7 Data import

7.1 Introduction

Working with data provided by R packages is a great way to learn data science tools, but you want to apply what you've learned to your own data at some point. In this chapter, you'll learn the basics of reading data files into R.

Specifically, this chapter will focus on reading plain-text rectangular files. We'll start with practical advice for handling features like column names, types, and missing data. You will then learn about reading data from multiple files at once and writing data from R to a file. Finally, you'll learn how to handcraft data frames in R.

7.1.1 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.

```
library(tidyverse)
```

7.2 Reading data from a file

To begin, we'll focus on the most common rectangular data file type: CSV, which is short for comma-separated values. Here is what a simple CSV file looks like. The first row, commonly called the header row, gives the column names, and the following six rows provide the data. The columns are separated, aka delimited, by commas.

```
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
```

Table 7.1 shows a representation of the same data as a table.

Table 7.1: Data from the students.csv file as a table.

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

We can read this file into R using `read_csv()`. The first argument is the most important: the path to the file. You can think about the path as the address of the file: the file is called `students.csv` and that it lives in the `data` folder.

```
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.
```

The code above will work if you have the `students.csv` file in a `data` folder in your project. You can download the `students.csv` file from https://pos.it/r4ds-students-csv or you can read it directly from that URL with:

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

When you run `read_csv()`, it prints out a message telling you the number of rows and columns of data, the delimiter that was used, and the column specifications (names of columns organized by the type of data the column contains). It also prints out some information about retrieving the full column specification and how to quiet this message. This message is an integral part of readr, and we'll return to it in Section 7.3.

7.2.1 Practical advice

Once you read data in, the first step usually involves transforming it in some way to make it easier to work with in the rest of your analysis. Let's take another look at the `students` data with that in mind.

```
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 the `favourite.food` column, there are a bunch of food items, and then the character string `N/A`, which should have been a real `NA` that R will recognize as "not available". This is something we can address using the `na` argument. By default, `read_csv()` only recognizes empty strings (`""`) in this dataset as `NA`s, and we want it to also recognize the character string `"N/A"`.

```
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
```

You might also notice that the `Student ID` and `Full Name` columns are surrounded by backticks. That's because they contain spaces, breaking R's usual rules for variable names; they're **non-syntactic** names. To refer to these variables, you need to surround them with backticks, ` :

```
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
```

An alternative approach is to use `janitor::clean_names()` to use some heuristics to turn them all into snake case at once[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
```

Another common task after reading in data is to consider variable types. For example, `meal_plan` is a categorical variable with a known set of possible values, which in R should be represented as a factor:

```
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
```

Note that the values in the `meal_plan` variable have stayed the same, but the type of variable denoted underneath the variable name has changed from character (`<chr>`) to factor (`<fct>`). You'll learn more about factors in Chapter 16.

Before you analyze these data, you'll probably want to fix the `age` column. Currently, `age` is a character variable because one of the observations is typed out as `five` instead of a numeric `5`. We discuss the details of fixing this issue in Chapter 20.

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

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
```

A new function here is `if_else()`, which has three arguments. The first argument `test` should be a logical vector. The result will contain the value of the second argument, `yes`, when `test` is `TRUE`, and the value of the third argument, `no`, when it is `FALSE`. Here we're saying if `age` is the character string `"five"`, make it `"5"`, and if not leave it as `age`. You will learn more about `if_else()` and logical vectors in Chapter 12.

7.2.2 Other arguments

There are a couple of other important arguments that we need to mention, and they'll be easier to demonstrate if we first show you a handy trick: `read_csv()` can read text strings that you've created and formatted like a CSV file:

```
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
```

Usually, `read_csv()` uses the first line of the data for the column names, which is a very common convention. But it's not uncommon for a few lines of metadata to be included 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.) `#`:

```
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
```

In other cases, 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`:

```
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
```

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

