r4ds-ch7

Ch 7 Data import

7.1 Introduction

7.1.1 Prerequisites

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.2     v tibble    3.2.1
## v lubridate 1.9.4     v tidyr     1.3.1
## v purrr     1.0.4
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

7.2 Reading data from a file

First argument to read a comma-separated values file, the most common rectangular data file type, is the filepath

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

7.2.1 Practical advice

After reading in data, first step is to transform it to make it work with rest of your analysis

```
students
```

```
## # A tibble: 6 x 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 favourite.food there is "N/A" which should be a real NA, we can do this by adding "N/A" to the na argument in read_csv(), which by default is only empty strings (" ")

```
students <- read.csv("https://pos.it/r4ds-students-csv", na = c("N/A", ""))
students
```

```
##   Student.ID        Full.Name     favourite.food            mealPlan  AGE
## 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
```

Initially, Student ID and Full Name columns were surrounded by backticks (in console), because they contain spaces, which breaks R's usual variable name rules (they are non-syntactic names) To refer to these variables, need to use backticks, `

```
# students |>
#   rename(
#     student_id = `Student ID`,
#     full_name = `Full Name`
#   )
```

An alternative approach is to use janitor::clean_names() to turn them all into snakecase at once

```
students |> janitor::clean_names()
```

```
##   student_id        full_name     favourite_food            meal_plan  age
## 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 is to consider variable value types meal_plan is a categorical variable with known set of possible values, which in R should be represented as a factor

```
students |>
  janitor::clean_names() |>
  mutate(meal_plan = factor(meal_plan))
```

```
##   student_id        full_name     favourite_food           meal_plan  age
## 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
```

The values under meal_plan stay the same but the variable type changed from character () to factor ()

Before analysis, will also want to fix age column, which is currently a character variable because one of the observations is "five" instead of 5

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

```
##   student_id        full_name     favourite_food           meal_plan age
## 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
```

if_else() has three arguments, first is test (logical vector), result will be value of second argument (yes) when test is true, and the third argument (no) when test is false

7.2.2 Other arguments

read_csv() can read text strings you created and formatted like a CSV file

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

```
## Rows: 2 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): a, b, c
##
## 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.
```

```
## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

Use skip = n to skip first n lines (for metadeta) or use comment = "#" to drop all lines that start with "#"

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

```
## Rows: 1 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): x, y, z
##
## 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.

## # A tibble: 1 x 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 = "#"
)
```

```
## Rows: 1 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): x, y, z
##
## 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.

## # A tibble: 1 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
```

Data might not have column names, use col_names = FALSE to tell read_csv() to not treat first line as header and instead label sequentially from X1 to Xn

```
read_csv(
  "1,2,3
  4,5,6",
  col_names = FALSE
)
```

```
## Rows: 2 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): X1, X2, X3
##
## 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.
```

```
## # A tibble: 2 x 3
##      X1    X2    X3
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

Can also pass col_names a character vector to be used as column names

```
read_csv(
  "1,2,3
  4,5,6",
  col_names = c("x", "y", "z")
)
```

```
## Rows: 2 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): x, y, z
##
## 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.
```

```
## # A tibble: 2 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

7.2.3 Other file types

- read_csv_2() reads semicolon-separated files, common in countries that use "," as decimal marker
- read_tsv() reads tab-separated files
- read_delim() reads files with any delimiter, attempting to automatically guess it if you don't specify
- read_fwf() reads fixed-width files, can specify fields by their widths with fwf_widths() or by positions with fwf_positions()
- read_table() reads a common variation of fixed-width files where columns are separated by whitespace
- 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 "|"? read_delim(), specifically, read_delim(data, delim = "|")

2. Apart from file, skip, comment, what other arguments do read_csv() and read_tsv() have in common? col_names, col_types, col_select, id (name of column to store the filepath), locale (controls things like time zone, encoding, decimal mark), trim_ws (trim whitespace), n_max (max # of lines read), guess_max (max # of lines to use for guessing column types), name_repair (handling col names), num_threads (# of processing threads for parsing/lazy reading), progress (progress bar), show_col_types, skip_empty_rows, lazy

3. What are the most important arguments to read_fwf()? col_positions, as created by fwf_empty(), fwf_widths(), fwf_positions(), or fwf_cols() which lets you supply named arguments of paired start/end positions or column widths

4. Sometimes strings in a CSV file contain commas, to prevent them causing problems they need to be surrounded by a quoting character like " or ', by default read_csv() assumes it will be " To read the following text into a data frame, what argument to read_csv() do you need to specify?

