Managing Data in R

Glenn Williams

University of Sunderland

2021-10-26 (updated: 2021-10-31)

Importing Data into R

Understanding File Systems

Before we read in any data, we need to understand file paths on your computer.

- Computers are often split up into different **drives** where you can store data: e.g. a C drive on a PC, a Macintosh HD drive on newer Macs.
- The more drives you have, the more places data can live.
- Usually we organise files into folders on these drives.
- Any file is saved at a unique **paths** (places) on these drives.

Understanding File Systems

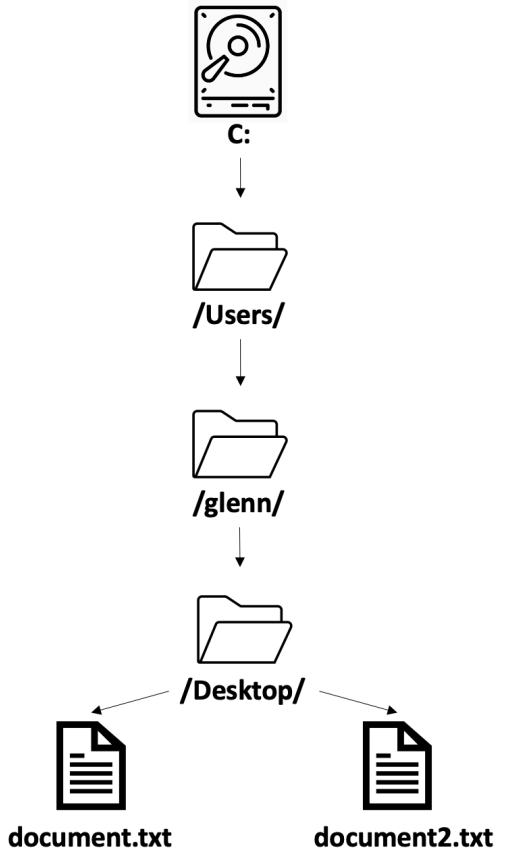
We have a file on our Desktop called "document.txt". This is a text file (.txt extension).

How does the computer know where it is?

- On a PC, it might live at `C:\Users\glenn\Desktop\document.txt`.
- Another file, "document2.txt", could live at `C:\Users\glenn\Desktop\document2.txt`.

You need to be able to **uniquely identify** files within file paths.

They either need unique names in a given location, or can have the same name in a different location.



A flow chart of these files on the system.

How R Reads Files

By default, R will either ask you to:

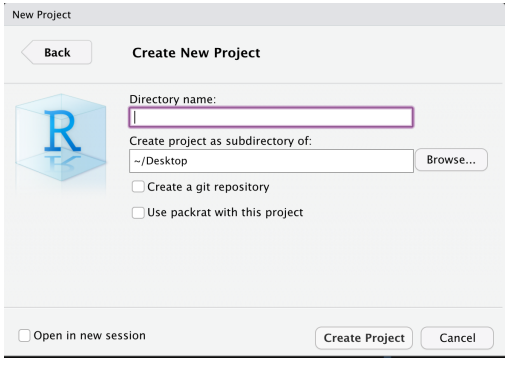
- Specify the **exact (absolute) file path** to access a file (e.g. `C:\Users\glenn\desktop\document.txt`), or...
- Specify a file path **relative** to the **working directory**.

Using a relative file path is best if you want your code to work on any computer besides the one you wrote your code in. That's because other computers aren't likely to have a `C:\Users\glenn\desktop` path.

Where's the **working directory** though?

The Working Directory

- The working directory in R by default is wherever you open up an R file, or wherever R is installed if you open RStudio without first opening a file.
- **Depending on how you start R, it could be anywhere!**
- RStudio Projects and the `here` package to the rescue!



Making an RStudio Project.

- If you open an **RStudio Project**, your working directory is wherever the project is.
- If you read files in with `here`, no matter where your working directory is right now, it reads files **relative** to where the RStudio Project lives.

Having your work associated with an **RStudio project ensures the working directory is always the same** in any session and on any computer.

Projects and Here: An Example

Read a file called **raw_data.csv** from a folder on our **desktop** called **my_data**.

Method

1. Using an **absolute file path** requires you to state exactly where the file is on your computer. This will only work on your computer.
2. Using a **relative file path** is better, but only works if the working directory is the my_data folder.
3. Using an **RStudio Project** fixes the working directory to the my_data folder. You can then read files relative to that folder using the **here function**.

