

# Data Wrangling

September 8, 2022

**Unit 1:** R for data mining

**Unit 2:** Prediction fundamentals

**Unit 3:** Regression-based methods

**Unit 4:** Tree-based methods

**Unit 5:** Deep learning

**Lecture 1:** Intro to modern data mining

**Lecture 2:** Data visualization

**Lecture 3:** Data transformation

**Lecture 4:** Data wrangling

**Lecture 5:** Unit review and quiz in class

## 1 Introduction

Unlike `diamonds`, data from the real world are not already built into an R package and are rarely as clean. This lecture is about **data wrangling**, the art of getting your data into R in a useful form for visualization and modeling. These notes draw on Chapters 10-15 from R4DS.

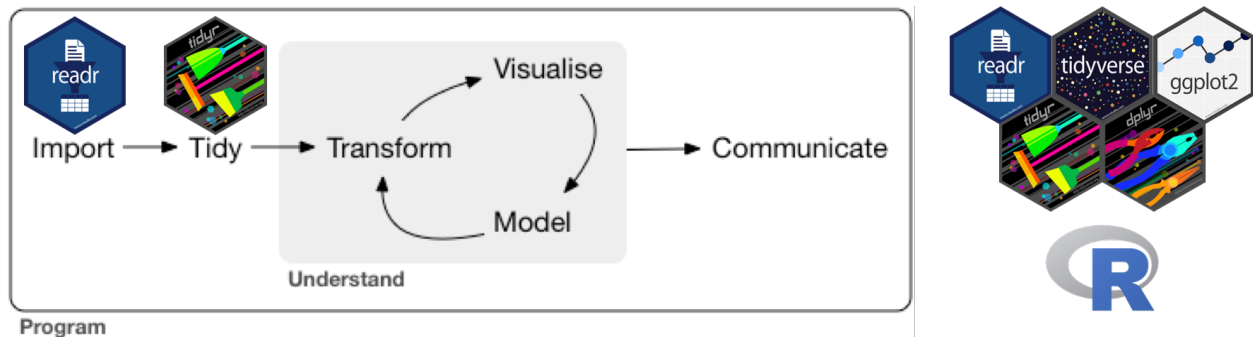


Figure 1: Image source: R4DS Chapter 9.

We will cover:

- Data import using `readr` (getting the data into R)
- Tidy data (the most convenient data format to work with in R)
- Data tidying using `tidyr` (getting our data into a format amenable to analysis)

Let's load the `tidyverse`:

```
library(tidyverse)
```

## 2 Data import (R4DS Chapter 11)

Data come in several different formats, e.g. comma-separated values (csv), tab-separated values (tsv), or Excel files. To read files in csv or tsv formats, use `read_csv` and `read_tsv`, respectively. These are both part of the `readr` package, which is part of the `tidyverse`. These functions are very similar to each other. To read Excel files, use the `read_excel` function from the `readxl` package.

Let's see how `read_csv` works. The simplest way of calling it is to specify just one argument (the location of the file you'd like to read):

```
heights = read_csv(file = "heights.csv")

## Rows: 1192 Columns: 6
## -- Column specification -----
## Delimiter: ","
## chr (2): sex, race
## dbl (4): earn, height, ed, 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.
heights

## # A tibble: 1,192 x 6
##   earn height sex      ed age race
##   <dbl> <dbl> <chr> <dbl> <dbl> <chr>
## 1 50000  74.4 male    16   45 white
## 2 60000  65.5 female  16   58 white
## 3 30000  63.6 female  16   29 white
## 4 50000  63.1 female  16   91 other
## 5 51000  63.4 female  17   39 white
## 6  9000  64.4 female  15   26 white
## 7 29000  61.7 female  12   49 white
## 8 32000  72.7 male    17   46 white
## 9  2000  72.0 male    15   21 hispanic
## 10 27000  72.2 male    12   26 white
## # ... with 1,182 more rows
## # i Use `print(n = ...)` to see more rows
```

Note that `read_csv` has automatically inferred the types of each column. It also made the assumption that the first line of the file are the column names. Sometimes, this is not the case. If column names are absent, you should specify the `col_names` argument either as `FALSE` or as a character vector of column names. Sometimes the files you'd like to read contain headers, i.e. one or more lines of metadata before the actual data starts. In this case, you can either skip a fixed number of lines (e.g. the first three) via `skip = 3` or skip any lines starting with a certain character (e.g. `#`) via `comment = "#"`. It's a good idea to first open the data file before deciding how to import it.

