Managing Data in R

Data Wrangling in R

Glenn Williams

University of Sunderland

2021-10-27 (updated: 2021-10-27)

Loading our Cleaned Data

Let's load up a data set.

Remember, `read_csv()` takes one argument: where is your data!

```
data_a <- read_csv(here("data", "factorial_data_a.csv"))
```

```
## Parsed with column specification:
## cols(
##   row = col_double(),
##   subj_id = col_character(),
##   list_id = col_double(),
##   item_id = col_double(),
##   A = col_character(),
##   B = col_character(),
##   Y = col_double()
## )
```

The rest of our data is in a second table. Let's load this in too.

```
data_b <- read_csv(here("data", "factorial_data_b.csv"))
```

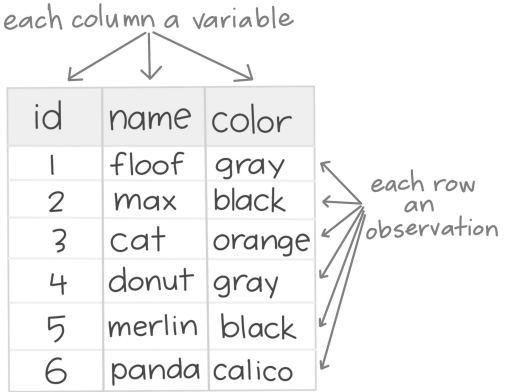
Tidy Data

Our data is in a tidy/long format.

> "TIDY DATA is a standard way of mapping the meaning of a dataset to its structure."
>
> –HADLEY WICKHAM

In tidy data:

- each variable forms a column
- each observation forms a row
- each cell is a single measurement