Code

```
# using absolute paths
read_csv(
  "C:/Users/glenn/Desktop/
  my_data/raw_data.csv"
)


# using relative paths
read_csv("my_data/raw_data.csv")



# using the here function
read_csv(here("raw_data.csv"))
```

(3) is best for reproducibility.

Reading and Writing Data in R

There are a few inbuilt functions in R that allow us to read in data from a file and to write data to a file, but they sometimes take arguments in different orders and can behave unexpectedly.

- The `tidyverse` has a number of functions for both tasks that are consistent and tell you important information.
- All `tidyverse` functions that read data in from different formats start with **read**_ and end in the file format after the underscore; e.g. read_csv(), read_delim(), read_tsv().
- All `tidyverse` functions that write data to file in different formats start with **write**_ and end in the name of the file format; e.g. write_csv(), write_delim(), write_tsv().

Reading Data into R

An Example

- We already have some data stored in the /data/ folder. It's stored as a .csv file.
- Values in the file are separated by commas (hence, **comma-seperated values; csv**).
- It's a good idea to save data (e.g. from Excel) in this format for sharing; it's lightweight and can be read for free by most programs.

Reading Data into R

An Example

- We read the data in using `read_csv()`
- We tell R where to find the data using `here()`.
- We then assign the data to an object called **raw_data** so we can work with it in R.
- By default, `read_csv()` tells us how R parsed each column (i.e. what the data is stored as).

```
raw_data <- read_csv(here("data", "factorial_data.csv"))
```

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

Interim Recap

We know...

- how **file paths** work and that these paths can be very different depending upon your computer set up.
- why **relative file paths** set up in a **working directory** is often the most user-friendly method of accessing and writing files.
- why using **RStudio Projects** and the `here` package makes working with files easier.
- how to read different file types into R.