Exercise: Import `heights2.csv`.

### 3 Tidy data (R4DS Chapter 12)

“Happy families are all alike; every unhappy family is unhappy in its own way.” – Leo Tolstoy

“Tidy datasets are all alike, but every messy dataset is messy in its own way.” – Hadley Wickham

In this section, you will learn a consistent way to organise your data in R, an organisation called tidy data. Getting your data into this format requires some upfront work, but that work pays off in the long term. Once you have tidy data and the tidy tools provided by packages in the tidyverse, you will spend much less time munging data from one representation to another, allowing you to spend more time on the analytic questions at hand.

There are multiple ways to represent the same data:

```
table1
```

```
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>      <int> <int>      <int>
## 1 Afghanistan 1999     745   19987071
## 2 Afghanistan 2000    2666   20595360
## 3 Brazil       1999   37737   172006362
## 4 Brazil       2000   80488   174504898
## 5 China        1999  212258  1272915272
## 6 China        2000  213766  1280428583
```

table2

```
## # A tibble: 12 x 4
##   country      year type      count
##   <chr>      <int> <chr>      <int>
## 1 Afghanistan 1999 cases         745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000 cases         2666
## 4 Afghanistan 2000 population 20595360
## 5 Brazil       1999 cases        37737
## 6 Brazil       1999 population 172006362
## 7 Brazil       2000 cases        80488
## 8 Brazil       2000 population 174504898
## 9 China        1999 cases        212258
## 10 China       1999 population 1272915272
## 11 China       2000 cases        213766
## 12 China       2000 population 1280428583
```

table3

```
## # A tibble: 6 x 3
##   country      year rate
##   * <chr>      <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

table4a

```
## # A tibble: 3 x 3
##   country      `1999` `2000`
##   * <chr>      <int> <int>
## 1 Afghanistan     745    2666
## 2 Brazil          37737  80488
## 3 China          212258  213766
```

table4b

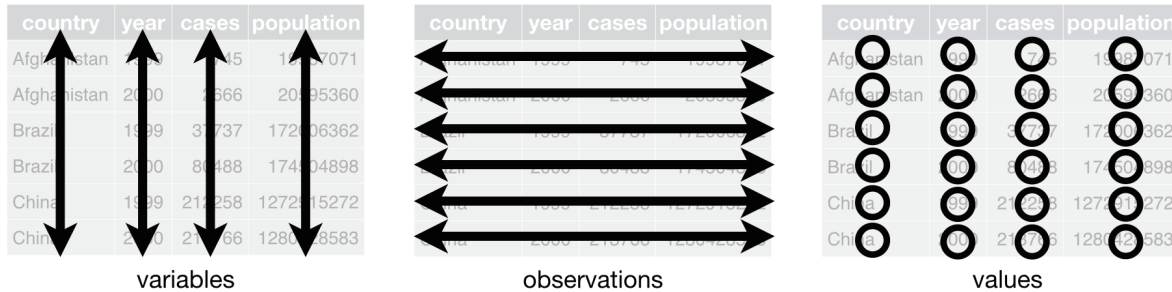
```
## # A tibble: 3 x 3
##   country      `1999` `2000`
##   * <chr>      <int> <int>
## 1 Afghanistan 19987071 20595360
## 2 Brazil      172006362 174504898
## 3 China      1272915272 1280428583
```

These are all representations of the same underlying data, but they are not equally easy to use. One dataset, the tidy dataset (`table1`), will be much easier to work with inside the tidyverse.

There are three interrelated rules which make a dataset tidy:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

The figure below shows the rules visually.



All the packages in the **tidyverse** are designed to work with tidy data. The **tidyr** package is designed to get non-tidy data into tidy format.

Exercise: Using prose, describe how the variables and observations are organised in each of the sample tables.

## 4 Pivoting

Once you get a non-tidy dataset, the first step is to figure out what the variables and observations are. Then, you want to get the variables into columns and get observations into rows.

- If one variable is spread across multiple columns, you'll need to `pivot_longer`.
- If one observation is scattered across multiple rows, you'll need to `pivot_wider`.

### 4.1 Longer

A common problem is a dataset where some of the column names are not names of variables, but *values* of a variable. Take `table4a`: the column names 1999 and 2000 represent values of the year variable, the values in the 1999 and 2000 columns represent values of the cases variable, and each row represents two observations, not one.

`table4a`