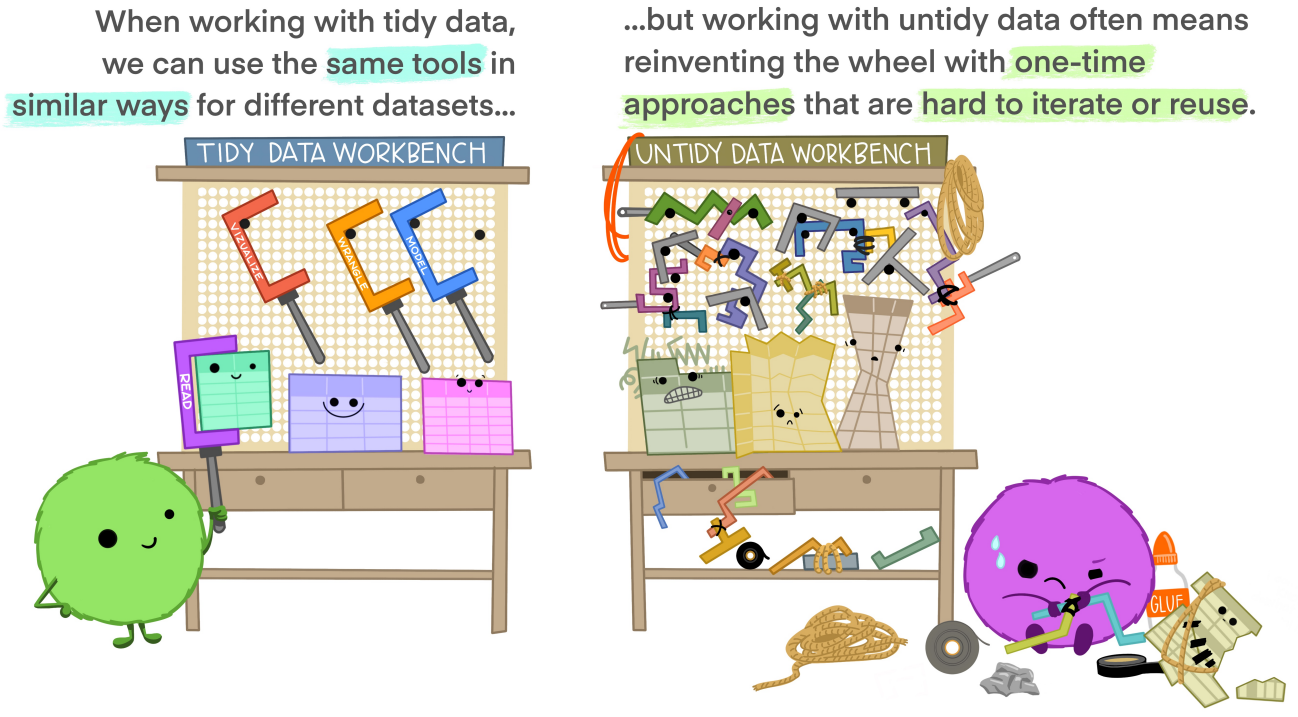


Wickham, H. (2014). Tidy Data. Journal of Statistical Software *59* (10). DOI: 10.18637/jss.v059.i10

Illustrations from the Openscapes blog Tidy Data for reproducibility, efficiency, and collaboration by Julia Lowndes and Allison Horst

Why Use Tidy Data?

This makes cleaning, summarising, plotting, and modelling easier!



Illustrations from the Openscapes blog Tidy Data for reproducibility, efficiency, and collaboration by Julia Lowndes and Allison Horst

Common Data Processing Tasks

The most common data manipulation problems can be handled effectively by `dplyr`, a package within the tidyverse:

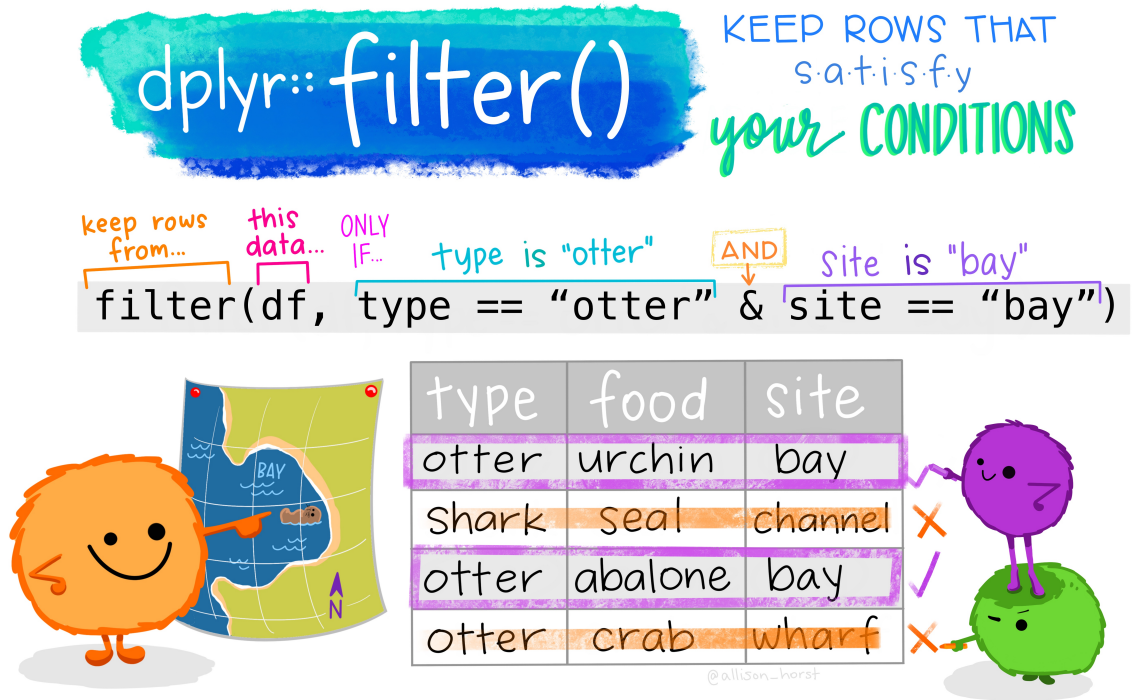
One Table:

- `filter()`: filters our data to keep observations that only meet given conditions.
- `select()`: selects a subset of columns in our table.
- `mutate()`: changes our data in some way by the instructions you provide.