Manipulating Data in R

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"))
```

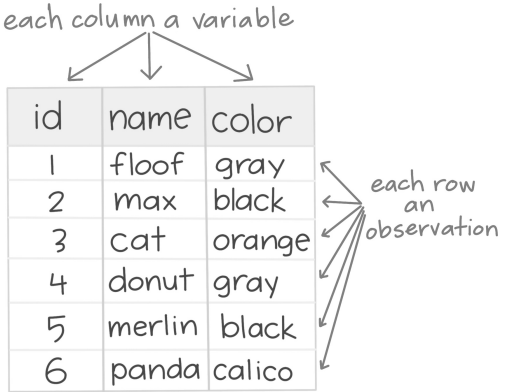
Tidy Data

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

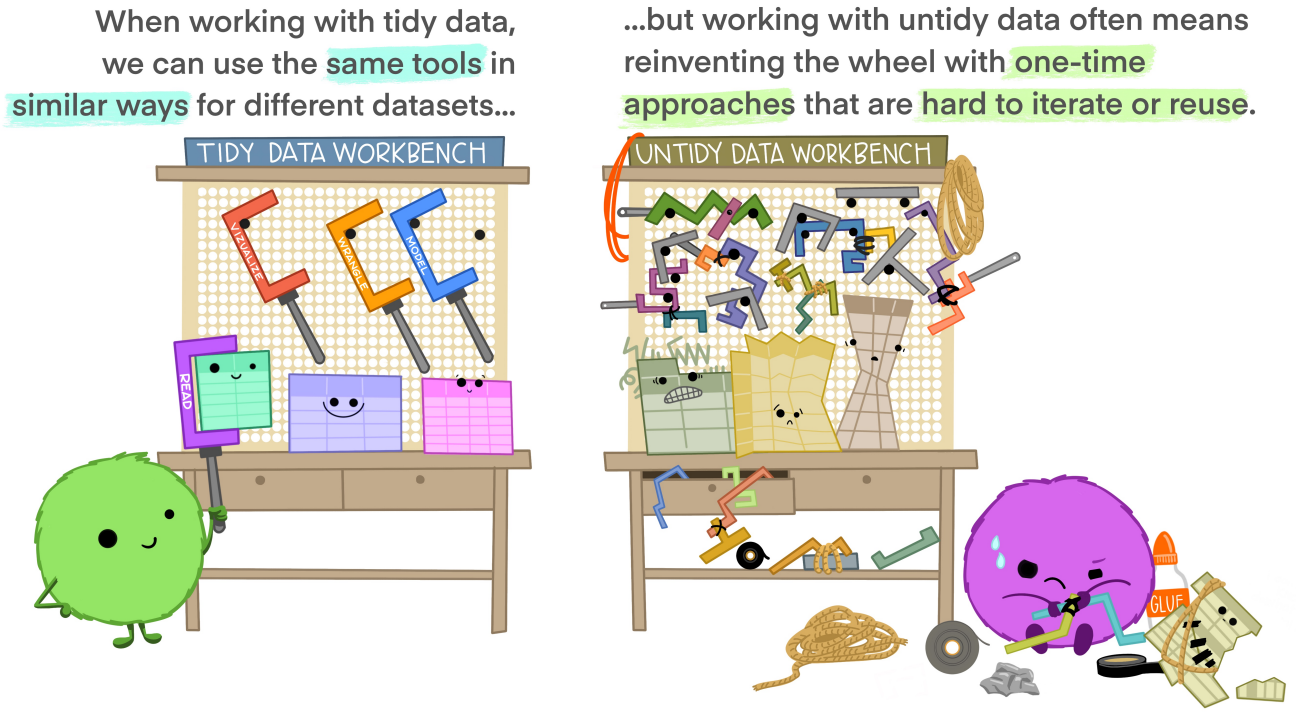


Wickham, H. (2014). Tidy Data. Journal of Statistical Software *59* (10). DOI: 10.18637/jss.v059.i10

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:

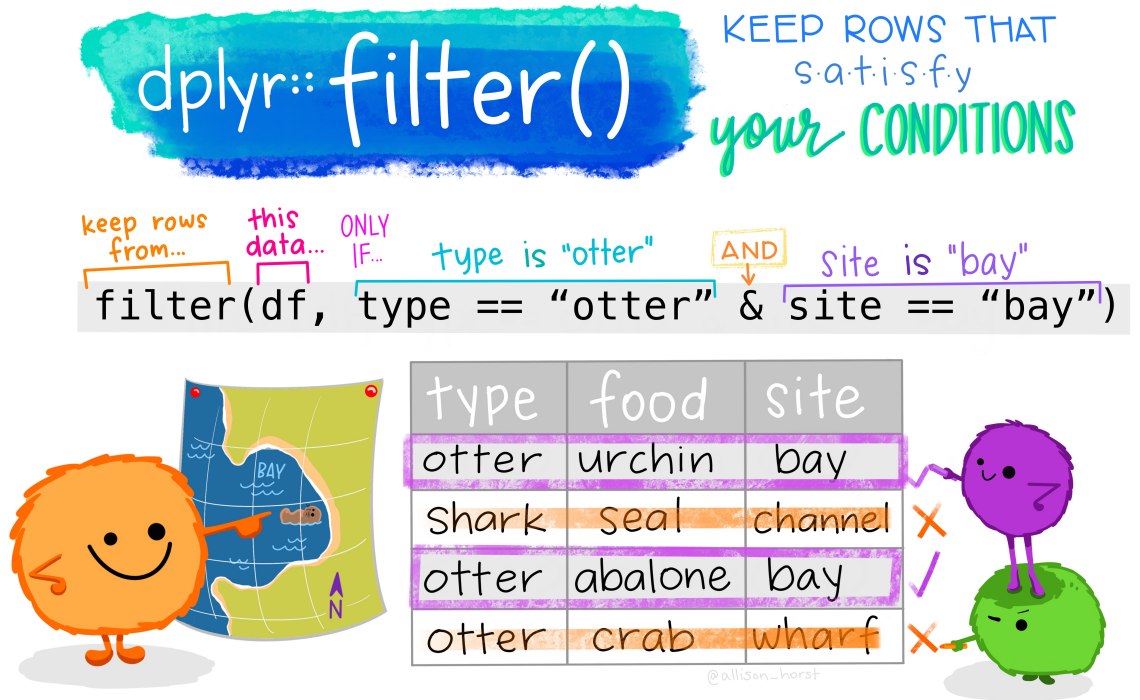
One Table:

- `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
```

Interim 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**.

Data Checking in R

Understanding our Data

To have a good understanding of how you should **clean and analyse your data**, you have to **understand how your data was made**. Ask yourself:

- How was the study performed?
- What do the variables in the data set represent?
- What levels can these variables have?
- Was a consistent coding scheme used throughout?
- How is missing data coded (if at all)?

Answering these questions makes you understand your data better, so you'll make fewer mistakes and do a better job of analysing it.