```
# read_csv(data, quote = "\'")
```

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")  # No third header
```

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

```
## Rows: 2 Columns: 2
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (1): a
## num (1): b
##
## 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.
```

```
## # A tibble: 2 x 2
##       a     b
##   <dbl> <dbl>
## 1     1    23
## 2     4    56
```

```
read_csv("a,b,c\n1,2\n1,2,3,4")  # 2 values row 2, 4 in row 3
```

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

```
## Rows: 2 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (2): a, b
## num (1): c
##
## 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.

## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2    NA
## 2     1     2    34
```

```
read_csv("a,b\n\"1")  # Extra quote before 1
```

```
## Rows: 0 Columns: 2
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): a, b
##
## 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.

## # A tibble: 0 x 2
## # i 2 variables: a <chr>, b <chr>
```

```
read_csv("a,b\n1,2\na,b")  # a,b will turn columns into characters
```

```
## Rows: 2 Columns: 2
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): a, b
##
## 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.

## # A tibble: 2 x 2
##   a     b
##   <chr> <chr>
## 1 1     2
## 2 a     b
```

```
read_csv("a;b\n1;3")  # Need to use read_csv2()
```

```
## Rows: 1 Columns: 1
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): a;b
##
## 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.
```

```
## # A tibble: 1 x 1
##   `a;b`
##   <chr>
## 1 1;3
```

Since no third header was used, 2,3 and 5,6 get squeezed into column b There is a NA in column c in row 1 because only 2 values in that row, and 3,4 get squeezed into column c in row 2 The extra quote means nothing is read Values being a,b in row 2 means 1,2 get coerced to character strings Since "," still used as separator, a;b and 1;3 get squeezed into one value

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

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

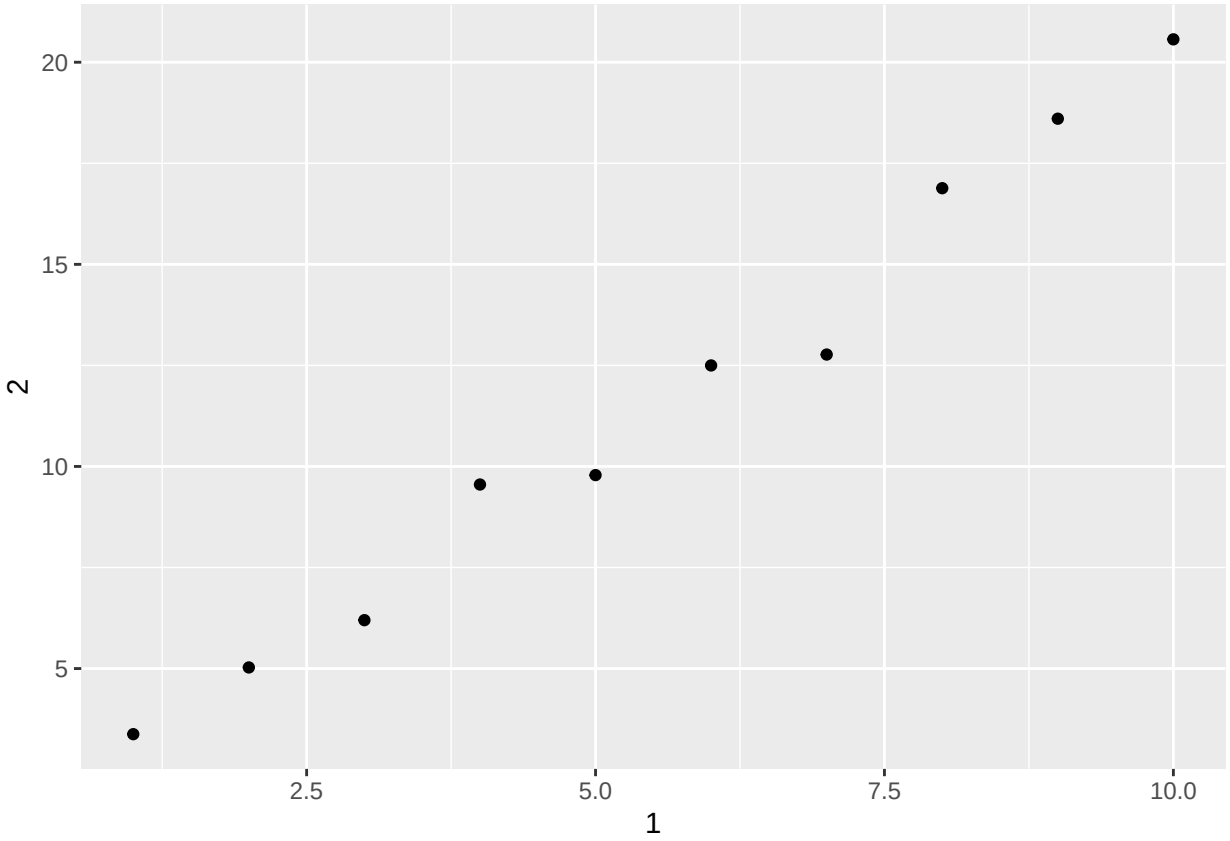
a. Extracting the the variable called 1

```
annoying$`1`
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

b. Plotting a scatterplot of 1 vs 2