```
## # A tibble: 3 x 3
##   country    `1999` `2000`
## * <chr>      <int>  <int>
## 1 Afghanistan    745    2666
## 2 Brazil         37737   80488
## 3 China          212258  213766
```

To tidy a dataset like this, we need to **pivot** the offending columns into a new pair of variables. To describe that operation we need three parameters:

- `cols`: The set of columns whose names are values, not variables. In this example, those are the columns 1999 and 2000.

- `names_to`: The name of the variable to move the column names to. Here it is `year`.
- `values_to`: The name of the variable to move the column values to. Here it's `cases`.

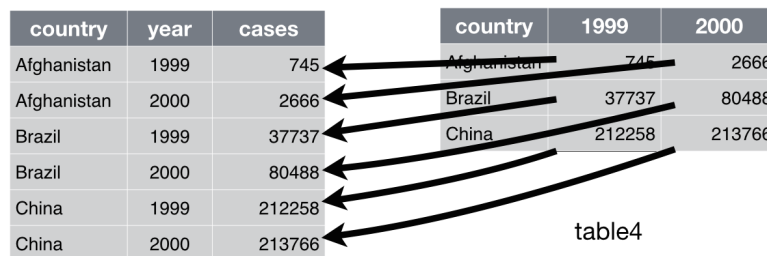
Together those parameters generate the call to `pivot_longer()`:

```
table4a %>%
  pivot_longer(cols = c(`1999`, `2000`), names_to = "year", values_to = "cases")
```

```
## # A tibble: 6 x 3
##   country    year  cases
##   <chr>      <chr> <int>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000  80488
## 5 China       1999  212258
## 6 China       2000 213766
```

Note that 1999 and 2000 are **non-syntactic names** (because they don't start with a letter) so we have to surround them in backticks.

In the final result, the pivoted columns are dropped, and we get new year and cases columns. Otherwise, the relationships between the original variables are preserved. Visually, this is shown in the figure below.



Exercise: Use `pivot_longer()` to tidy `table4b` in a similar fashion. What is the difference between the code used to tidy `table4a` and `table4b`?

To combine the tidied versions of `table4a` and `table4b` into a single tibble, we need to use `left_join()` from the `dplyr` package. See Section 5 below.

## 4.2 Wider

`pivot_wider()` is the opposite of `pivot_longer()`. You use it when an observation is scattered across multiple rows. For example, take `table2`: an observation is a country in a year, but each observation is spread across two rows.

```
table2

## # A tibble: 12 x 4
##   country    year type      count
##   <chr>      <int> <chr>      <int>
## 1 Afghanistan 1999 cases        745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000 cases        2666
## 4 Afghanistan 2000 population 20595360
```

