R: Tidy Data & Data Import

Sakol Suethanapornkul

10 December 2020

Preamble

Data come in various formats. Some of the most common ones are:

- spreadsheet files (.xlsx or .csv);
- plain text files (.txt); or
- SPSS files (.sav).

Thus far, we have worked with data that are part of R packages. But in real-world contexts, we often work with "our" data that are in one of the above formats. How do we get those data into R?

Preamble

In this part, we will learn how to read in rectangular files (i.e., ones that consist of rows and columns) into R with `readr`

`readr` is part of `tidyverse`. So, before we go any further, let's call the package.

```
library(tidyverse)
```

Part I: Pro Tip #2

It is important that our data be tidy. This ensures that we can apply transformations to our data using tools in the `tidyverse` package.

EXERCISE: What do tidy data have in common?

Part I: Pro Tip #2

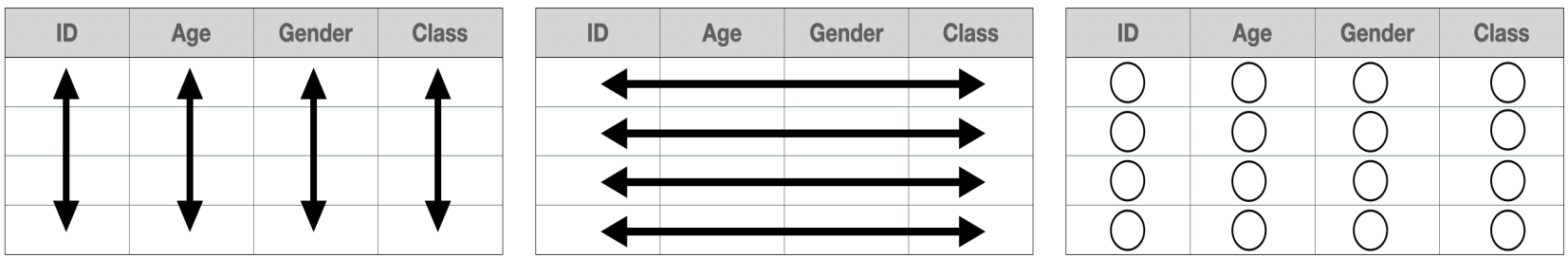
Principles of tidy data:

- Each variable must have its own column;
- Each observation must have its own row; and
- Each value must have its own cell.

Part I: Pro Tip #2

Principles of tidy data:

- Each variable must have its own column;
- Each observation must have its own row; and
- Each value must have its own cell.



Credit: R for Data Science

Part I: Pro Tip #2

EXERCISE: Look carefully at each spreadsheet. Is it tidy? If not, what can we do to fix it?

Part I: Pro Tip #2

EXERCISE: Look carefully at each spreadsheet. Is it tidy? If not, what can we do to fix it?

[1]

A	B	C	D	E	F	G
Test	Group:Control	Group:Experiment		Test	Group:Control	Group:Experiment
Pre-test	15	16		Post-test	18	20
	17	17			19	20
	13	14			15	19
	21	21			23	28
	22	20			22	26
	16	16			17	21
	25	26			27	26
	21	21			23	25
	22	22			21	23
	18	17			20	23
	15	16			18	16
	16	16			18	16
	16	14			17	17
	21	23			21	23
	20	22			24	23
	19	19			20	20

Part I: Pro Tip #2

EXERCISE: Look carefully at each spreadsheet. Is it tidy? If not, what can we do to fix it?

[2]

A	B	C	D	E	F	G
ID	Motivation_Set1	Motivation_Set2	Gender	Rating	Enjoyment	Grade
1	3.5	4.1	M	7	5	4
2	2.5	4.2	M	7	4	3
3	2	3.7	F	6	2	1
4	2.7	4.2	F	8	5	2
5	2.8	5	T	4	3	3
6	4	4	F	9	4	3
7	3.8	4.2	T	6	4	2
8	2.9	3.2	M	5	5	2
9	4	5	T	7	3	1
10	3.6	3.7	M	7	3	4

Part I: Pro Tip #2

EXERCISE: Look carefully at each spreadsheet. Is it tidy? If not, what can we do to fix it?

[3]

A	B	C	D	E
ID	Type	Value	Total	Prop
1	Pre	5	20	5/20
1	Post	3	20	3/20
1	Motivation	4	5	4/5
2	Pre	6	20	6/20
2	Post	7	20	7/20
2	Motivation	4	5	4/5
3	Pre	8	20	8/20
3	Post	8	20	8/20
3	Motivation	3	5	3/5
4	Pre	11	20	11/20
4	Post	7	20	7/20
4	Motivation	4	5	4/5

Part I: Pro Tip #2

Tidy data aren't just about R (though they help make vectorized operations in R shine).

So, make sure your data are tidy everywhere (e.g., in Excel spreadsheet etc.)

Part II: `read_csv()`

Now that we know how tidy data look, let's work our way through getting these data into R!

We will use `read_csv()` to practice data import. Once you master it, you can apply the knowledge to other kinds of files.

Part II: `read_csv()`

`read_csv()` works with comma separated files (i.e., the ones that end with .csv).

- The first thing we need is a file name:

```
income <- read_csv(file = "Income.csv")
```

Part II: `read_csv()`

`read_csv()` works with comma separated files (i.e., the ones that end with .csv).

- The first thing we need is a file name:

```
income <- read_csv(file = "Income.csv")
```

- But if that file is in a different folder:

```
income <- read_csv(file = "./Data/Income.csv")
```

NOTE: Use `getwd()` to see your current directory.

