MTH 3270 Notes 4

6 Tidy Data (6)

6.1 Introduction: The "tidyr" Package

- "***Tidy***" data are data arranged in rows and columns such that:

  1. The rows, called ***observations*** (or ***cases***), each refer to an individual person (or other specific, unique, and similar sort of thing).
  2. The columns, called ***variables***, each have the same sort of value recorded for each row, e.g. people's ages.

  All of the data sets we've seen so far have been "tidy."

- The "tidyr" package contains several functions for " tidying" data.

  It also contains functions for *iterating* a statistical analysis *by group*. Type:

```
help(package = tidyr)
```

  to see a list of the functions (and data sets) contained in "tidyr".

6.2 Using pivot_longer() and pivot_wider()

- Sometimes a **single variable** is spread across **multiple columns**. Other times, a single **observation** is scattered across **multiple rows**.

  For example, here are two data frames that contain the *same data* (student GPAs), but *arranged differently*, ***wide*** in the first case and ***narrow*** (or ***long***) in the second:

```
gpasWide

##   StudentID Semester1 Semester2 Semester3
## 1       111      2.54      3.42      3.93
## 2       112      2.90      3.19      3.18
## 3       113      3.99      3.45      2.89
## 4       114      2.99      2.78      3.70
## 5       115      3.67      2.68      2.81
```

```
gpasNarrow

##    StudentID  Semester  GPA
## 1        111 Semester1 2.54
## 2        111 Semester2 3.42
## 3        111 Semester3 3.93
## 4        112 Semester1 2.90
## 5        112 Semester2 3.19
## 6        112 Semester3 3.18
## 7        113 Semester1 3.99
## 8        113 Semester2 3.45
## 9        113 Semester3 2.89
## 10       114 Semester1 2.99
```

```
## 11        114 Semester2 2.78
## 12        114 Semester3 3.70
## 13        115 Semester1 3.67
## 14        115 Semester2 2.68
## 15        115 Semester3 2.81
```

In the **wide** format, the **variable** (GPA) is "widened" across multiple columns (Semesters 1-3). In the **narrow** format, those columns have been "lengthened" into a single column.

- The following functions, from the `"tidyr"` package, are useful for converting data back and forth between the **wide** and **narrow** formats:

```
pivot_longer()  # Convert from wide to narrow by stacking columns.
pivot_wider()   # Convert from narrow to wide by unstacking a column.
```

- To convert a data frame from the **wide** format, like `gpasWide`, to the **narrow** format, like `gpasNarrow`, use `pivot_longer()`.

For example, here's `gpasWide` (shown above):

```
studID <- c(111:115)
gpa1 <- c(2.54, 2.90, 3.99, 2.99, 3.67)
gpa2 <- c(3.42, 3.19, 3.45, 2.78, 2.68)
gpa3 <- c(3.93, 3.18, 2.89, 3.70, 2.81)

gpasWide <- data.frame(StudentID = studID,
                       Semester1 = gpa1,
                       Semester2 = gpa2,
                       Semester3 = gpa3)
```

This converts it to **narrow** format:

```
gpasNarrow <- pivot_longer(data = gpasWide,
                           cols = c(Semester1, Semester2, Semester3),
                           names_to = "Semester",
                           values_to = "GPA")

gpasNarrow

## # A tibble: 15 x 3
##    StudentID Semester    GPA
##        <int> <chr>     <dbl>
##  1       111 Semester1  2.54
##  2       111 Semester2  3.42
##  3       111 Semester3  3.93
##  4       112 Semester1  2.9
##  5       112 Semester2  3.19
##  6       112 Semester3  3.18
##  7       113 Semester1  3.99
##  8       113 Semester2  3.45
##  9       113 Semester3  2.89
## 10       114 Semester1  2.99
## 11       114 Semester2  2.78
## 12       114 Semester3  3.7
## 13       115 Semester1  3.67
## 14       115 Semester2  2.68
## 15       115 Semester3  2.81
```

You're free to invent any name for the `names_to` argument. Above, we used `"Semester"`. It's used as the name of the **categorical** variable in the **narrow** data frame whose *categories* are the *names* of the columns in the **wide** format (`Semester1`, `Semester2`, `Semester3` above) that get "lengthened".

You're also free to invent a name for the `values_to` argument. We used `"GPA"` above. It's the name of the variable in the **narrow** data frame that contains the *values* from the "lengthened" columns (student GPAs above).