Multiple Tables:

- `bind_rows()`: adds rows from one data frame to another (if they have the same columns).
- `bind_cols()`: adds columns to one data frame from another (if they have the same number of rows).

Filter



Artwork by @allison_horst

Filter

Let's say we want to keep only observations from our first participant. This might be useful for exploring data individually.

```
filter(data_a, subj_id == 4)
```

```
## # A tibble: 40 x 7
##     row subj_id list_id item_id A     B          Y
##   <dbl> <chr>     <dbl>   <dbl> <chr> <chr> <dbl>
## 1   121 4             4       1 A2    B2     222.
## 2   122 4             4       2 A2    B2     242.
## 3   123 4             4       3 A2    B2     251.
## 4   124 4             4       4 A2    B2     200.
## # … with 36 more rows
```

We have only 40 observations now, all from subject 4.

Combining Filter Conditions

Maybe we want to check if subject 4 has any missing data in the variable Y. How might we do this?

We could apply two separate filters, or combine our filter conditions using logical operations.

```
filter(data_a, subj_id == 4 & is.na(Y))
```

```
## # A tibble: 1 x 7
##     row subj_id list_id item_id A     B          Y
##   <dbl> <chr>     <dbl>   <dbl> <chr> <chr> <dbl>
## 1   147 4             4      27 A1    B2       NA
```

Note that **NAs are special in R**, so we have to use is.na() which checks for any NA values explicitly, rather than Y == NA.

It looks like subject 4 has 1 missing response!

Select

Select works by defining the names of **columns you'd like to keep**, and in what order!

Artwork by @allison_horst

Select

Imagine we only want to **keep the subject_id** column.

```
select(data_a, subj_id)
```

```
## # A tibble: 1,040 x 1
##   subj_id
##   <chr>
## 1 one
## 2 one
## 3 one
## 4 one
## # … with 1,036 more rows
```

Select

Imagine we want to **remove the list_id column**. We simply put a minus before the column name.

```
select(data_a, -list_id)
```

```
## # A tibble: 1,040 x 6
##     row subj_id item_id A     B          Y
##   <dbl> <chr>      <dbl> <chr> <chr> <dbl>
## 1     1 one            1 A2    B2     234.
## 2     2 one            2 A2    B2     295.
## 3     3 one            3 A2    B2     285.
## 4     4 one            4 A2    B2     258.
## # … with 1,036 more rows
```

Select

Maybe we'd like to **reorganise our columns so columns A and B come first and everything else after**. Select A and B, and then everything else using `everything()`; a dplyr helper function.

```
select(data_a, A, B, everything())
```

```
## # A tibble: 1,040 x 7
##   A     B       row subj_id list_id item_id     Y
##   <chr> <chr> <dbl> <chr>     <dbl>   <dbl> <dbl>
## 1 A2    B2        1 one           4       1  234.
## 2 A2    B2        2 one           4       2  295.
## 3 A2    B2        3 one           4       3  285.
## 4 A2    B2        4 one           4       4  258.
## # … with 1,036 more rows
```

Muatate

What if we want to **change our data**? Use `mutate()`.

Mutate

With mutate we can **change the data within columns**, or **create new columns** of data entirely.