Reading in Data

In the data folder of this repository are files called **factorial_data_a.csv** and **factorial_data_b.csv**. If I'm working from this project folder I can read it in and combine the data as follows:

```
# read the data
data_a <- read_csv(here("data", "factorial_data_a.csv"))
data_b <- read_csv(here("data", "factorial_data_b.csv"))

# combine the data
combined_data <- bind_rows(data_a, data_b)

# print it
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
```

Some Notes on the Data

What does the data look like? The `head()` function allows you to see the top of your data set.

```
head(combined_data)
```

```
## # A tibble: 6 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.
## 5     5 one           4       5 A2    B2     268.
## 6     6 one           4       6 A2    B2     282.
```

The function `tail()` does the opposite to `head()`. Try it out!

Understanding our Data

Let's assume we gave people a task where they had to read **sentences that imply an upwards or downwards motion (variable A)** and then **identify targets at the top or bottom of the screen (variable B)** as quickly as they could. We captured their response times in the task (variable Y).

We have the following columns:

- **subj_id** = Subject/participant ID for the person who gave the data to us.
- **list_id** = List ID for how items were randomised.
- **item_id** = Item ID.
- **variable A** = Implied sentence location (upwards or downwards); e.g. "the bird flew in the sky."
- **variable B** = Location of target (up or down on the screen).

We should check that the data is coded as expected.

Inspecting Data

- Our first step to understanding our data should be to **check that we have read it into R correctly**, and that `read_csv()` has sensibly guessed at our data types.
- We can do this using the `glimpse()` function. This gives us a **look at our data on its side**, so we can see how our data are stored and how they are coded.

```
glimpse(combined_data)
```

```
## Rows: 2,400
## Columns: 7
## $ row     <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
## $ subj_id <chr> "one", "one", "one", "one", "one", "one", "one", "one", "o
## $ list_id <dbl> 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
## $ item_id <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
## $ A       <chr> "A2", "A2", "A2", "A2", "A2", "A2", "A2", "A2", "A2", "A2"
## $ B       <chr> "B2", "B2", "B2", "B2", "B2", "B2", "B2", "B2", "B2", "B2"
## $ Y       <dbl> 233.5056, 294.6706, 284.6235, 257.9815, 267.9257, 282.1853
```

There's a few problems with our coding system here. Subject ID should probably be numeric, and we need consistent codes for A and B.

Inspecting Data

- We can get a summary of our data, albeit in a confusing printout, using `summary()`.
- Here, we'll reduce the summary only to our numeric columns for easier presentation (selecting columns, only if they are numeric).

```
summary(select_if(combined_data, is.numeric))
```

```
##       row             list_id        item_id             Y          
##  Min.   :   1.0   Min.   :1.00   Min.   : 1.00   Min.   :-200.0  
##  1st Qu.: 600.8   1st Qu.:1.75   1st Qu.:10.75   1st Qu.: 231.7  
##  Median :1200.5   Median :2.50   Median :20.50   Median : 251.6  
##  Mean   :1200.5   Mean   :2.50   Mean   :20.50   Mean   : 255.4  
##  3rd Qu.:1800.2   3rd Qu.:3.25   3rd Qu.:30.25   3rd Qu.: 270.6  
##  Max.   :2400.0   Max.   :4.00   Max.   :40.00   Max.   :4000.0  
##                                                  NA's   :45
```

Our Y outcome, reaction time, seems to have a minimum of a negative value. We need to get rid of this clear error.

Inspecting Codes

Our list IDs and Item IDs are properly coded, but we should check our variables A and B, which indicate the conditions of the study.

- We can ask for only values from one column using dollar indexing; e.g. `data$column`.
- The `unique()` function tells us what unique observations we have in a column.

Which unique values do we have here?

```
unique(combined_data$A)
```

```
## [1] "A2" "A1" "a1" "a2"
```

```
unique(combined_data$B)
```

```
## [1] "B2"  "B1"  "b2"  "BB2" "BB1" "b1"
```

It looks like we have a mix of issues here. R is **case sensitive, meaning A is different to a**. We can make some initial fixes here by making everything the same case. Do this by using the `tolower()` function.

Fixing Problems

Set the values in A and B to lower case.

```
# change values
combined_data$A <- tolower(combined_data$A)
combined_data$B <- tolower(combined_data$B)
```

How does it look?

```
unique(combined_data$A)
```

```
## [1] "a2" "a1"
```

```
unique(combined_data$B)
```

```
## [1] "b2"  "b1"  "bb2" "bb1"
```

We've fixed the codes for variable A, but variable B still has some problems. Someone has mistakenly entered bb2 and bb1 as codes.

Let's change these!

Fixing Remaining Problems

We'll use the `case_when()` function again here.