- We can use the "helper" functions from `select()` (i.e. `starts_with()`, `ends_with()`, `contains()`, and `num_range()`) to specify columns to be "lengthened" in `pivot_longer()`. For example, to use `num_range()` to obtain the same result as the above, type:

```
gpasNarrow <- pivot_longer(data = gpasWide,
                           cols = num_range("Semester", 1:3),
                           names_to = "Semester",
                           values_to = "GPA")
```

- To covert from the **narrow** format to **wide**, use `pivot_wider()`.

For example, here's `gpasNarrow` (shown above):

```
studID <- rep(c(111:115), times = 3)
semester <- rep(c("Semester1", "Semester2", "Semester3"), each = 5)
gpa <- c(2.54, 2.90, 3.99, 2.99, 3.67, 3.42, 3.19, 3.45, 2.78, 2.68, 3.93, 3.18,
         2.89, 3.70, 2.81)

gpasNarrow <- data.frame(StudentID = studID,
                         GPA = gpa,
                         Semester = semester)
```

This converts it to **wide** format:

```
pivot_wider(data = gpasNarrow,
            names_from = Semester,
            values_from = GPA)

## # A tibble: 5 x 4
##   StudentID Semester1 Semester2 Semester3
##       <int>     <dbl>     <dbl>     <dbl>
## 1       111      2.54      3.42      3.93
## 2       112      2.9       3.19      3.18
## 3       113      3.99      3.45      2.89
## 4       114      2.99      2.78      3.7
## 5       115      3.67      2.68      2.81
```

In this case, the `names_from` and `values_from` arguments must be variables in the data frame being "widened" (`Semester` and `GPA` in `gpasNarrow` above).

- The `StudentID` variable in `gpasNarrow`, which for each student is constant across `Semester`s, is needed to match `GPA`s for a given student across `Semesters` to compose a row in `gpasWide`. Without the `StudentID` variable in `gpasNarrow`, `pivot_wider()` would return an error message.

Section 6.2 Exercises

Exercise 1 Here's a data frame containing responses for four individuals in each of three treatment groups in an experiment:

```
xWide <- data.frame(GrpA = c(1, 4, 2, 3),
                    GrpB = c(7, 5, 8, 6),
                    GrpC = c(9, 9, 8, 7))
xWide

##   GrpA GrpB GrpC
## 1    1    7    9
## 2    4    5    9
## 3    2    8    8
## 4    3    6    7
```

Write a command involving `pivot_longer()` that converts `xWide` to **narrow** format. Name the columns `Grp` and `Y`. Report your R command. You should end up with this:

```
xNarrow

## # A tibble: 12 x 2
##    Grp       Y
##    <chr> <dbl>
##  1 GrpA      1
##  2 GrpB      7
##  3 GrpC      9
##  4 GrpA      4
##  5 GrpB      5
##  6 GrpC      9
##  7 GrpA      2
##  8 GrpB      8
##  9 GrpC      8
## 10 GrpA      3
## 11 GrpB      6
## 12 GrpC      7
```

Exercise 2 Here are data from a study in which a variable `Y` was recorded on each of five subjects before and after an intervention:

```
xNarrow <- data.frame(Subject = c(1:5, 1:5),
                      Period = c("Before", "Before", "Before", "Before",
                                 "Before", "After", "After", "After",
                                 "After", "After"),
                      Y = c(22, 45, 32, 45, 30, 60, 44, 24, 56, 59),
                      stringsAsFactors = FALSE)
```

```
xNarrow

##    Subject Period  Y
## 1        1 Before 22
## 2        2 Before 45
## 3        3 Before 32
## 4        4 Before 45
## 5        5 Before 30
## 6        1  After 60
## 7        2  After 44
## 8        3  After 24
## 9        4  After 56
## 10       5  After 59
```

a) Write a command involving `pivot_wider()` that converts `xNarrow` to a **wide** format. Report your R command. You should end up with this:

```
xWide

## # A tibble: 5 x 3
##   Subject Before After
##     <int>  <dbl> <dbl>
## 1       1     22    60
## 2       2     45    44
## 3       3     32    24
## 4       4     45    56
## 5       5     30    59
```

b) The `Subject` variable in `xNarrow` is needed to match `Y` values for a given subject across `Periods` to compose their row in `xWide`. What would happen if the `Subject` variable was missing? Try it by running the command you wrote for part *a* on the following data frame:

```
xNarrowNoSubject <- data.frame(Period = c("Before", "Before", "Before",
                                          "Before", "Before", "After", "After",
                                          "After", "After", "After"),
                               Y = c(22, 45, 32, 45, 30, 60, 44, 24, 56, 59),
                               stringsAsFactors = FALSE)
```

Exercise 3 This exercise involves using the "helper" functions (from `select()`) in `pivot_longer()`. Recall that `num_range("x", 1:3)` matches `x1`, `x2`, `x3`.

