ihn per E-Mail an einen Kollegen in einem anderen Land schicken.

Die Funktion `parse_number()` geht das zweite Problem an: Sie ignoriert nicht numerische Zeichen vor und nach der Zahl. Das ist gerade bei Währungs- und Prozentwerten nützlich, funktioniert aber auch, um Zahlen aus Textpassagen zu extrahieren:

```
parse_number("$100")
#> [1] 100
parse_number("20%")
#> [1] 20
parse_number("It cost $123.45")
#> [1] 123
```

Das letzte Problem lässt sich mit der Kombination von `parse_number()` und Gebietsschema in den Griff bekommen, da `parse_number()` das »Gruppentrennzeichen« ignoriert:

```
# In Amerika verwendet
parse_number("$123,456,789")
#> [1] 1.23e+08

# In vielen Teilen Europas verwendet
parse_number(
  "123.456.789",
  locale = locale(grouping_mark = ".")
)
#> [1] 1.23e+08

# In der Schweiz verwendet
parse_number(
  "123'456'789",
  locale = locale(grouping_mark = "'")
)
#> [1] 1.23e+08
```

## Strings

Es scheint, dass `parse_character()` wirklich simpel sein sollte – die Funktion bräuchte lediglich ihre Eingabe zurückzugeben. Leider ist das Leben nicht so einfach, da es mehrere Methoden gibt, um den gleichen String darzustellen. Um die

Hintergründe zu verstehen, müssen wir uns detailliert damit beschäftigen, wie die Stringdarstellung des Computers aussieht. In R können wir mit der Funktion `charToRaw()` zur zugrunde liegenden Darstellung vorstoßen:

```
charToRaw("Hadley")
#> [1] 48 61 64 6c 65 79
```

Jede Hexadezimalzahl steht für ein Informationsbyte: `48` ist H, `61` ist a und so weiter. Die Zuordnung zwischen Hexadezimalzahl und Zeichen ist die sogenannte Codierung, und in diesem Fall heißt sie ASCII, was für *American Standard Code for Information Interchange* (Amerikanischer Standardcode für den Informationsaustausch) steht. Aus dem Namen lässt sich schon ableiten, dass ASCII für die Darstellung der Zeichen in der englischen Sprache gut geeignet ist.

Komplizierter wird es für andere Sprachen als Englisch. In der Anfangszeit der Computertechnik gab es viele konkurrierende Standards für die Codierung von nicht englischen Zeichen. Wollte man einen String korrekt interpretieren, musste man sowohl die Werte als auch die Codierung wissen. Zwei gebräuchliche Codierungen waren zum Beispiel Latin1 (aka ISO-8859-1, für westeuropäische Sprachen verwendet) und Latin2 (aka ISO-8859-2, für osteuropäische Sprachen verwendet). In Latin1 ist dem Hexadezimalwert b1 das Zeichen »±« zugeordnet, in Latin2 aber » ą«! Erfreulicherweise gibt es heute einen Standard, der fast überall unterstützt wird: UTF-8. UTF-8 kann praktisch alle Zeichen codieren, die heute von Menschen verwendet werden, und darüber hinaus viele zusätzliche Symbole (wie zum Beispiel Emojis!).

Das Paket **readr** verwendet UTF-8 überall: Es geht davon aus, dass Ihre Daten UTF-8-codiert sind, wenn Sie sie lesen, und verwendet die UTF-8-Codierung immer beim Schreiben. Diese Standardeinstellung ist zwar gut, scheitert aber bei Daten, die von älteren Systemen stammen, die UTF-8 nicht verstehen. Sollte Ihnen das passieren, sehen Ihre Strings höchst merkwürdig aus, wenn Sie sie ausgeben. Manchmal sind vielleicht nur ein oder zwei Zeichen verfälscht, ein andermal erhalten Sie vollkommenen Kauderwelsch. Zum Beispiel:

```
x1 <- "El Ni\xf1o was particularly bad this year"
x2 <- "\x82\xb1\x82\xf1\x82\xc9\x82\xbf\x82\xcd"
```

Um das Problem zu beseitigen, müssen Sie die Codierung in `parse_character()` spezifizieren:

```
parse_character(x1, locale = locale(encoding = "Latin1"))
#> [1] "El Niño was particularly bad this year"
parse_character(x2, locale = locale(encoding = "Shift-JIS"))
#> [1] "こんにちは"
```