If column B has the value "bb1" change it to "b1", otherwise if it has "bb2" change it to "b2", else just leave values as they are.

```
combined_data <- mutate(
  combined_data,
  B = case_when(
    B == "bb1" ~ "b1",
    B == "bb2" ~ "b2",
    TRUE ~ B
  )
)
```

How do the values look now?

```
unique(combined_data$B)
```

```
## [1] "b2" "b1"
```

Nice and consistent!

Identifying Remaining Problems

- We can see that subject IDs are stored as characters. Why is that? Let's see what unique values we have here.

```
unique(combined_data$subj_id)
```

```
##  [1] "one"          "2"            "3"            "4"            "5"
##  [6] "6"            "7"            "8"            "9"            "10"
## [11] "11"           "12"           "13"           "14"           "15"
## [16] "16"           "17"           "18"           "19"           "20"
## [21] "21"           "22"           "23"           "24"           "25"
## [26] "26"           "27"           "twenty-eight" "29"           "30"
## [31] "31"           "32"           "33"           "34"           "35"
## [36] "36"           "37"           "38"           "39"           "40"
## [41] "41"           "42"           "43"           "44"           "45"
## [46] "46"           "47"           "48"           "49"           "50"
## [51] "51"           "52"           "53"           "54"           "55"
## [56] "56"           "57"           "58"           "59"           "last"
```

We have a mix of integers, and the two values "one" and "last" as IDs. We need to make this consistent.

Converting Values Conditionally

Let's change the "one" and "last" values to more sensible values. Notice we have to stick with characters for now so the column has the same data types in it.

```
combined_data$subj_id <- case_when(
  combined_data$subj_id == "one" ~ "1",
  combined_data$subj_id == "twenty-eight" ~ "28",
  combined_data$subj_id == "last" ~ "60",
  TRUE ~ combined_data$subj_id
)
```

How have the values changed?

```
unique(combined_data$subj_id)
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
## [16] "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28" "29"
## [31] "31" "32" "33" "34" "35" "36" "37" "38" "39" "40" "41" "42" "43" "44"
## [46] "46" "47" "48" "49" "50" "51" "52" "53" "54" "55" "56" "57" "58" "59"
```

Nicely, the numbers now go from 1 to 60! But they're still characters.

Fixing Remaining Problems

We can convert between data types using the `as.` functions. We have e.g. `as.character()`, `as.factor()`, and `as.numeric()`. We'll use `as.numeric()` to make these strings into numeric values now they're consistently coded.

```
combined_data$subj_id <- as.numeric(combined_data$subj_id)
```

How do the values look? Check the data again. All good!

```
glimpse(combined_data)
```

```
## Rows: 2,400
## Columns: 7
## $ row     <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
## $ subj_id <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
## $ list_id <dbl> 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
## $ item_id <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
## $ A       <chr> "a2", "a2", "a2", "a2", "a2", "a2", "a2", "a2", "a2", "a2"
## $ B       <chr> "b2", "b2", "b2", "b2", "b2", "b2", "b2", "b2", "b2", "b2"
## $ Y       <dbl> 233.5056, 294.6706, 284.6235, 257.9815, 267.9257, 282.1853
```

Fixing Remaining Problems

Finally, we can't have negative reaction times. Let's remove any reaction times below 100ms under the assumption they're far too quick for this task.

```
# filter only to RTs above 100ms
combined_data <- filter(combined_data, Y > 100)

# inspect it
combined_data
```

```
## # A tibble: 2,349 x 7
##     row subj_id list_id item_id A     B          Y
##   <dbl>   <dbl>   <dbl>   <dbl> <chr> <chr> <dbl>
## 1     1       1       4       1 a2    b2     234.
## 2     2       1       4       2 a2    b2     295.
## 3     3       1       4       3 a2    b2     285.
## 4     4       1       4       4 a2    b2     258.
## # … with 2,345 more rows
```

All done!

Save our Cleaned Data

Finally, we did all that work with our data. To save repeating all these steps again, we can save it to an external file. We do this using 'write_csv()`.

Let's save our file in the data folder. This takes two arguments:

1. Which R object will you save to file?
2. Where should you save it? What will the data be called?

```
write_csv(
  combined_data,
  here("data", "cleaned_data.csv")
)
```

The nice thing is, if you made a mistake in one of your steps, you can just change that step and rerun your code.

If you change things by hand, who knows if you'll even detect the mistake, never mind how difficult it might be to fix it!

Interim Recap

We know...

- how to read data into R.
- how to find out how our data are coded.
- how to identify unique elements of our data.
- how to get a summary of our data.
- how to convert and clean up data.
- how to perform case-specific operations using conditional logic.