```
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
```

These arguments are all you need to know to read the majority of CSV files that you'll encounter in practice. (For the rest, you'll need to carefully inspect your `.csv` file and read the documentation for `read_csv()`'s many other arguments.)

7.2.3 Other file types

Once you've mastered `read_csv()`, using readr's other functions is straightforward; it's just a matter of knowing which function to reach for:

- `read_csv2()` reads semicolon-separated files. These use `;` instead of `,` to separate fields and are common in countries that use `,` as the decimal marker.
- `read_tsv()` reads tab-delimited files.
- `read_delim()` reads in files with any delimiter, attempting to automatically guess the delimiter if you don't specify it.
- `read_fwf()` reads fixed-width files. You can specify fields by their widths with `fwf_widths()` or by their positions 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.

7.2.4 Exercises

1. What function would you use to read a file where fields were separated with "|"?
2. Apart from `file`, `skip`, and `comment`, what other arguments do `read_csv()` and `read_tsv()` have in common?
3. What are the most important arguments to `read_fwf()`?
4. 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 default, `read_csv()` assumes that the quoting character will be `"`. To read the following text into a data frame, what argument to `read_csv()` do you need to specify?

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

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

   ```
   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. Practice referring to non-syntactic names in the following data frame by:

a. Extracting the variable called `1`.
b. Plotting a scatterplot of `1` vs. `2`.
c. Creating a new column called `3`, which is `2` divided by `1`.
d. Renaming the columns to `one`, `two`, and `three`.

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

7.3 Controlling column types

A CSV file doesn't contain any information about the type of each variable (i.e. whether it's a logical, number, string, etc.), so readr will try to guess the type. This section describes how the guessing process works, how to resolve some common problems that cause it to fail, and, if needed, how to supply the column types yourself. Finally, we'll mention a few general strategies that are useful if readr is failing catastrophically and you need to get more insight into the structure of your file.

7.3.1 Guessing types

readr uses a heuristic to figure out the column types. For each column, it pulls the values of 1,000[2] rows spaced evenly from the first row to the last, ignoring missing values. It then works through the following questions:

- Does it contain only `F`, `T`, `FALSE`, or `TRUE` (ignoring case)? If so, it's a logical.
- Does it contain only numbers (e.g., `1`, `-4.5`, `5e6`, `Inf`)? If so, it's a number.
- Does it match the ISO8601 standard? If so, it's a date or date-time. (We'll return to date-times in more detail in Section 17.2).
- Otherwise, it must be a string.

You can see that behavior in action in this simple example:

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

This heuristic works well if you have a clean dataset, but in real life, you'll encounter a selection of weird and beautiful failures.

7.3.2 Missing values, column types, and problems

The most common way column detection fails is that a column contains unexpected values, and you get a character column instead of a more specific type. One of the most common causes for this is a missing value, recorded using something other than the `NA` that readr expects.

Take this simple 1 column CSV file as an example:

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

If we read it without any additional arguments, `x` becomes a character column:

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

In this very small case, you can easily see the missing value `.`. But what happens if you have thousands of rows with only a few missing values represented by `.`s sprinkled among them? One approach is to tell readr that `x` is a numeric column, and then see where it fails. You can do that with the `col_types` argument, which takes a named list where the names match the column names in the CSV file:

```
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)
```

Now `read_csv()` reports that there was a problem, and tells us we can find out more with `problems()`:

```
problems(df)
#> # A tibble: 1 × 5
#>     row   col expected actual file
#>   <int> <int> <chr>    <chr>  <chr>
#> 1     3     1 a double .      /tmp/RtmpuU0idZ/file1bf94ee4cdfe
```

This tells us that there was a problem in row 3, col 1 where readr expected a double but got a `.`. That suggests this dataset uses `.` for missing values. So then we set `na = "."`, the automatic guessing succeeds, giving us the numeric column that we want:

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

7.3.3 Column types

readr provides a total of nine column types for you to use:

- `col_logical()` and `col_double()` read logicals and real numbers. They're relatively rarely needed (except as above), since readr will usually guess them for you.
- `col_integer()` reads integers. We seldom distinguish integers and doubles in this book because they're functionally equivalent, but reading integers explicitly can occasionally be useful because they occupy half the memory of doubles.
- `col_character()` reads strings. This can be useful to specify explicitly when you have a column that is a numeric identifier, i.e., long series of digits that identifies an object but doesn't make sense to apply mathematical operations to. Examples include phone numbers, social security numbers, credit card numbers, etc.
- `col_factor()`, `col_date()`, and `col_datetime()` create factors, dates, and date-times respectively; you'll learn more about those when we get to those data types in Chapter 16 and Chapter 17.
- `col_number()` is a permissive numeric parser that will ignore non-numeric components, and is particularly useful for currencies. You'll learn more about it in Chapter 13.
- `col_skip()` skips a column so it's not included in the result, which can be useful for speeding up reading the data if you have a large CSV file and you only want to use some of the columns.