```
## 5 Brazil      1999 cases      37737
## 6 Brazil      1999 population 172006362
## 7 Brazil      2000 cases      80488
## 8 Brazil      2000 population 174504898
## 9 China       1999 cases      212258
## 10 China      1999 population 1272915272
## 11 China      2000 cases      213766
## 12 China      2000 population 1280428583
```

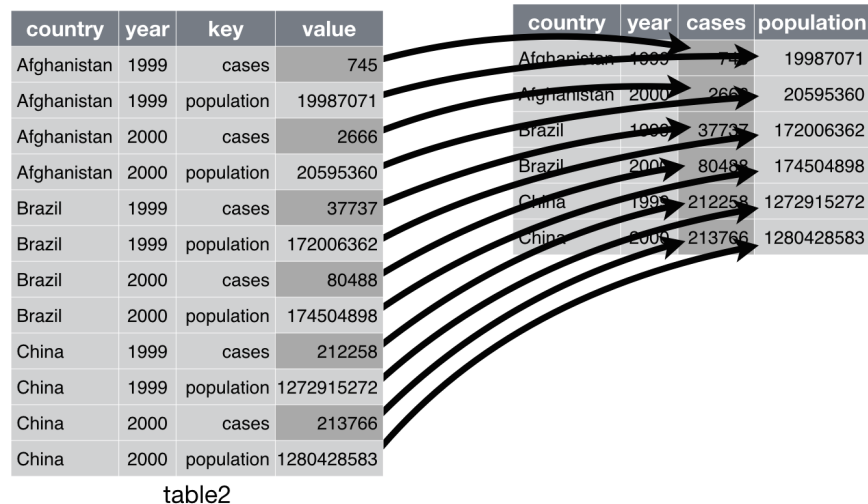
To tidy this up, we first analyse the representation in similar way to `pivot_longer()`. This time, however, we only need two parameters:

- The column to take variable names from. Here, it's `type`.
- The column to take values from. Here it's `count`.

Once we've figured that out, we can use `pivot_wider()`.

```
table2 %>%
  pivot_wider(names_from = type, values_from = count)
```

```
## # A tibble: 6 x 4
##   country    year cases population
##   <chr>      <int> <int>      <int>
## 1 Afghanistan 1999     745    19987071
## 2 Afghanistan 2000    2666   20595360
## 3 Brazil      1999   37737   172006362
## 4 Brazil      2000   80488   174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
```



Exercises:

1. Why does this code fail?

```
table4a %>%
  pivot_longer(cols = c(1999, 2000), names_to = "year", values_to = "cases")
# Error: Can't subset columns that don't exist.
# Locations 1999 and 2000 don't exist.
# There are only 3 columns.
```

2. Tidy the simple tibble below. Do you need to make it wider or longer? What are the variables?

```
tribble(  
  ~pregnant, ~male, ~female,  
  "yes",     NA,     10,  
  "no",      20,     12  
)
```

```
## # A tibble: 2 x 3  
##   pregnant male female  
##   <chr>    <dbl> <dbl>  
## 1 yes      NA      10  
## 2 no      20      12
```

## 5 Joining

It's rare that a data analysis involves only a single table of data. Typically you have many tables of data, and you must combine them to answer the questions that you're interested in. Collectively, multiple tables of data are called relational data because it is the relations, not just the individual datasets, that are important.

Recall the tidy versions of `table4a` and `table4b`:

```
tidy4a <- table4a %>%  
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases")  
tidy4b <- table4b %>%  
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "population")
```

tidy4a

```
## # A tibble: 6 x 3  
##   country    year cases  
##   <chr>    <chr> <int>  
## 1 Afghanistan 1999    745  
## 2 Afghanistan 2000   2666  
## 3 Brazil      1999  37737  
## 4 Brazil      2000  80488  
## 5 China       1999 212258  
## 6 China       2000 213766
```

tidy4b

