# Data import

## Introduction

Working with data provided by R packages is a great way to learn the tools of data science, but at some point you want to stop learning and start working with your own data. In this chapter, you'll learn how to read plain-text rectangular files into R. Here, we'll only scratch the surface of data import, but many of the principles will translate to other forms of data. We'll finish with a few pointers to packages that are useful for other types of data.

### Prerequisites

In this chapter, you'll learn how to load flat files in R with the __readr__ package, which is part of the core tidyverse.

```{r setup, message = FALSE}
library(tidyverse)
```

## Getting started

Most of readr's functions are concerned with turning flat files into data frames:

* `read_csv()` reads comma delimited files, `read_csv2()` reads semicolon
  separated files (common in countries where `,` is used as the decimal place),
  `read_tsv()` reads tab delimited files, and `read_delim()` reads in files
  with any delimiter.

* `read_fwf()` reads fixed width files. You can specify fields either by their
  widths with `fwf_widths()` or their position with `fwf_positions()`.
  `read_table()` reads a common variation of fixed width files where columns
  are separated by white space.

* `read_log()` reads Apache style log files. (But also check out
  [webreadr](https://github.com/Ironholds/webreadr) which is built on top
  of `read_log()` and provides many more helpful tools.)

These functions all have similar syntax: once you've mastered one, you can use the others with ease. For the rest of this chapter we'll focus on `read_csv()`. Not only are csv files one of the most common forms of data storage, but once you understand `read_csv()`, you can easily apply your knowledge to all the other functions in readr.

The first argument to `read_csv()` is the most important: it's the path to the file to read.

```{r, message = TRUE}
heights <- read_csv("data/heights.csv")
```

When you run `read_csv()` it prints out a column specification that gives the name and type of each column. That's an important part of readr, which we'll come back to in [parsing a file].

You can also supply an inline csv file. This is useful for experimenting with readr and for creating reproducible examples to share with others:

```{r}
read_csv("a,b,c
1,2,3
4,5,6")
```

In both cases `read_csv()` uses the first line of the data for the column names, which is a very common convention. There are two cases where you might want to tweak this behaviour:

1. Sometimes there are a few lines of metadata at the top of the file. You can
    use `skip = n` to skip the first `n` lines; or use `comment = "#"` to drop
    all lines that start with (e.g.) `#`.

    ```{r}
    read_csv("The first line of metadata
      The second line of metadata
      x,y,z
      1,2,3", skip = 2)

    read_csv("# A comment I want to skip
      x,y,z
      1,2,3", comment = "#")
    ```

1. The data might not have column names. You can use `col_names = FALSE` to
    tell `read_csv()` not to treat the first row as headings, and instead
    label them sequentially from `X1` to `Xn`:

    ```{r}
    read_csv("1,2,3\n4,5,6", col_names = FALSE)
    ```

    (`"\n"` is a convenient shortcut for adding a new line. You'll learn more
    about it and other types of string escape in [string basics].)

    Alternatively you can pass `col_names` a character vector which will be used as the column names:

    ```{r}
    read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
    ```

Another option that commonly needs tweaking is `na`: this specifies the value (or values) that are used to represent missing values in your file:

```{r}
read_csv("a,b,c\n1,2,.", na = ".")
```

This is all you need to know to read ~75% of CSV files that you'll encounter in practice. You can also easily adapt what you've learned to read tab separated files with `read_tsv()` and fixed width files with `read_fwf()`. To read in more challenging files, you'll need to learn more about how readr parses each column, turning them into R vectors.

### Compared to base R

If you've used R before, you might wonder why we're not using `read.csv()`. There are a few good reasons to favour readr functions over the base equivalents:

* They are typically much faster (~10x) than their base equivalents. Long running jobs have a progress bar, so you can see what's happening. If you're looking for raw speed, try `data.table::fread()`. It doesn't fit quite so well into the tidyverse, but it can be quite a bit faster.

* They produce tibbles, they don't convert character vectors to factors, use row names, or munge the column names. These are common sources of frustration with the base R functions.

* They are more reproducible. Base R functions inherit some behaviour from your operating system and environment variables, so import code that works on your computer might not work on someone else's.

### Exercises

1. What function would you use to read a file where fields were separated with
    "|"?

1. Apart from `file`, `skip`, and `comment`, what other arguments do
    `read_csv()` and `read_tsv()` have in common?

1. What are the most important arguments to `read_fwf()`?

1. Sometimes strings in a CSV file contain commas. To prevent them from causing problems they need to be surrounded by a quoting character, like
    `"` or `'`. By convention, `read_csv()` assumes that the quoting
    character will be `"`, and if you want to change it you'll need to
    use `read_delim()` instead. What arguments do you need to specify
    to read the following text into a data frame?