Here's a **wide** data frame in which a variable was recorded on each of three `Subjects` at four different time points:

```
xWide <- data.frame(Subject = c(1001, 1002, 1003),
                    t1 = c(22, 45, 32),
                    t2 = c(45, 30, 60),
                    t3 = c(44, 24, 56),
                    t4 = c(55, 27, 53))

xWide

##   Subject t1 t2 t3 t4
## 1    1001 22 45 44 55
## 2    1002 45 30 24 27
## 3    1003 32 60 56 53
```

Write a command involving `pivot_longer()` and the "helper" function `num_range()` that converts `xWide` to **narrow** format. Report your R command. You should end up with this:

```
xNarrow

## # A tibble: 12 x 3
##    Subject Time      Y
##      <dbl> <chr> <dbl>
##  1    1001 t1       22
##  2    1001 t2       45
##  3    1001 t3       44
##  4    1001 t4       55
##  5    1002 t1       45
##  6    1002 t2       30
##  7    1002 t3       24
##  8    1002 t4       27
##  9    1003 t1       32
## 10    1003 t2       60
## 11    1003 t3       56
## 12    1003 t4       53
```

Exercise 4 Here's the **wide** data frame from Exercise 3, but this time it includes each `Subject`'s `Gender`:

```
xWide <- data.frame(Subject = c(1001, 1002, 1003),
                    Gender = c("m", "f", "f"),
                    t1 = c(22, 45, 32),
                    t2 = c(45, 30, 60),
                    t3 = c(44, 24, 56),
                    t4 = c(55, 27, 53))
xWide

##   Subject Gender t1 t2 t3 t4
## 1    1001      m 22 45 44 55
## 2    1002      f 45 30 24 27
## 3    1003      f 32 60 56 53
```

The `Gender` of a `Subject` is *constant* (i.e. doesn't change over the four time points). Thus we'd want the `Gender` column to be duplicated four times in the **narrow** format just as the `Subject` column was in Exercise 3.

What happens to the `Gender` column when you convert `xWide` to **narrow** format? Try it, by typing:

```
xNarrow <- pivot_longer(data = xWide,
                        cols = num_range("t", 1:4),
                        names_to = "Time",
                        values_to = "Y")
```

6.3 Separating and Uniting Columns Using `separate()` and `unite()`

- Sometimes a **single column** contains **multiple variables**. Other times, a **single variable** is split across **multiple columns**.

  The following functions (from `"tidyr"`) are useful for separating and uniting columns

  ```
  separate()     # Separate a column that contains multiple variables.
  unite()        # Unite multiple columns across which a single variable
                 # is spread (the reverse of separate()).
  ```

- Here's an example in which **two variables**, GPA and letter grade, are in a **single column**:

```
studID <- c(111:115)
gpaandgrade <- c("2.54/C", "2.9/B", "3.99/A", "2.99/B", "3.67/A")

gpasGrades <- data.frame(StudentID = studID,
                         GPAandGrade = gpaandgrade)

gpasGrades

##   StudentID GPAandGrade
## 1       111      2.54/C
## 2       112       2.9/B
## 3       113      3.99/A
## 4       114      2.99/B
## 5       115      3.67/A
```

- To split the `GPAandGrade` column into two columns, using `separate()`, type:

```r
separate(data = gpasGrades,
         col = GPAandGrade,
         into = c("GPA", "Grade"),
         sep = "/")

##   StudentID  GPA Grade
## 1       111 2.54     C
## 2       112  2.9     B
## 3       113 3.99     A
## 4       114 2.99     B
## 5       115 3.67     A
```

We use the argument `col` to specify the name of the column to be separated, `into` to specify the names of the new columns (which we're free to invent), and `sep` to specify the `"character"` *separator* for forming the new columns.

For more info, look at the help page by typing:

```r
? separate
```

- The `unite()` function does the reverse of `separate()`: It forms a single column from multiple columns across which a single variable is spread.

  For more info, type:

```r
? unite
```

Section 6.3 Exercises

Exercise 5 After installing the `"dplyr"` package, make sure it's loaded:

```r
library(dplyr)
```

Here's a data frame containing the `Rate` of occurrences a rare disease (number of cases divided by population) and the `year` for three countries:

```r
diseases <- data.frame(country = c("Afghanistan", "Afghanistan",
                                   "Brazil", "Brazil", "China", "China"),
                       year = c(1999, 2000, 1999, 2000, 1999, 2000),
                       rate = c("745/19987071", "2666/20595360",
                                "37737/172006362", "80488/174504898",
                                "212258/1272915272", "213766/1280428583"))
diseases

##       country year              rate
## 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
```

Write a command involving `separate()` that separates the `rate` column into two columns named `cases` and `population`. Report your R command.

