10 Tibbles

10.1 Introduction

Throughout this book we work with "tibbles" instead of R's traditional `data.frame`. Tibbles *are* data frames, but they tweak some older behaviours to make life a little easier. R is an old language, and some things that were useful 10 or 20 years ago now get in your way. It's difficult to change base R without breaking existing code, so most innovation occurs in packages. Here we will describe the **tibble** package, which provides opinionated data frames that make working in the tidyverse a little easier. In most places, I'll use the term tibble and data frame interchangeably; when I want to draw particular attention to R's built-in data frame, I'll call them `data.frame`s.

If this chapter leaves you wanting to learn more about tibbles, you might enjoy `vignette("tibble")`.

10.1.1 Prerequisites

In this chapter we'll explore the **tibble** package, part of the core tidyverse.

```
library(tidyverse)
```

Copy

10.2 Creating tibbles

Almost all of the functions that you'll use in this book produce tibbles, as tibbles are one of the unifying features of the tidyverse. Most other R packages use regular data frames, so you might want to coerce a data frame to a tibble. You can do that with `as_tibble()`:

```
as_tibble(iris)
#> # A tibble: 150 x 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>
#> 1          5.1         3.5          1.4         0.2 setosa
#> 2          4.9         3            1.4         0.2 setosa
#> 3          4.7         3.2          1.3         0.2 setosa
#> 4          4.6         3.1          1.5         0.2 setosa
#> 5          5           3.6          1.4         0.2 setosa
#> 6          5.4         3.9          1.7         0.4 setosa
#> # … with 144 more rows
```

Copy

You can create a new tibble from individual vectors with `tibble()`. `tibble()` will automatically recycle inputs of length 1, and allows you to refer to variables that you just created, as shown below.

```
tibble(
  x = 1:5,
  y = 1,
  z = x ^ 2 + y
)
#> # A tibble: 5 x 3
#>       x     y     z
#>   <int> <dbl> <dbl>
#> 1     1     1     2
#> 2     2     1     5
#> 3     3     1    10
#> 4     4     1    17
#> 5     5     1    26
```

Copy

If you're already familiar with `data.frame()`, note that `tibble()` does much less: it never changes the type of the inputs (e.g. it never converts strings to factors!), it never changes the names of variables, and it never creates row names.

It's possible for a tibble to have column names that are not valid R variable names, aka **non-syntactic** names. For example, they might

not start with a letter, or they might contain unusual characters like a space. To refer to these variables, you need to surround them with backticks, `` ` `` :

```
tb <- tibble(
  `:)` = "smile",
  ` ` = "space",
  `2000` = "number"
)
tb
#> # A tibble: 1 x 3
#>   `:)`  ` `   `2000`
#>   <chr> <chr> <chr>
#> 1 smile space number
```

You'll also need the backticks when working with these variables in other packages, like ggplot2, dplyr, and tidyr.

Another way to create a tibble is with `tribble()`, short for **tr**ansposed tibble. `tribble()` is customised for data entry in code: column headings are defined by formulas (i.e. they start with `~`), and entries are separated by commas. This makes it possible to lay out small amounts of data in easy to read form.

```
tribble(
  ~x, ~y, ~z,
  #--|--|----
  "a", 2, 3.6,
  "b", 1, 8.5
)
#> # A tibble: 2 x 3
#>   x         y     z
#>   <chr> <dbl> <dbl>
#> 1 a         2   3.6
#> 2 b         1   8.5
```

I often add a comment (the line starting with `#`), to make it really clear where the header is.

10.3 Tibbles vs. data.frame

There are two main differences in the usage of a tibble vs. a classic `data.frame`: printing and subsetting.