Wie finden Sie die richtige Codierung heraus? Wenn Sie Glück haben, ist sie irgendwo in der Datendokumentation verzeichnet. Da das leider nur selten der Fall ist, stellt **readr** die Funktion `guess_encoding()` bereit, die Ihnen beim Ermitteln der Codierung hilft. Die Funktion ist nicht narrensicher und arbeitet besser,

wenn Sie Unmengen von Text haben (nicht wie hier), doch sie ist immerhin ein sinnvoller Ausgangpunkt. Möglicherweise müssen Sie mehrere verschiedene Codierungen ausprobieren, bis Sie die richtige finden:

```
guess_encoding(charToRaw(x1))
#>     encoding confidence
#> 1 ISO-8859-1       0.46
#> 2 ISO-8859-9       0.23
guess_encoding(charToRaw(x2))
#>   encoding confidence
#> 1   KOI8-R       0.42
```

Das erste Argument an `guess_encoding()` kann entweder ein Pfad zu einer Datei oder – wie in diesem Fall – ein roher Vektor sein (nützlich, wenn die Strings bereits in R existieren).

Codierungen sind ein umfangreiches und komplexes Thema, und ich habe hier nur an der Oberfläche gekratzt. Wenn Sie mehr lernen möchten, sollten Sie die ausführliche Erläuterung unter *http://kunststube.net/encoding/* lesen.

## Faktoren

R stellt kategoriale Variablen, die einen bekannten Satz von möglichen Werten haben, durch Faktoren dar. Übergeben Sie an `parse_factor()` einen Vektor von bekannten Stufen (`levels`), um eine Warnung zu generieren, wenn ein unerwarteter Wert erscheint:

```
fruit <- c("apple", "banana")
parse_factor(c("apple", "banana", "bananana"), levels = fruit)
#> Warning: 1 parsing failure.
#> row col           expected   actual
#>   3  -- value in level set bananana
#> [1] apple  banana <NA>
#> attr(,"problems")
#> # A tibble: 1 × 4
#>     row   col           expected   actual
#>   <int> <int>              <chr>    <chr>
#> 1     3    NA value in level set bananana
#> Levels: apple banana
```

Wenn Sie aber viele problematische Einträge haben, ist es oftmals einfacher, sie als Zeichenvektoren zu belassen und dann die Tools zu nutzen, die Sie in Kapitel 11 und Kapitel 12 kennenlernen, um die Datei zu bereinigen.

## Datum, Datums-/Zeitwerte und Zeiten

Bei Daten mit Datums- und/oder Zeitwerten können Sie unter drei Parsern wählen, je nachdem, ob es sich um ein Datum (die Anzahl der Tage seit 1970-01-01), einen Datums-/Zeitwert (die Anzahl der Sekunden seit Mitternacht 1970-01-01) oder eine Uhrzeit (die Anzahl der Sekunden seit Mitternacht) handelt. Wenn Sie

die Funktionen ohne zusätzliche Argumente aufrufen, gelten folgende Beschreibungen:

- `parse_datetime()` erwartet einen Datums-/Zeitwert nach ISO8601. Der internationale Standard ISO8601 beschreibt ein Format, bei dem die Elemente eines Datums vom größten zum kleinsten – Jahr, Monat, Tag, Stunde, Minute, Sekunde – angeordnet sind:

  ```
  parse_datetime("2010-10-01T2010")
  #> [1] "2010-10-01 20:10:00 UTC"

  # Wenn time fehlt, wird die Zeit auf Mitternacht gesetzt
  parse_datetime("20101010")
  #> [1] "2010-10-10 UTC"
  ```

  Dies ist der wichtigste Standard für Datums-/Zeitwerte, und wenn Sie häufig mit derartigen Werten arbeiten, sollten Sie den Artikel unter *https://de.wikipedia.org/wiki/ISO_8601* lesen.

- `parse_date()` erwartet das vierstellige Jahr, ein Trennzeichen, den Monat, ein Trennzeichen und dann den Tag (Trennzeichen ist - oder /):

  ```
  parse_date("2010-10-01")
  #> [1] "2010-10-01"
  ```

- `parse_time()` erwartet die Stunde, einen Doppelpunkt, die Minuten und optional einen Doppelpunkt und die Sekunden sowie ebenfalls optional einen AM-/PM-Bezeichner:

  ```
  library(hms)
  parse_time("01:10 am")
  #> 01:10:00
  parse_time("20:10:01")
  #> 20:10:01
  ```

  Da Basis-R keine großartige integrierte Klasse für Zeitdaten besitzt, verwenden wir diejenige aus dem Paket **hms**.