Exercise 6 Here's a data frame containing `Phosphate` and `Nitrate` levels in the South Platte River in Denver, Colorado, with the `Year`, `Month`, and `Day` of each observation:

```r
year <- c(2017, 2017, 2017, 2017, 2017, 2017, 2017, 2018, 2018, 2018, 2018, 2018,
          2018, 2018)
month <- c(6, 6, 7, 7, 7, 8, 8, 6, 6, 7, 7, 7, 8, 8)
day <- c(4, 18, 2, 16, 30, 13, 27, 3, 17, 1, 15, 29, 12, 26)
phosphate <- c(2.42, 3.50, 1.78, 2.46, 0.66, 1.16, 0.68, 0.90, 1.11, 1.25, 2.28,
               1.36, 0.43, 2.90)
nitrate <- c(3.38, 3.87, 1.28, 3.45, NA, 3.64, 1.88, 6.16, 2.55, 2.98, 3.90, 3.31,
             4.19, 5.35)

river <- data.frame(Year = year,
                    Month = month,
                    Day = day,
                    Phosphate = phosphate,
                    Nitrate = nitrate)
head(river)

##   Year Month Day Phosphate Nitrate
## 1 2017     6   4      2.42    3.38
## 2 2017     6  18      3.50    3.87
## 3 2017     7   2      1.78    1.28
## 4 2017     7  16      2.46    3.45
## 5 2017     7  30      0.66      NA
## 6 2017     8  13      1.16    3.64
```

Write a command involving `unite()` that combines the `Month`, `Day`, and `Year` columns into a single column named `Date` having the form month/day/year. Report your R command. You should end up with this:

```r
head(new_river)

##        Date Phosphate Nitrate
## 1  6/4/2017      2.42    3.38
## 2 6/18/2017      3.50    3.87
## 3  7/2/2017      1.78    1.28
## 4 7/16/2017      2.46    3.45
## 5 7/30/2017      0.66      NA
## 6 8/13/2017      1.16    3.64
```

6.4 Data Intake

- There are other ways to read data into R besides `read.csv()` and `read.table()`.
- ***Web scraping*** refers to reading data from an HTML web page. The `"rvest"` package (more specifically, the `"xml2"` package upon which `"rvest"` is built) has functions for **web scraping** (aka **"*harvesting*"** data).

  Among those functions are the following.

  ```r
  read_html()       # Read an HTML file into R from its URL.
  html_nodes()      # Select nodes (elements) from an HTML file that has
                    # been read into R.
  html_table()      # Convert an HTML table into a data frame.
  ```

- Reading data from a web page into R is a three-step process:
  1. Read the entire HTML file into R by downloading it from a URL using `read_html()`.
  2. Extract the data table(s) from the HTML file using `html_nodes()`.
  3. Convert the data table(s) into R data frames using `html_table()`.

- For example (from Section 6.4.1.2 of our textbook *Modern Data Science with R*), the Wikipedia page

  `https://en.wikipedia.org/wiki/Mile_run_world_record_progression`.

  has tables showing the progression of world record times for the mile run. Each table corresponds to a particular group of runners (e.g. professionals, amateurs, males, females, etc.).

  To read the data into R, we first type:

```
library(rvest)
```

```
url <- "https://en.wikipedia.org/wiki/Mile_run_world_record_progression"
tables <- url %>% read_html() %>% html_nodes("table")
```

The `tables` object isn't a data frame, it's a *list*, each element of which is the HTML code for one table:

```
is.list(tables)

## [1] TRUE

length(tables)

## [1] 12

tables

## {xml_nodeset (12)}
##  [1] <table class="wikitable"><tbody>\n<tr>\n<th>Time</th>\n<th>Athlete</th> ...
##  [2] <table class="wikitable"><tbody>\n<tr>\n<th>Time</th>\n<th>Athlete</th> ...
##  [3] <table class="wikitable"><tbody>\n<tr>\n<th>Time</th>\n<th>Athlete</th> ...
##  [4] <table class="wikitable"><tbody>\n<tr>\n<th>Time</th>\n<th>Auto</th>\n< ...
##  [5] <table class="wikitable"><tbody>\n<tr>\n<th>Time</th>\n<th>Auto</th>\n< ...
##  [6] <table class="wikitable"><tbody>\n<tr>\n<th>Time</th>\n<th>Athlete</th> ...
##  [7] <table class="wikitable"><tbody>\n<tr>\n<th>Time</th>\n<th>Athlete</th> ...
##  [8] <table class="wikitable"><tbody>\n<tr>\n<th>Time</th>\n<th>Auto</th>\n< ...
##  [9] <table class="wikitable"><tbody>\n<tr>\n<th>Time</th>\n<th>Athlete</th> ...
## [10] <table class="wikitable"><tbody>\n<tr>\n<th>Time</th>\n<th>Athlete</th> ...
## [11] <table class="nowraplinks mw-collapsible autocollapse navbox-inner" sty ...
## [12] <table class="nowraplinks navbox-subgroup" style="border-spacing:0"><tb ...
```