```
## # A tibble: 6 x 3  
##   country    year population  
##   <chr>    <chr>      <int>  
## 1 Afghanistan 1999  19987071  
## 2 Afghanistan 2000  20595360  
## 3 Brazil      1999  172006362  
## 4 Brazil      2000  174504898  
## 5 China       1999 1272915272  
## 6 China       2000 1280428583
```

Joining two tables requires one or more **key** columns that are shared between the two tables. In this case, the key columns are `country` and `year`. There are several kinds of joins (see [R4DS Chapter 13](#)), but the most common is the **left join** (`left_join()` in `dplyr`). Given two tables `x` and `y`, `left_join(x,y)` tries to join `y` into `x`, keeping all rows in `x` (even if for some rows in `x` the key does not match any rows in `y`):

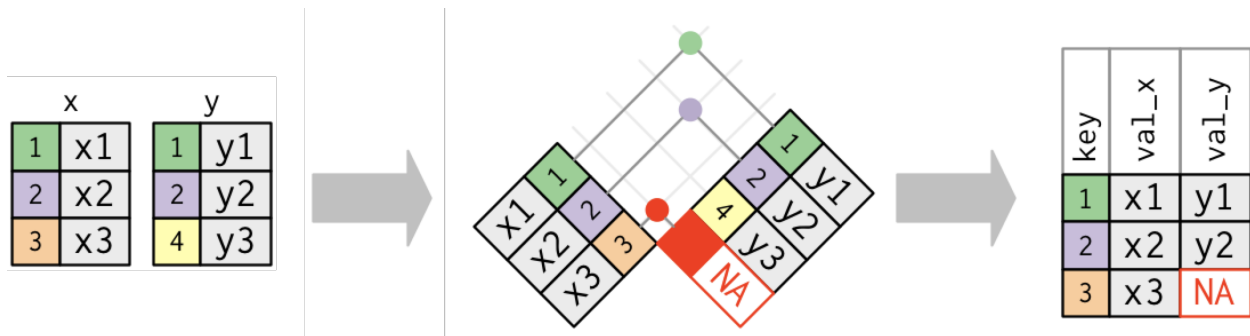


Figure 2: Left join (figure adapted from R4DS Ch. 13)

Let's apply `left_join()` to `tidy4a` and `tidy4b`:

```
left_join(x = tidy4a, y = tidy4b, by = c("country", "year"))
```

```
## # A tibble: 6 x 4
##   country    year  cases population
##   <chr>      <chr> <int>    <int>
## 1 Afghanistan 1999     745   19987071
## 2 Afghanistan 2000    2666  20595360
## 3 Brazil      1999   37737  172006362
## 4 Brazil      2000   80488  174504898
## 5 China       1999  212258 1272915272
## 6 China       2000  213766 1280428583
```

Exercise: Consider the two tibbles below. What is the key column? Without writing any code, can you predict how many rows and columns `left_join(x,y)` and `left_join(y,x)` will have?

```
x <- tribble(
  ~state, ~population,
  "PA", 12.8,
  "TX", 28.6,
  "NY", 19.5
)
y <- tribble(
  ~state, ~capital,
  "TX", "Austin",
  "CA", "Sacramento",
  "NY", "New York City",
  "MI", "Lansing"
)
```

## 6 Separating

So far you've learned how to tidy `table2` and `table4`, but not `table3`. `table3` has a different problem: we have one column (`rate`) that contains two variables (`cases` and `population`). To fix this problem, we'll need the `separate()` function.

`separate()` pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take `table3`:

```
table3
```



```
## # A tibble: 6 x 3
##   country    year rate
## * <chr>    <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China       2000 213766/1280428583
```

The `rate` column contains both `cases` and `population` variables, and we need to split it into two variables. `separate()` takes the name of the column to separate, and the names of the columns to separate into, as shown below.

```
table3 %>%
  separate(rate, into = c("cases", "population"))
```

```
## # A tibble: 6 x 4
##   country    year cases population
##   <chr>    <int> <chr>   <chr>
## 1 Afghanistan 1999 745     19987071
## 2 Afghanistan 2000 2666    20595360
## 3 Brazil      1999 37737   172006362
## 4 Brazil      2000 80488   174504898
## 5 China       1999 212258  1272915272
## 6 China       2000 213766  1280428583
```

country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

table3

By default, `separate()` will split values wherever it sees a non-alphanumeric character (i.e. a character that isn't a number or letter). For example, in the code above, `separate()` split the values of `rate` at the forward slash characters. If you wish to use a specific character to separate a column, you can pass the character to the `sep` argument of `separate()`. For example, we could rewrite the code above as:

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/")
```

```
## # A tibble: 6 x 4
##   country    year cases population
```

```
##   <chr>      <int> <chr> <chr>
## 1 Afghanistan 1999 745   19987071
## 2 Afghanistan 2000 2666  20595360
## 3 Brazil      1999 37737 172006362
## 4 Brazil      2000 80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
```