10.3.1 Printing

Tibbles have a refined print method that shows only the first 10 rows, and all the columns that fit on screen. This makes it much easier to work with large data. In addition to its name, each column reports its type, a nice feature borrowed from `str()`:

```
tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
#> # A tibble: 1,000 x 5
#>   a                   b              c     d e
#>   <dttm>              <date>     <int> <dbl> <chr>
#> 1 2020-10-09 13:55:17 2020-10-16     1 0.368 n
#> 2 2020-10-10 08:00:26 2020-10-21     2 0.612 l
#> 3 2020-10-10 02:24:06 2020-10-31     3 0.415 p
#> 4 2020-10-09 15:45:23 2020-10-30     4 0.212 m
#> 5 2020-10-09 12:09:39 2020-10-27     5 0.733 i
#> 6 2020-10-09 23:10:37 2020-10-23     6 0.460 n
#> # … with 994 more rows
```

Tibbles are designed so that you don't accidentally overwhelm your console when you print large data frames. But sometimes you need more output than the default display. There are a few options that can help.

First, you can explicitly `print()` the data frame and control the number of rows (`n`) and the `width` of the display. `width = Inf` will display all columns:

```
nycflights13::flights %>%
  print(n = 10, width = Inf)
```

Copy

You can also control the default print behaviour by setting options:

- `options(tibble.print_max = n, tibble.print_min = m)` : if more than `n` rows, print only `m` rows. Use `options(tibble.print_min = Inf)` to always show all rows.
- Use `options(tibble.width = Inf)` to always print all columns, regardless of the width of the screen.

You can see a complete list of options by looking at the package help with `package?tibble` .

A final option is to use RStudio's built-in data viewer to get a scrollable view of the complete dataset. This is also often useful at the end of a long chain of manipulations.

```
nycflights13::flights %>%
  View()
```

Copy

10.3.2 Subsetting

So far all the tools you've learned have worked with complete data frames. If you want to pull out a single variable, you need some new tools, `$` and `[[` . `[[` can extract by name or position; `$` only extracts by name but is a little less typing.

```
df <- tibble(
  x = runif(5),
  y = rnorm(5)
)

# Extract by name
df$x
#> [1] 0.73296674 0.23436542 0.66035540 0.03285612 0.46049161
df[["x"]]
#> [1] 0.73296674 0.23436542 0.66035540 0.03285612 0.46049161

# Extract by position
df[[1]]
#> [1] 0.73296674 0.23436542 0.66035540 0.03285612 0.46049161
```

Copy

To use these in a pipe, you'll need to use the special placeholder `.` :

```
df %>% .$x
#> [1] 0.73296674 0.23436542 0.66035540 0.03285612 0.46049161
df %>% .[["x"]]
#> [1] 0.73296674 0.23436542 0.66035540 0.03285612 0.46049161
```

Copy

Compared to a `data.frame` , tibbles are more strict: they never do partial matching, and they will generate a warning if the column you are trying to access does not exist.

10.4 Interacting with older code

Some older functions don't work with tibbles. If you encounter one of these functions, use `as.data.frame()` to turn a tibble back to a `data.frame` :

```
class(as.data.frame(tb))
#> [1] "data.frame"
```

Copy

The main reason that some older functions don't work with tibble is the `[` function. We don't use `[` much in this book because `dplyr::filter()` and `dplyr::select()` allow you to solve the same problems with clearer code (but you will learn a little about it in vector subsetting). With base R data frames, `[` sometimes returns a data frame, and sometimes returns a vector. With tibbles, `[` always

returns another tibble.

10.5 Exercises

1. How can you tell if an object is a tibble? (Hint: try printing `mtcars`, which is a regular data frame).

2. Compare and contrast the following operations on a `data.frame` and equivalent tibble. What is different? Why might the default data frame behaviours cause you frustration?

```
df <- data.frame(abc = 1, xyz = "a")
df$x
df[, "xyz"]
df[, c("abc", "xyz")]
```

Copy

3. If you have the name of a variable stored in an object, e.g. `var <- "mpg"`, how can you extract the reference variable from a tibble?

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

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

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

Copy

5. What does `tibble::enframe()` do? When might you use it?

6. What option controls how many additional column names are printed at the footer of a tibble?