Part II: `read_csv()`

`getwd()` returns the absolute path to our current directory, which we can use in `read_csv()`

But this is generally not recommended. Assuming that you always have a folder called "Data" in any of your R projects, it's better to use a relative path when you read in a file:

```
income <- read_csv(file = "./Data/Income.csv")
```

NOTE: Use forward slashes (/) for path, regardless of your computer's OS (Mac or Windows).

Part II: `read_csv()`

`read_csv()` works with comma separated files (i.e., the ones that end with .csv).

- The second thing we need is column names:

```
income <- read_csv(file = "./Data/Income.csv",
                   col_names = TRUE )    #default
```

Part II: read_csv()

read_csv() works with comma separated files (i.e., the ones that end with .csv).

- The second thing we need is column names:

```
income <- read_csv(file = "./Data/Income.csv",
                   col_names = TRUE )    #default
```

- If column name is missing, use FALSE:

```
income <- read_csv(file = "./Data/Income.csv",
                   col_names = FALSE )
```

EXERCISE: what does col_names = FALSE do?

Part II: read_csv()

Alternatively, col_names accepts a character vector, which will be used as column names:

```
#DO NOT RUN
income <- read_csv(file = "./Data/Income.csv",
                   col_names = c("year",
                                 "option",
                                 "status")
                   )
```

Part II: `read_csv()`

`read_csv()` works with comma separated files (i.e., the ones that end with .csv).

- The third thing we need is NAs:

```
income <- read_csv(file = "./Data/Income.csv",
                   na = c("", "NA") ) #default
```

Part II: read_csv()

read_csv() works with comma separated files (i.e., the ones that end with .csv).

- The third thing we need is NAs:

```
income <- read_csv(file = "./Data/Income.csv",
                   na   = c("", "NA") ) #default
```

- If there's no missing values, you can omit this argument or:

```
income <- read_csv(file = "./Data/Income.csv",
                   na   = character() )
```

Part II: read_csv()

But if your missing value is coded as something else (e.g., 999), pass that number to na as a character string:

```
#EXAMPLE
income <- read_csv(file = "./Data/Income.csv",
                   na   = c("618", "715", "775") )
```

Part II: `read_csv()`

If you're curious about how `read_csv()` works behind the scenes, check out this site. This understanding will help us with the next few slides.

Part II: `read_csv()`

Thus far, we have ignored the message in red that `read_csv()` prints out to the console. But now we are ready to talk about it.

Part II: `read_csv()`

Thus far, we have ignored the message in red that `read_csv()` prints out to the console. But now we are ready to talk about it.

`read_csv()` (and other functions in `readr`) automatically guesses the type of each column when it reads in files.

Part II: `read_csv()`

`readr` uses a number of heuristics to guess what each column is. But these heuristics can fail with a large file (1000 rows+).

For example,

```
challenge <- read_csv(readr_example("challenge.csv") )
```

NOTE: `readr_example()` finds the path to one of the files included with the package.

Part II: `read_csv()`

If we look at the message, we see that the `x` column is parsed as a double vector, and the `y` column, a logical vector.

EXERCISE: View the data frame. What do you see? What is going on with `y`?

Part II: `read_csv()`

If we look at the message, we see that the `x` column is parsed as a double vector, and the `y` column, a logical vector.

Looking back at the console, we see warning which tells us what went wrong.

Part II: read_csv()

So, to fix this problem, you can provide specific column type to read_csv() with col_types:

```
challenge <- read_csv(readr_example("challenge.csv")
                      col_types = cols(
                        x = col_double(),
                        y = col_date()
                        #from y = col_logical()
                        )
                      )
```

Part II: `read_csv()`

We can provide different types of columns to `col_types`.

For instance, if we want to be explicit about column types in our `income` data set, we can treat country names as characters, while leaving all the other columns as doubles.

```
income <- read_csv(file = "./Data/Income.csv",
                   col_types = cols(
                     .default = col_double(),
                     country  = col_character()
                     )
                   )
```

Part II: read_csv()

EXERCISE: What will happen if we run this code?

```
income <- read_csv(file = "./Data/Income.csv",
                   col_types = cols(
                     .default = col_double()
                     )
                   )
```

Part III: Extra Bits

How to. . .

- read csv files from the Internet (e.g., this site)

Part III: Extra Bits

How to. . .

- read csv files from the Internet (e.g., this site)

```
ent_survey <- read_csv(file = "html-link",
                       col_types = cols(
                         .default = col_character()
                         )
                       )
```

Part III: Extra Bits

How to. . .

- read csv files from the Internet (e.g., this site)
- import tab-delimited files (e.g., this site)

Part III: Extra Bits

How to. . .

- read csv files from the Internet (e.g., this site)
- import tab-delimited files (e.g., this site)

```
mri <- read_delim(file  = "html-link",
                  delim = "\t")

#or

mri <- read_table(file  = "html-link")
```

Part IV: `write_csv()`

After we finish wrangling with data frames, we may want to write them back to disk. `readr` offers a useful function `write_csv()` for that purpose!

Part IV: `write_csv()`

After we finish wrangling with data frames, we may want to write them back to disk. `readr` offers a useful function `write_csv()` for that purpose!

- The first argument is a named data frame
- The second argument is a file name

```
write_csv(income, "GDP.csv")
```

Check out this page for more information.

Part IV: `write_csv()`

To put a newly created file in a folder different from a current working space, use:

```
write_csv(income, "./Data/GDP.csv")
```

For other types of files, you can use `write_delim()`:

```
write_delim(income, "GDP.csv",
            delim = "\t")
```

Wrap-up

In this part, we have discussed the following functions:

- `read_csv()`;
- `read_delim()`; and
- `write_csv()`

We have also learned how to:

- specify column types;
- provide column names; and
- read in files from the Internet