Look carefully at the column types: you'll notice that `cases` and `population` are character columns. This is the default behaviour in `separate()`: it leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask `separate()` to try and convert to better types using `convert = TRUE`:

```
table3 %>%
  separate(rate, into = c("cases", "population"), convert = TRUE)
```

```
## # A tibble: 6 x 4
##   country      year cases population
##   <chr>      <int> <int>      <int>
## 1 Afghanistan 1999    745   19987071
## 2 Afghanistan 2000   2666  20595360
## 3 Brazil      1999  37737 172006362
## 4 Brazil      2000  80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
```

You can also pass a vector of integers to `sep`. `separate()` will interpret the integers as positions to split at. Positive values start at 1 on the far-left of the strings; negative value start at -1 on the far-right of the strings. When using integers to separate strings, the length of `sep` should be one less than the number of names in `into`.

You can use this arrangement to separate the last two digits of each year. This make this data less tidy, but is useful in other cases.

```
table3 %>%
  separate(year, into = c("century", "year"), sep = 2)
```

```
## # A tibble: 6 x 4
##   country      century year rate
##   <chr>      <chr>   <chr> <chr>
## 1 Afghanistan 19      99   745/19987071
## 2 Afghanistan 20      00  2666/20595360
## 3 Brazil      19      99  37737/172006362
## 4 Brazil      20      00  80488/174504898
## 5 China       19      99  212258/1272915272
## 6 China       20      00  213766/1280428583
```

## 7 Missing values

Missing values, marked with NA, are often present in real datasets. Consider the following simple dataset:

```
stocks <- tibble(
  year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr    = c( 1,    2,    3,    4,    2,    3,    4),
  return = c(1.88, 0.59, 0.35, NA, 0.92, 0.17, 2.66)
)
stocks
```

```
## # A tibble: 7 x 3
##   year   qtr return
##   <dbl> <dbl> <dbl>
## 1  2015     1  1.88
## 2  2015     2  0.59
## 3  2015     3  0.35
## 4  2015     4  NA
## 5  2016     2  0.92
## 6  2016     3  0.17
## 7  2016     4  2.66
```

The NA means that the return for the fourth quarter of 2015 is missing. Changing the representation of a dataset can create more missing values. For example, let's pivot wider:

```
stocks %>%
  pivot_wider(names_from = year, values_from = return)
```

```
## # A tibble: 4 x 3
##   qtr `2015` `2016`
##   <dbl> <dbl> <dbl>
## 1     1  1.88  NA
## 2     2  0.59  0.92
## 3     3  0.35  0.17
## 4     4  NA    2.66
```

We see now that the return for the first quarter of 2016, which does not appear in the original dataset (implicitly missing), becomes an NA (explicitly missing).

Usually it's a good idea to treat missing values with care, e.g. by thinking about why those values might be missing in the first place. The simplest approach to dealing with missing values in a dataset is to remove all rows containing any missing values. This can be done via `na.omit()`. For example:

```
stocks %>%
  na.omit()
```

```
## # A tibble: 6 x 3
##   year   qtr return
##   <dbl> <dbl> <dbl>
## 1  2015     1  1.88
## 2  2015     2  0.59
## 3  2015     3  0.35
## 4  2016     2  0.92
## 5  2016     3  0.17
## 6  2016     4  2.66
```

## 8 References:

- [Data import cheat sheet](#)
- [tidyr cheat sheet](#)
- [R4DS Chapters 10-15](#)