```
annoying |>
  ggplot(aes(x = `1`, y = `2`)) +
  geom_point()
```



c. Creating a new column 3, which is 2 divided by 1

```
annoying <- annoying |>
  mutate(
    `3` = `2` / `1`
  )
```

d. Renaming the columns to one, two, and three

```
annoying |>
  rename(
    one = `1`,
    two = `2`,
    three = `3`
  )
```

```
## # A tibble: 10 x 3
##      one   two three
##    <int> <dbl> <dbl>
##  1     1  3.38  3.38
##  2     2  5.03  2.51
##  3     3  6.20  2.07
##  4     4  9.55  2.39
##  5     5  9.79  1.96
```

```
##  6     6 12.5   2.08
##  7     7 12.8   1.82
##  8     8 16.9   2.11
##  9     9 18.6   2.07
## 10    10 20.6   2.06
```

7.3 Controlling column types

7.3.1 Guessing types

CSV file doesn't contain any info about the type of each variable, so readr guesses by using a heuristic to figure out column types For each column, it pulls the values of 1000^2 (can override with guess_max) rows spaced evenly from first row to last, ignoring missing values, then works through following questions - Does it contain F, T, FALSE, or TRUE (ignoring case)? If so, logical - Does it contain only numbers? If so, it's a number - Does it match the ISO8601 standard? If so, it's a date or date-time - Otherwise, must be a string

Here's an example of that behavior

```
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
")
```

```
## Rows: 3 Columns: 4
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (1): string
## dbl  (1): numeric
## lgl  (1): logical
## date (1): date
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 3 x 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
```

Heuristic works well if have clean dataset, but often will encounter weird/beautiful failures

7.3.2 Missing values, column types, and problems

Most common way column detection fails is that a column contains unexpected values, and you get a character column instead of more specific type, one of most common causes is missing value recorded as something other than NA

Simple example

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

If read without any additional arguments, x becomes character column

```
read_csv(simple_csv)
```

```
## Rows: 4 Columns: 1
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): x
##
## 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.
```

```
## # A tibble: 4 x 1
##   x
##   <chr>
## 1 10
## 2 .
## 3 20
## 4 30
```

In this case, can see . is missing value, but what if there are thousands of rows and just a few .s

One approach is to tell readr that x is a numeric column, then see where it fails Can do this with col_types argument which takes named list where names match 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 a problem and says to use problems() to find out more

```
problems(df)
```

```
## # A tibble: 1 x 5
##     row   col expected actual file
##   <int> <int> <chr>    <chr>  <chr>
## 1     3     1 a double .      /private/var/folders/qy/5pm1cj9n2tg60w8wm_b_xx180~
```

This suggests that the dataset uses . for missing values

Now we can set na = ".", and the automatic guessing will work

```
read_csv(simple_csv, na = ".")
```

```
## Rows: 4 Columns: 1
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (1): x
##
## 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.

## # A tibble: 4 x 1
##       x
##   <dbl>
## 1    10
## 2    NA
## 3    20
## 4    30
```

7.3.3 Column types

readr provides 9 column types: - col_logical() and col_double() read logicals and real numbers, relatively rarely needed (except as above), since readr will usually guess for you - col_integer() reads integers, seldom distinguish them from doubles because functionally equivalent but reading integers can be sometimes useful since they occupy half the memory of doubles - col_character() reads strings, useful to specify explicitly when have column that is a numeric identifier, like a series of digits that identifies an object but doesn't make sense to apply arithmetic to, such as phone numbers - col_factor(), col_date(), and col_datetime() create factors, dates, and date-times respectively - col_number() is permissive numeric parser that will ignore non-numeric components, particularly useful for currencies - col_skip() skips a column so not included in result, useful for speeding up reading data from large CSV file if only want to use some of the columns

Also possible to override 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 x 3
##   x     y     z
##   <chr> <chr> <chr>
## 1 1     2     3
```