Wenn diese Standardeinstellungen für Ihre Datumswerte nicht funktionieren, können Sie Ihr eigenes Datums-/Zeitformat mit folgenden Elementen kreieren:

**Jahr**

`%Y` (4 Ziffern)

`%y` (2 Ziffern; 00–69 → 2000–2069, 70–99 → 1970–1999)

**Monat**

`%m` (2 Ziffern)

`%b` (abgekürzter Name, zum Beispiel »Jan«)

`%B` (vollständiger Name, »Januar«)

**Tag**

`%d` (2 Ziffern).

`%e` (optional führendes Leerzeichen)

**Zeit**

`%H` (24-Stunden-Format, 0–23)

`%I` (0–12, muss mit `%p` verwendet werden)

`%p` (Anzeige für AM/PM)

`%M` (Minuten)

`%S` (Sekunden als Ganzzahl)

`%OS` (Sekunden als Gleitkommazahl)

`%Z` (Zeitzone – ein Name, zum Beispiel `America/Chicago`). Hinweis: Vorsicht vor Abkürzungen. Für einen Amerikaner ist »EST« eine kanadische Zeitzone, die keine Sommerzeit kennt. Das ist die Eastern Standard Time! Im Abschnitt »Zeitzonen« auf Seite 235 kommen wir darauf zurück.

`%z` (als Zeitabstand von UTC, zum Beispiel `+0800`).

**Nichtziffern**

`%.` (überspringt ein Nichtziffernzeichen)

`%*` (überspringt beliebig viele Nichtziffernzeichen)

Das richtige Format lässt sich am besten herausfinden, wenn man einige Beispiele in einem Zeichenvektor erstellt und mit einer der Parsing-Funktionen testet. Zum Beispiel:

```
parse_date("01/02/15", "%m/%d/%y")
#> [1] "2015-01-02"
parse_date("01/02/15", "%d/%m/%y")
#> [1] "2015-02-01"
parse_date("01/02/15", "%y/%m/%d")
#> [1] "2001-02-15"
```

Wenn Sie die Formatbezeichner `%b` oder `%B` mit nicht englischen Monatsnamen verwenden, müssen Sie das Argument `lang` für `locale()` festlegen. Konsultieren Sie hierfür die Liste der integrierten Sprachen in `date_names_langs()`. Sollte Ihre Sprache noch nicht eingebunden sein, erstellen Sie einen eigenen Bezeichner mit `date_names()`:

```
parse_date("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
#> [1] "2015-01-01"
```

## Übungen

1. Welches sind die wichtigsten Argumente an `locale()`?
2. Was passiert, wenn Sie `decimal_mark` und `grouping_mark` auf dasselbe Zeichen setzen? Was geschieht mit dem Standardwert von `grouping_mark`, wenn Sie `decimal_mark` auf `","` setzen? Was passiert mit dem Standardwert von `decimal_mark`, wenn Sie `grouping_mark` auf `"."` setzen?

3. Auf die Optionen `date_format` und `time_format` für `locale()` bin ich nicht eingegangen. Was bewirken sie? Konstruieren Sie ein Beispiel, das zeigt, wann sie sinnvoll sein können.
4. Erstellen Sie für Ihr Land ein neues `locale`-Objekt, das die Einstellungen für die Dateitypen kapselt, die Sie am häufigsten lesen.
5. Wie unterscheiden sich `read_csv()` und `read_csv2()`?
6. Welche Codierungen sind in Europa am gebräuchlichsten? Welche Codierungen werden in Asien vor allem verwendet? Finden Sie es über eine Suche in einer Suchmaschine heraus.
7. Erzeugen Sie den korrekten Formatstring, um die folgenden Werte für Datum und Uhrzeit zu parsen:

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # Dec 30, 2014
t1 <- "1705"
t2 <- "11:15:10.12 PM"
```

# Eine Datei parsen

Nachdem Sie nun wissen, wie Sie einen einzelnen Vektor parsen, kommen wir zum Anfang zurück und untersuchen, wie **readr** eine Datei parst. In diesem Abschnitt lernen Sie, wie

- **readr** automatisch den Typ jeder Spalte herleitet,
- die Standardspezifikation überschrieben wird.

## Strategie

Das Paket **readr** ermittelt anhand einer Heuristik den Typ jeder Spalte: Es liest die ersten tausend Zeilen und verwendet eine (mäßig konservative) Heuristik, um den Typ jeder Spalte herzuleiten. Diesen Vorgang können Sie nachbilden mit einem Zeichenvektor mithilfe der Funktion `guess_parser()`, die die beste Annahme von **readr** zurückgibt, und der Funktion `parse_guess()`, die entsprechend dieser Annahme die Spalte parst:

```
guess_parser("2010-10-01")
#> [1] "date"
guess_parser("15:01")
#> [1] "time"
guess_parser(c("TRUE", "FALSE"))
#> [1] "logical"
guess_parser(c("1", "5", "9"))
```

```
#> [1] "integer"
guess_parser(c("12,352,561"))
#> [1] "number"

str(parse_guess("2010-10-10"))
#> Date[1:1], format: "2010-10-10"
```

Die Heuristik probiert die folgenden Typen aus und stoppt bei einer gefundenen Übereinstimmung:

`logical`
: Enthält nur »F«, »T«, »FALSE« oder »TRUE«.

`integer`
: Enthält nur numerische Zeichen (und -).

`double`
: Enthält nur gültige `double`-Werte (inklusive Zahlen wie `4.5e-5`).

`number`
: Enthält gültige `double`-Werte mit Gruppierungstrennzeichen.

`time`
: Entspricht dem standardmäßigen `time_format`.

`date`
: Entspricht dem standardmäßigen `date_format`.

`datetime`
: Jedes Datum nach ISO8601.

Wenn keine dieser Regeln greift, bleibt die Spalte als Vektor von Strings erhalten.

## Probleme

Diese Standardeinstellungen funktionieren bei größeren Dateien nicht immer. Prinzipiell gibt es zwei Probleme:

- Die ersten tausend Zeilen könnten ein Sonderfall sein, und **readr** errät einen Typ, der nicht genügend allgemein ist. Zum Beispiel könnte eine `double`-Spalte in den ersten tausend Zeilen nur Ganzzahlen enthalten.
- Die Spalte könnte viele fehlende Werte enthalten. Wenn in den ersten tausend Zeilen nur `NA`s stehen, nimmt **readr** an, dass es sich um einen Zeichenvektor handelt, obwohl Sie die Spalte wahrscheinlich spezifischer parsen möchten.

Im Paket `readr` ist eine schwierige CSV-Datei, mit der sich diese beiden Probleme veranschaulichen lassen:

```
challenge <- read_csv(readr_example("challenge.csv"))
#> Parsed with column specification:
#> cols(
#>   x = col_integer(),
#>   y = col_character()
#> )
#> Warning: 1000 parsing failures.
#>  row col               expected             actual
#> 1001   x no trailing characters .23837975086644292
#> 1002   x no trailing characters .41167997173033655
#> 1003   x no trailing characters .7460716762579978
#> 1004   x no trailing characters .723450553836301
#> 1005   x no trailing characters .614524137461558
#> .... ... ...................... ..................
#> See problems(...) for more details.
```

(Die Funktion `readr_example()` findet den Pfad zu den Dateien, die im Paket enthalten sind.)

Es gibt zwei Ausgaben: die Spaltenspezifikation, die nach Analyse der ersten tausend Zeilen generiert wird, und die ersten fünf Parsing-Fehler. Es empfiehlt sich immer, die Probleme mit `problems()` herauszuziehen, um sie detailliert untersuchen zu können:

```
problems(challenge)
#> # A tibble: 1,000 × 4
#>     row   col               expected             actual
#>   <int> <chr>                  <chr>              <chr>
#> 1  1001     x no trailing characters .23837975086644292
#> 2  1002     x no trailing characters .41167997173033655
#> 3  1003     x no trailing characters  .7460716762579978
#> 4  1004     x no trailing characters   .723450553836301
#> 5  1005     x no trailing characters   .614524137461558
#> 6  1006     x no trailing characters   .473980569280684
#> # ... with 994 more rows
```

Eine gute Strategie ist es, Spalte für Spalte zu bearbeiten, bis keine Probleme mehr auftauchen. Im Beispiel sind jede Menge Parsing-Probleme mit der x-Spalte zu sehen – nach dem Ganzzahlwert folgen weitere Zeichen. Das legt nahe, dass wir stattdessen einen `double`-Parser verwenden müssen. Um den Aufruf zu korrigieren, kopieren Sie die Spaltenspezifikation und fügen sie in den ursprünglichen Aufruf ein:

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_integer(),
    y = col_character()
  )
)
```