    ```{r, eval = FALSE}
    "x,y\n1,'a,b'"
    ```

1. Identify what is wrong with each of the following inline CSV files.
    What happens when you run the code?

    ```{r, eval = FALSE}
    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")
    ```

## Parsing a vector

Before we get into the details of how readr reads files from disk, we need to take a little detour to talk about the `parse_*()` functions. These functions take a character vector and return a more specialised vector like a logical, integer, or date:

```{r}
str(parse_logical(c("TRUE", "FALSE", "NA")))
str(parse_integer(c("1", "2", "3")))
str(parse_date(c("2010-01-01", "1979-10-14")))
```

These functions are useful in their own right, but are also an important building block for readr. Once you've learned how the individual parsers work in this section, we'll circle back and see how they fit together to parse a complete file in the next section.

Like all functions in the tidyverse, the `parse_*()` functions are uniform: the first argument is a character vector to parse, and the `na` argument specifies which strings should be treated as missing:

```{r}
parse_integer(c("1", "231", ".", "456"), na = ".")
```

If parsing fails, you'll get a warning:

```{r}
x <- parse_integer(c("123", "345", "abc", "123.45"))
```

And the failures will be missing in the output:

```{r}
x
```

If there are many parsing failures, you'll need to use `problems()` to
get the complete set. This returns a tibble, which you can then
manipulate with dplyr.

```{r}
problems(x)
```

Using parsers is mostly a matter of understanding what's available and
how they deal with different types of input. There are eight
particularly important parsers:

1. `parse_logical()` and `parse_integer()` parse logicals and integers
    respectively. There's basically nothing that can go wrong with
these
    parsers so I won't describe them here further.

1. `parse_double()` is a strict numeric parser, and `parse_number()`
    is a flexible numeric parser. These are more complicated than you
might
    expect because different parts of the world write numbers in
different
    ways.

1. `parse_character()` seems so simple that it shouldn't be necessary.
But
    one complication makes it quite important: character encodings.

1. `parse_factor()` create factors, the data structure that R uses to
represent
    categorical variables with fixed and known values.

1. `parse_datetime()`, `parse_date()`, and `parse_time()` allow you to
    parse various date & time specifications. These are the most
complicated
    because there are so many different ways of writing dates.

The following sections describe these parsers in more detail.

### Numbers

It seems like it should be straightforward to parse a number, but three
problems make it tricky:

1. People write numbers differently in different parts of the world.
   For example, some countries use `.` in between the integer and
fractional
   parts of a real number, while others use `,`.

1. Numbers are often surrounded by other characters that provide some
   context, like "$1000" or "10%".

1. Numbers often contain "grouping" characters to make them easier to
read,
   like "1,000,000", and these grouping characters vary around the
world.

To address the first problem, readr has the notion of a "locale", an
object that specifies parsing options that differ from place to place.
When parsing numbers, the most important option is the character you
use for the decimal mark. You can override the default value of `.` by
creating a new locale and setting the `decimal_mark` argument:

```{r}
parse_double("1.23")
parse_double("1,23", locale = locale(decimal_mark = ","))
```

readr's default locale is US-centric, because generally R is US-centric
(i.e. the documentation of base R is written in American English). An
alternative approach would be to try and guess the defaults from your
operating system. This is hard to do well, and, more importantly, makes
your code fragile: even if it works on your computer, it might fail
when you email it to a colleague in another country.

`parse_number()` addresses the second problem: it ignores non-numeric
characters before and after the number. This is particularly useful for
currencies and percentages, but also works to extract numbers embedded
in text.

```{r}
parse_number("$100")
parse_number("20%")
parse_number("It cost $123.45")
```

The final problem is addressed by the combination of `parse_number()`
and the locale as `parse_number()` will ignore the "grouping mark":

```{r}
# Used in America
parse_number("$123,456,789")

# Used in many parts of Europe
parse_number("123.456.789", locale = locale(grouping_mark = "."))

# Used in Switzerland
parse_number("123'456'789", locale = locale(grouping_mark = "'"))
```

### Strings {#readr-strings}

It seems like `parse_character()` should be really simple --- it could just return its input. Unfortunately life isn't so simple, as there are multiple ways to represent the same string. To understand what's going on, we need to dive into the details of how computers represent strings. In R, we can get at the underlying representation of a string using `charToRaw()`:

```{r}
charToRaw("Hadley")
```

Each hexadecimal number represents a byte of information: `48` is H, `61` is a, and so on. The mapping from hexadecimal number to character is called the encoding, and in this case the encoding is called ASCII. ASCII does a great job of representing English characters, because it's the __American__ Standard Code for Information Interchange.