Another useful helper is cols_only() which reads in only columns you specify

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

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

7.4 Reading data from multiple files

Sometimes might have data for multiple months, with each month as a separate file With read_csv() can read these in at once and stack them into a single data frame

```
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")
```

```
## Rows: 19 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): month
## dbl (4): year, brand, item, n
##
## 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.
```

```
## # A tibble: 19 x 6
##    file                          month     year brand  item     n
##    <chr>                         <chr>    <dbl> <dbl> <dbl> <dbl>
##  1 https://pos.it/r4ds-01-sales January   2019     1  1234     3
##  2 https://pos.it/r4ds-01-sales January   2019     1  8721     9
##  3 https://pos.it/r4ds-01-sales January   2019     1  1822     2
##  4 https://pos.it/r4ds-01-sales January   2019     2  3333     1
##  5 https://pos.it/r4ds-01-sales January   2019     2  2156     9
##  6 https://pos.it/r4ds-01-sales January   2019     2  3987     6
##  7 https://pos.it/r4ds-01-sales January   2019     2  3827     6
##  8 https://pos.it/r4ds-02-sales February  2019     1  1234     8
##  9 https://pos.it/r4ds-02-sales February  2019     1  8721     2
## 10 https://pos.it/r4ds-02-sales February  2019     1  1822     3
## 11 https://pos.it/r4ds-02-sales February  2019     2  3333     1
## 12 https://pos.it/r4ds-02-sales February  2019     2  2156     3
## 13 https://pos.it/r4ds-02-sales February  2019     2  3987     6
## 14 https://pos.it/r4ds-03-sales March     2019     1  1234     3
## 15 https://pos.it/r4ds-03-sales March     2019     1  3627     1
## 16 https://pos.it/r4ds-03-sales March     2019     1  8820     3
## 17 https://pos.it/r4ds-03-sales March     2019     2  7253     1
## 18 https://pos.it/r4ds-03-sales March     2019     2  8766     3
## 19 https://pos.it/r4ds-03-sales March     2019     2  8288     6
```

The id argument adds a new column called file that identifies the file where the data came from, this helps trace back to original source when files don't have an identifying column

If have many files to read in, can be cumbersome to write out their names as a list Can instead use base list.files() function to find files by matching a pattern in the file names

```
# sales_files <- list.files("data", pattern = "sales\\.csv$", full.names = TRUE)
```

7.5 Writing to a file

readr comes with two functions: write_csv() and write_tsv() Most important arguments are x (data frame to save) and file (where to save) 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 file back in, note that the variable type information that was set up is lost because when save a CSV starting over with plain text file again

```
students
```

```
##   student_id        full_name     favourite_food           meal_plan age
## 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")
```

```
## Rows: 6 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): full_name, favourite_food, meal_plan
## dbl (2): student_id, age
##
## 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.
```

```
## # A tibble: 6 x 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, need to recreate column specification every time you load in, there are two main alternatives:

1. write_rds() and read_rds() are uniform wrappers around base functions readRDS() and saveRDS(), they store data in R's custom binary format RDS which means that when you reload the object, you load the exact same R object you stored

```
write_rds(students, "students.rds")
read_rds("students.rds")
```

```
##   student_id        full_name     favourite_food           meal_plan age
## 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 lets you read and write parquet files, a fast binary file format that can be shared across programming languages

```
library(arrow)
```

```
##
## Attaching package: 'arrow'
```

```
## The following object is masked from 'package:lubridate':
##
##     duration
```

```
## The following object is masked from 'package:utils':
##
##     timestamp
```

```
write_parquet(students, "students.parquet")
read_parquet("students.parquet")
```

```
## # A tibble: 6 x 5
##   student_id full_name        favourite_food     meal_plan             age
##        <int> <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 faster than RDS and is usable outside of R, but requires arrow package

7.6 Data entry

Sometimes need to assemble a tibble "by hand" doing a little data entry in R script There are two useful functions to do this which differ in whether you layout tibble by columns or 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 x 3
##       x y         z
##   <dbl> <chr> <dbl>
## 1     1 h      0.08
## 2     2 m      0.83
## 3     5 g      0.6
```

This can make it hard to see how rows are related, so an alternative is tribble()

tribble() is short for transposed tibble, which lets you lay out your data row by row, and it is customized for data entry in code, meaning column headings start with ~ and entries are separated by columns

This lets you 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 x 3
##       x y         z
##   <dbl> <chr> <dbl>
## 1     1 h      0.08
## 2     2 m      0.83
## 3     5 g      0.6
```