It's also possible to override the default column by switching from `list()` to `cols()` and specifying `.default`:

```
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
```

Another useful helper is `cols_only()` which will read in only the columns you specify:

```
read_csv(
  another_csv,
  col_types = cols_only(x = col_character())
)
```

```
#> # A tibble: 1 × 1
#>   x
#>   <chr>
#> 1 1
```

7.4 Reading data from multiple files

Sometimes your data is split across multiple files instead of being contained in a single file. For example, you might have sales data for multiple months, with each month's data in a separate file: `01-sales.csv` for January, `02-sales.csv` for February, and `03-sales.csv` for March. With `read_csv()` you can read these data in at once and stack them on top of each other in a single data frame.

```
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
#> # ℹ 13 more rows
```

Once again, the code above will work if you have the CSV files in a `data` folder in your project. You can download these files from https://pos.it/r4ds-01-sales, https://pos.it/r4ds-02-sales, and https://pos.it/r4ds-03-sales or you can read them directly with:

```
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")
```

The `id` argument adds a new column called `file` to the resulting data frame that identifies the file the data come from. This is especially helpful in circumstances where the files you're reading in do not have an identifying column that can help you trace the observations back to their original sources.

If you have many files you want to read in, it can get cumbersome to write out their names as a list. Instead, you can use the base `list.files()` function to find the files for you by matching a pattern in the file names. You'll learn more about these patterns in Chapter 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"
```

7.5 Writing to a file

readr also comes with two useful functions for writing data back to disk: `write_csv()` and `write_tsv()`. The most important arguments to these functions are `x` (the data frame to save) and `file` (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.

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

Now let's read that csv file back in. Note that the variable type information that you just set up is lost when you save to CSV because you're starting over with reading from a plain text file again:

```
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
```

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 main 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. This means that when you reload the object, you are loading the *exact same* R object that you stored.

   ```
   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
```

2. The arrow package allows you to read and write parquet files, a fast binary file format that can be shared across programming languages. We'll return to arrow in more depth in Chapter 22.

```
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 tends to be much faster than RDS and is usable outside of R, but does require the arrow package.

7.6 Data entry

Sometimes you'll need to assemble a tibble "by hand" doing a little data entry in your R script. There are two useful functions to help you do this which differ in whether you layout the tibble by columns or by rows. `tibble()` works by column:

```
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
```

Laying out the data by column can make it hard to see how the rows are related, so an alternative is `tribble()`, short for **tr**ansposed t**ibble**, which lets you lay out your data row by row. `tribble()` is customized for data entry in code: column headings start with `~` and entries are separated by commas. This makes it possible to lay out small amounts of data in an easy to read form:

```
tribble(
  ~x, ~y, ~z,
  1, "h", 0.08,
  2, "m", 0.83,
  5, "g", 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
```

7.7 Summary

In this chapter, you've learned how to load CSV files with `read_csv()` and to do your own data entry with `tibble()` and `tribble()`. You've learned how csv files work, some of the problems you might encounter, and how to overcome them. We'll come to data import a few times in this book: Chapter 20 from Excel and Google Sheets, Chapter 21 will show you how to load data from databases, Chapter 22 from parquet files, Chapter 23 from JSON, and Chapter 24 from websites.

We're just about at the end of this section of the book, but there's one important last topic to cover: how to get help. So in the next chapter, you'll learn some good places to look for help, how to create a reprex to maximize your chances of getting good help, and some general advice on keeping up with the world of R.

---

1. The janitor package is not part of the tidyverse, but it offers handy functions for data cleaning and works well within data pipelines that use `|>`. ↩

2. You can override the default of 1000 with the `guess_max` argument. ↩