Things get more complicated for languages other than English. In the early days of computing there were many competing standards for encoding non-English characters, and to correctly interpret a string you needed to know both the values and the encoding. For example, two common encodings are Latin1 (aka ISO-8859-1, used for Western European languages) and Latin2 (aka ISO-8859-2, used for Eastern European languages). In Latin1, the byte `b1` is "±", but in Latin2, it's "ą"! Fortunately, today there is one standard that is supported almost everywhere: UTF-8. UTF-8 can encode just about every character used by humans today, as well as many extra symbols (like emoji!).

readr uses UTF-8 everywhere: it assumes your data is UTF-8 encoded when you read it, and always uses it when writing. This is a good default, but will fail for data produced by older systems that don't understand UTF-8. If this happens to you, your strings will look weird when you print them. Sometimes just one or two characters might be messed up; other times you'll get complete gibberish. For example:

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

x1
x2
```

To fix the problem you need to specify the encoding in `parse_character()`:

```{r}
parse_character(x1, locale = locale(encoding = "Latin1"))
parse_character(x2, locale = locale(encoding = "Shift-JIS"))
```

How do you find the correct encoding? If you're lucky, it'll be included somewhere in the data documentation. Unfortunately, that's rarely the case, so readr provides `guess_encoding()` to help you figure it out. It's not foolproof, and it works better when you have lots of text (unlike here), but it's a reasonable place to start. Expect to try a few different encodings before you find the right one.

```{r}
guess_encoding(charToRaw(x1))
guess_encoding(charToRaw(x2))
```

The first argument to `guess_encoding()` can either be a path to a file, or, as in this case, a raw vector (useful if the strings are already in R).

Encodings are a rich and complex topic, and I've only scratched the surface here. If you'd like to learn more I'd recommend reading the detailed explanation at <http://kunststube.net/encoding/>.

### Factors {#readr-factors}

R uses factors to represent categorical variables that have a known set of possible values. Give `parse_factor()` a vector of known `levels` to generate a warning whenever an unexpected value is present:

```{r}
fruit <- c("apple", "banana")
parse_factor(c("apple", "banana", "bananana"), levels = fruit)
```

But if you have many problematic entries, it's often easier to leave as character vectors and then use the tools you'll learn about in [strings] and [factors] to clean them up.

### Dates, date-times, and times {#readr-datetimes}

You pick between three parsers depending on whether you want a date (the number of days since 1970-01-01), a date-time (the number of seconds since midnight 1970-01-01), or a time (the number of seconds since midnight). When called without any additional arguments:

*   `parse_datetime()` expects an ISO8601 date-time. ISO8601 is an
    international standard in which the components of a date are
    organised from biggest to smallest: year, month, day, hour, minute,
    second.

    ```{r}
    parse_datetime("2010-10-01T2010")
    # If time is omitted, it will be set to midnight
    parse_datetime("20101010")
    ```

    This is the most important date/time standard, and if you work with
    dates and times frequently, I recommend reading
    <https://en.wikipedia.org/wiki/ISO_8601>

*   `parse_date()` expects a four digit year, a `-` or `/`, the month, a `-`
    or `/`, then the day:

~~~~
    ```{r}
    parse_date("2010-10-01")
    ```

*   `parse_time()` expects the hour, `:`, minutes, optionally `:` and
seconds,
    and an optional am/pm specifier:

    ```{r}
    library(hms)
    parse_time("01:10 am")
    parse_time("20:10:01")
    ```

    Base R doesn't have a great built in class for time data, so we use
    the one provided in the hms package.

If these defaults don't work for your data you can supply your own
date-time `format`, built up of the following pieces:

Year
: `%Y` (4 digits).
: `%y` (2 digits); 00-69 -> 2000-2069, 70-99 -> 1970-1999.

Month
: `%m` (2 digits).
: `%b` (abbreviated name, like "Jan").
: `%B` (full name, "January").

Day
: `%d` (2 digits).
: `%e` (optional leading space).