Let's create a new column. This will combine the text in columns A and B. We will use `paste()` to do this. The argument `sep = "+"` will add a plus between the entries.

```
mutate(data_a, AB = paste(A, B, sep = "+"))
```

```
## # A tibble: 1,040 x 8
##     row subj_id list_id item_id A     B          Y AB
##   <dbl> <chr>     <dbl>   <dbl> <chr> <chr> <dbl> <chr>
## 1     1 one           4       1 A2    B2     234. A2+B2
## 2     2 one           4       2 A2    B2     295. A2+B2
## 3     3 one           4       3 A2    B2     285. A2+B2
## 4     4 one           4       4 A2    B2     258. A2+B2
## # … with 1,036 more rows
```

Mutate

Perhaps instead we want to **change the data in a column in place**? Let's get the natural logarithm on the scores in Y.

```
mutate(data_a, Y = log(Y))
```

```
## # A tibble: 1,040 x 7
##     row subj_id list_id item_id A     B          Y
##   <dbl> <chr>     <dbl>   <dbl> <chr> <chr> <dbl>
## 1     1 one           4       1 A2    B2     5.45
## 2     2 one           4       2 A2    B2     5.69
## 3     3 one           4       3 A2    B2     5.65
## 4     4 one           4       4 A2    B2     5.55
## # … with 1,036 more rows
```

Converting Values Conditionally

- **Conditionals** are statements we give the computer where we want it to do something **only if a condition is met**.
- Using the tidyverse, we have a nice way to set these up using the function `case_when()`. It looks like this:

```
case_when(
  data == value ~ replacement,
  TRUE ~ data
)
```

- We give this function an argument relating to what **case** we care about (e.g. when the data is equal to a value).
- To the right of the ~ we tell R what the value should be in this case.
- We finally set a default value (`TRUE`), i.e. all other cases. Here, we tell R to just use the original value in the data.

Mutate

- `case_when()` can work on individual vectors of data, or on columns within a data frame if we combine it with the `mutate()` function.
- Here, we'll we'll recode an item in the item_id column. Let's change item 1 to item 20.

```
mutate(
  data_a,
  item_id = case_when(
    item_id == 1 ~ 20,
    TRUE ~ item_id
  )
)
```

```
## # A tibble: 1,040 x 7
##     row subj_id list_id item_id A     B          Y
##   <dbl> <chr>     <dbl>   <dbl> <chr> <chr> <dbl>
## 1     1 one           4      20 A2    B2     234.
## 2     2 one           4       2 A2    B2     295.
## 3     3 one           4       3 A2    B2     285.
## 4     4 one           4       4 A2    B2     258.
## # … with 1,036 more rows
```

Bind Rows

Our data is stored in two separate tables. Let's join them together using `bind_rows()`.

```
# bind it together
combined_data <- bind_rows(data_a, data_b)

# print it out
combined_data
```

```
## # A tibble: 2,400 x 7
##     row subj_id list_id item_id A     B          Y
##   <dbl> <chr>     <dbl>   <dbl> <chr> <chr> <dbl>
## 1     1 one           4       1 A2    B2     234.
## 2     2 one           4       2 A2    B2     295.
## 3     3 one           4       3 A2    B2     285.
## 4     4 one           4       4 A2    B2     258.
## # … with 2,396 more rows
```

Now our data has a total of 2400 rows. We've combined the tables together!

Bind Cols

We can bind columns together in a similar way.

- First, we'll extract the columns A and B, both only having the first 4 rows of our data each.
- Then to show how the function works, we'll bind these objects together using `bind_cols()`.

```
# get our data
col_a <- combined_data[1:4, "A"]
col_b <- combined_data[1:4, "B"]

# bind it together
bind_cols(col_a, col_b)
```

```
## # A tibble: 4 x 2
##   A     B
##   <chr> <chr>
## 1 A2    B2
## 2 A2    B2
## 3 A2    B2
## 4 A2    B2
```

Recap

We know...

- What **tidy data** is, and why we want our data to be in this format.
- How to **manipulate one-table data** for cleaning and preparation for analysis/graphing, including:
    - How to **filter** observations out of our data set.
    - How to **select** particular columns in our data set.
    - How to **mutate** or modify existing data and even create new columns of data.
- How to bind rows and columns together from **multiple tables**.