Dann können Sie den Typ der x-Spalte anpassen:

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_character()
  )
)
```

Damit ist das erste Problem behoben, doch ein Blick auf die letzten Zeilen zeigt, dass es sich um Datumswerte handelt, die in einem Zeichenvektor gespeichert sind:

```
tail(challenge)
#> # A tibble: 6 × 2
#>       x          y
#>   <dbl>      <chr>
#> 1 0.805 2019-11-21
#> 2 0.164 2018-03-29
#> 3 0.472 2014-08-04
#> 4 0.718 2015-08-16
#> 5 0.270 2020-02-04
#> 6 0.608 2019-01-06
```

Um dies zu korrigieren, geben Sie an, dass y eine Datumsspalte ist:

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_date()
  )
)
tail(challenge)
#> # A tibble: 6 × 2
#>       x          y
#>   <dbl>     <date>
#> 1 0.805 2019-11-21
#> 2 0.164 2018-03-29
#> 3 0.472 2014-08-04
#> 4 0.718 2015-08-16
#> 5 0.270 2020-02-04
#> 6 0.608 2019-01-06
```

Zu jeder `parse_xyz()`-Funktion gehört eine korrespondierende `col_xyz()`-Funktion. Die Funktion `parse_xyz()` verwenden Sie, wenn die Daten bereits in einem Zeichenvektor in R stehen; `col_xyz()` verwenden Sie, wenn Sie **readr** anweisen wollen, die Daten zu laden.

Ich empfehle, immer `col_types` bereitzustellen, was Sie aus der von **readr** gelieferten Ausgabe übernehmen. Damit ist gewährleistet, dass Sie immer ein einheitliches und reproduzierbares Skript für den Datenimport haben. Wenn Sie sich auf die Standardannahmen verlassen und sich Ihre Daten ändern, fährt **readr** fort und

liest sie ein. Möchten Sie wirklich streng sein, nehmen Sie `stop_for_problems()`: Dadurch wird ein Fehler ausgelöst und das Skript gestoppt, falls irgendwelche Parsing-Probleme auftreten.

## Andere Strategien

Es gibt noch einige andere allgemeine Strategien, die Ihnen beim Parsen von Dateien helfen:

- Im vorherigen Beispiel hatten wir ein bisschen Pech: Wenn wir nur eine Zeile mehr als die Standardanzahl betrachten, können wir in einem Zug korrekt parsen:

```
challenge2 <- read_csv(
                readr_example("challenge.csv"),
                guess_max = 1001
              )
#> Parsed with column specification:
#> cols(
#>   x = col_double(),
#>   y = col_date(format = "")
#> )
challenge2
#> # A tibble: 2,000 × 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
```

- Manchmal ist es leichter, Probleme zu diagnostizieren, wenn man einfach alle Spalten als Zeichenvektoren einliest:

```
challenge2 <- read_csv(readr_example("challenge.csv"),
  col_types = cols(.default = col_character())
)
```

  Dies ist insbesondere nützlich in Verbindung mit der Funktion `type_convert()`, die die Parsing-Heuristik auf die Zeichenspalten in einem Dataframe anwendet:

```
df <- tribble(
  ~x,    ~y,
  "1",  "1.21",
  "2",  "2.32",
  "3",  "4.56"
)
df
#> # A tibble: 3 × 2
#>       x     y
```

```
#>   <chr> <chr>
#> 1     1  1.21
#> 2     2  2.32
#> 3     3  4.56

# Note the column types
type_convert(df)
#> Parsed with column specification:
#> cols(
#>   x = col_integer(),
#>   y = col_double()
#> )
#> # A tibble: 3 × 2
#>       x     y
#>   <int> <dbl>
#> 1     1  1.21
#> 2     2  2.32
#> 3     3  4.56
```

- Wenn Sie eine sehr große Datei einlesen, werden Sie `n_max` auf eine eher kleine Zahl wie 10.000 oder 100.000 setzen. Das beschleunigt die Durchläufe, während Sie allgemeine Probleme eliminieren.
- Sollten größere Parsing-Probleme auftreten, ist es manchmal einfacher, lediglich einen Zeichenvektor der Zeilen mit `read_lines()` oder sogar einen Zeichenvektor der Länge 1 mit `read_file()` einzulesen. Dann können Sie Ihre Kenntnisse zum Parsen von Strings (was Sie später noch lernen) anwenden, um exotischere Formate zu parsen.