Time
: `%H` 0-23 hour.
: `%I` 0-12, must be used with `%p`.
: `%p` AM/PM indicator.
: `%M` minutes.
: `%S` integer seconds.
: `%OS` real seconds.
: `%Z` Time zone (as name, e.g. `America/Chicago`). Beware of
abbreviations:
  if you're American, note that "EST" is a Canadian time zone that does
not
  have daylight savings time. It is _not_ Eastern Standard Time! We'll
  come back to this [time zones].
: `%z` (as offset from UTC, e.g. `+0800`).

Non-digits
: `%.` skips one non-digit character.
: `%*` skips any number of non-digits.

The best way to figure out the correct format is to create a few
examples in a character vector, and test with one of the parsing
functions. For example:

```{r}
parse_date("01/02/15", "%m/%d/%y")
parse_date("01/02/15", "%d/%m/%y")
parse_date("01/02/15", "%y/%m/%d")
```

If you're using `%b` or `%B` with non-English month names, you'll need
to set the  `lang` argument to `locale()`. See the list of built-in
languages in `date_names_langs()`, or if your language is not already
included, create your own with `date_names()`.

```{r}
parse_date("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
```

### Exercises

1.  What are the most important arguments to `locale()`?

1.  What happens if you try and set `decimal_mark` and `grouping_mark`
    to the same character? What happens to the default value of
    `grouping_mark` when you set `decimal_mark` to ","? What happens
    to the default value of `decimal_mark` when you set the
`grouping_mark`
    to "."?

1.  I didn't discuss the `date_format` and `time_format` options to
    `locale()`. What do they do? Construct an example that shows when
    they might be useful.

1.  If you live outside the US, create a new locale object that
encapsulates
    the settings for the types of file you read most commonly.

1.  What's the difference between `read_csv()` and `read_csv2()`?

1.  What are the most common encodings used in Europe? What are the
    most common encodings used in Asia? Do some googling to find out.

1.  Generate the correct format string to parse each of the following
    dates and times:

    ```{r}
    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"
    ```

## Parsing a file
~~~~

Now that you've learned how to parse an individual vector, it's time to return to the beginning and explore how readr parses a file. There are two new things that you'll learn about in this section:

1. How readr automatically guesses the type of each column.
1. How to override the default specification.

### Strategy

readr uses a heuristic to figure out the type of each column: it reads the first 1000 rows and uses some (moderately conservative) heuristics to figure out the type of each column. You can emulate this process with a character vector using `guess_parser()`, which returns readr's best guess, and `parse_guess()` which uses that guess to parse the column:

```{r}
guess_parser("2010-10-01")
guess_parser("15:01")
guess_parser(c("TRUE", "FALSE"))
guess_parser(c("1", "5", "9"))
guess_parser(c("12,352,561"))

str(parse_guess("2010-10-10"))
```

The heuristic tries each of the following types, stopping when it finds a match:

* logical: contains only "F", "T", "FALSE", or "TRUE".
* integer: contains only numeric characters (and `-`).
* double: contains only valid doubles (including numbers like `4.5e-5`).
* number: contains valid doubles with the grouping mark inside.
* time: matches the default `time_format`.
* date: matches the default `date_format`.
* date-time: any ISO8601 date.

If none of these rules apply, then the column will stay as a vector of strings.

### Problems

These defaults don't always work for larger files. There are two basic problems:

1. The first thousand rows might be a special case, and readr guesses a type that is not sufficiently general. For example, you might have a column of doubles that only contains integers in the first 1000 rows.

1. The column might contain a lot of missing values. If the first 1000 rows contain only `NA`s, readr will guess that it's a character vector, whereas you probably want to parse it as something more specific.

readr contains a challenging CSV that illustrates both of these problems:

```{r}
challenge <- read_csv(readr_example("challenge.csv"))
```

(Note the use of `readr_example()` which finds the path to one of the files included with the package)

There are two printed outputs: the column specification generated by looking at the first 1000 rows, and the first five parsing failures. It's always a good idea to explicitly pull out the `problems()`, so you can explore them in more depth:

```{r}
problems(challenge)
```

A good strategy is to work column by column until there are no problems remaining. Here we can see that there are a lot of parsing problems with the `x` column - there are trailing characters after the integer value. That suggests we need to use a double parser instead.

To fix the call, start by copying and pasting the column specification into your original call:

```{r, eval = FALSE}
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_integer(),
    y = col_character()
  )
)
```

Then you can tweak the type of the `x` column:

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

That fixes the first problem, but if we look at the last few rows, you'll see that they're dates stored in a character vector:

```{r}
tail(challenge)
```

You can fix that by specifying that `y` is a date column:

```{r}
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_date()
  )
)
tail(challenge)
```

Every `parse_xyz()` function has a corresponding `col_xyz()` function.
You use `parse_xyz()` when the data is in a character vector in R
already; you use `col_xyz()` when you want to tell readr how to load
the data.

I highly recommend always supplying `col_types`, building up from the
print-out provided by readr. This ensures that you have a consistent
and reproducible data import script. If you rely on the default guesses
and your data changes, readr will continue to read it in. If you want
to be really strict, use `stop_for_problems()`: that will throw an
error and stop your script if there are any parsing problems.

### Other strategies

There are a few other general strategies to help you parse files:

*   In the previous example, we just got unlucky: if we look at just
    one more row than the default, we can correctly parse in one shot:

    ```{r}
    challenge2 <- read_csv(readr_example("challenge.csv"), guess_max = 1001)
    challenge2
    ```

*   Sometimes it's easier to diagnose problems if you just read in all
    the columns as character vectors:

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

    This is particularly useful in conjunction with `type_convert()`,
    which applies the parsing heuristics to the character columns in a data
    frame.

    ```{r}
    df <- tribble(
      ~x,  ~y,
      "1", "1.21",
      "2", "2.32",
      "3", "4.56"
    )
    df

    # Note the column types
    type_convert(df)
    ```

*   If you're reading a very large file, you might want to set `n_max` to
    a smallish number like 10,000 or 100,000. That will accelerate your
    iterations while you eliminate common problems.

*   If you're having major parsing problems, sometimes it's easier
    to just read into a character vector of lines with `read_lines()`,
    or even a character vector of length 1 with `read_file()`. Then you
    can use the string parsing skills you'll learn later to parse
    more exotic formats.

## Writing to a file

readr also comes with two useful functions for writing data back to
disk: `write_csv()` and `write_tsv()`. Both functions increase the
chances of the output file being read back in correctly by:

* Always encoding strings in UTF-8.

* Saving dates and date-times in ISO8601 format so they are easily
  parsed elsewhere.

If you want to export a csv file to Excel, use `write_excel_csv()` ---
this writes a special character (a "byte order mark") at the start of
the file which tells Excel that you're using the UTF-8 encoding.

The most important arguments are `x` (the data frame to save), and
`path` (the location to save it). You can also specify how missing
values are written with `na`, and if you want to `append` to an
existing file.

```{r, eval = FALSE}
write_csv(challenge, "challenge.csv")
```

Note that the type information is lost when you save to csv:

```{r, warning = FALSE}
challenge
write_csv(challenge, "challenge-2.csv")
read_csv("challenge-2.csv")
```

~~~
```

This makes CSVs a little unreliable for caching interim results---you
need to recreate the column specification every time you load in. There
are two alternatives:

1.  `write_rds()` and `read_rds()` are uniform wrappers around the base
    functions `readRDS()` and `saveRDS()`. These store data in R's
custom
    binary format called RDS:

    ```{r}
    write_rds(challenge, "challenge.rds")
    read_rds("challenge.rds")
    ```

1.  The feather package implements a fast binary file format that can
    be shared across programming languages:

    ```{r, eval = FALSE}
    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
    ```

Feather tends to be faster than RDS and is usable outside of R. RDS
supports list-columns (which you'll learn about in [many models]);
feather currently does not.

```{r, include = FALSE}
file.remove("challenge-2.csv")
file.remove("challenge.rds")
```

## Other types of data

To get other types of data into R, we recommend starting with the
tidyverse packages listed below. They're certainly not perfect, but
they are a good place to start. For rectangular data:

* __haven__ reads SPSS, Stata, and SAS files.

* __readxl__ reads excel files (both `.xls` and `.xlsx`).

* __DBI__, along with a database specific backend (e.g. __RMySQL__,
  __RSQLite__, __RPostgreSQL__ etc) allows you to run SQL queries
against a
  database and return a data frame.

For hierarchical data: use __jsonlite__ (by Jeroen Ooms) for json, and
__xml2__ for XML. Jenny Bryan has some excellent worked examples at
<https://jennybc.github.io/purrr-tutorial/>.

For other file types, try the [R data import/export manual]
(https://cran.r-project.org/doc/manuals/r-release/R-data.html) and the
[__rio__](https://github.com/leeper/rio) package.
~~~