## In eine Datei schreiben

Das Paket **readr** enthält auch zwei nützliche Funktionen, um Daten zurück auf die Festplatte zu schreiben: `write_csv()` und `write_tsv()`. Beide Funktionen erhöhen die Chancen, dass die Ausgabedatei korrekt wieder zurückgelesen wird, denn sie

- codieren Strings immer in UTF-8,
- speichern Datumswerte und Datums-/Zeitwerte im ISO8601-Format, sodass diese auch an anderen Stellen leicht geparst werden können.

Wenn Sie eine CSV-Datei nach Excel exportieren möchten, nehmen Sie die Funktion `write_excel_csv()` – diese schreibt nämlich ein spezielles Zeichen (eine »Bytereihenfolgemarke«) an den Anfang der Datei, an der Excel erkennt, dass die UTF-8-Codierung verwendet wird.

Die wichtigsten Argumente sind `x` (der zu speichernde Dataframe) und `path` (der Speicherort). Außerdem können Sie festlegen, wie fehlende Werte mit `na` geschrieben werden und ob Sie die Ausgabe an eine vorhandene Datei anfügen wollen:

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

Beachten Sie, dass die Typinformationen verloren gehen, wenn Sie in eine CSV-Datei speichern:

```
challenge
#> # A tibble: 2,000 × 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
write_csv(challenge, "challenge-2.csv")
read_csv("challenge-2.csv")
#> Parsed with column specification:
#> cols(
#>   x = col_double(),
#>   y = col_character()
#> )
#> # A tibble: 2,000 × 2
#>       x     y
#>   <dbl> <chr>
#> 1   404  <NA>
#> 2  4172  <NA>
#> 3  3004  <NA>
#> 4   787  <NA>
#> 5    37  <NA>
#> 6  2332  <NA>
#> # ... with 1,994 more rows
```

CSV-Dateien sind deswegen ein wenig unzuverlässig, um Zwischenergebnisse zu speichern – Sie müssen die Spaltenspezifikation jedes Mal erneut herstellen, wenn Sie die Datei laden. Es gibt aber zwei Alternativen:

- Die Funktionen `write_rds()` und `read_rds()` sind einheitliche Wrapper um die Basisfunktionen `readRDS()` und `saveRDS()`. Diese speichern Daten im benutzerdefinierten Binärformat von R namens RDS:

```
write_rds(challenge, "challenge.rds")
read_rds("challenge.rds")
#> # A tibble: 2,000 × 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
```

- Das Paket **feather** implementiert ein schnelles Binärdateiformat, das sich über Programmiersprachen hinweg gemeinsam nutzen lässt:

```
library(feather)
write_feather(challenge, "challenge.feather")
read_feather("challenge.feather")
#> # A tibble: 2,000 x 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
```

Gegenüber RDS ist **feather** tendenziell schneller und auch außerhalb von R verwendbar. RDS unterstützt Listenspalten (dazu mehr in Kapitel 20), **feather** derzeit nicht.

## Andere Datentypen

Um Daten anderer Typen nach R zu bekommen, sollten Sie mit den nachfolgend aufgeführten Tidyverse-Paketen beginnen. Zweifellos sind sie nicht perfekt, doch immerhin eine gute Ausgangsbasis. Für »rechteckige« Daten gibt es die folgenden Pakete:

- **haven** liest SPSS-, Stata- und SAS-Dateien.
- **readxl** liest Excel-Dateien (sowohl *.xls* als auch *.xlsx*).
- **DBI** erlaubt es, zusammen mit einem datenbankspezifischen Backend (zum Beispiel **RMySQL**, **RSQLite**, **RPostgreSQL** und so weiter), SQL-Abfragen gegen eine Datenbank auszuführen und einen Dataframe zurückzugeben.

Für hierarchische Daten können Sie **jsonlite** (von Jeroen Ooms) für JSON und **xml2** für XML verwenden. Jenny Bryan hat einige ausgezeichnet ausgearbeitete Beispiele unter *https://jennybc.github.io/purrr-tutorial/* veröffentlicht.

Für andere Dateitypen sollten Sie das R-Manual zum Import/Export konsultieren (*https://cran.r-project.org/doc/manuals/r-release/R-data.html*) und das Paket **rio** (*https://github.com/leeper/rio*) ausprobieren.