To convert one of the tables (the third one, say) to a data frame, type:

```
Table3 <- html_table(tables[[3]])
Table3

## # A tibble: 5 x 5
##   Time  Athlete             Nationality    Date              Venue
##   <chr> <chr>               <chr>          <chr>             <chr>
## 1 4:52  Cadet Marshall      United Kingdom 2 September 1852  Addiscome
## 2 4:45  Thomas Finch        United Kingdom 3 November 1858   Oxford
## 3 4:45  St. Vincent Hammick United Kingdom 15 November 1858  Oxford
## 4 4:40  Gerald Surman       United Kingdom 24 November 1859  Oxford
## 5 4:33  George Farran       United Kingdom 23 May 1862       Dublin
```

Section 6.4 Exercises

Exercise 7 Make sure the `"rvest"` package is loaded:

```
library(rvest)
```

Using the approach described above, and the same Wikipedia page,

`https://en.wikipedia.org/wiki/Mile_run_world_record_progression`.

create an R data frame containing the data from the **fourth** table of world record times for the mile run. Report your R commands.

Exercise 8 The Wikipedia page,

`https://en.wikipedia.org/wiki/World_population`.

has tables showing the world's populations organized in various ways.

Using the approach described above, create an R data frame containing the data from the **fifth** table of populations. Report your R commands. You should end up with this:

```
head(Table5)

## # A tibble: 6 x 6
##    Rank Country       Population    `% of world` Date        `Source(official ~
##   <int> <chr>         <chr>         <chr>        <chr>       <chr>
## 1     1 China         1,412,078,000 17.8%        23 Feb 2022 National populati~
## 2     2 India         1,388,396,236 17.5%        23 Feb 2022 National populati~
## 3     3 United States 333,276,387   4.20%        23 Feb 2022 National populati~
## 4     4 Indonesia     269,603,400   3.40%        1 Jul 2020  National annual p~
## 5     5 Pakistan      220,892,331   2.78%        1 Jul 2020  UN Projection[95]
## 6     6 Brazil        214,390,509   2.70%        23 Feb 2022 National populati~
```

6.5 Cleaning Data

6.5.1 Recoding

- Sometimes **categorical** data are coded as **integers**.

  We can **recode** them as `"character"` values by creating a ***codebook*** data frame indicating the correspondence between **integer** and `"character"` values, then using `"dplyr"`'s `left_join()`.

Data Set: `Houses`

The `Houses` data set (from Section 6.4.1.1 of our textbook *Modern Data Science with R*), is in the **houses-for-sale.txt** file at

`http://sites.msudenver.edu/ngrevsta/wp-content/uploads/sites/416/2021/02/houses-for-sale.txt`

It contains data on 1,728 houses for sale in Saratoga, NY, such as `living_area`, `price`, `bedrooms`, and `bathrooms`. The data on house systems such as `sewer` and `heat` have been stored as *numbers*, even though they are really *categorical*.

The variables are:

`price`	The selling price (US dollars).
`lot_size`	The lot size (acres).
`waterfront`	Whether the house is on the waterfront (0 = no, 1 = yes).
`age`	The age of the house (years).
`land_value`	The land value (US dollars).
`construction`	Whether the house is newly constructed (0 = no, 1 = yes).
`air_cond`	Whether the house has air conditioning (0 = no, 1 = yes).
`fuel`	Type of fuel (2 = gas, 3 = electric, 4 = oil).
`heat`	Type of heat (2 = hot air, 3 = hot water, 4 = electric).
`sewer`	Type of sewer (1 = none, 2 = private, 3 = public).
`living_area`	Living area (in square feet).
`pct_college`	Percent of residents in the neighborhood with college degrees.
`bedrooms`	Number of bedrooms.
`fireplaces`	Number of fireplaces.
`bathrooms`	Number of bathrooms.
`rooms`	Number of rooms.

- For example, below we create the `Houses` data frame:

```
myURL <-
  "http://sites.msudenver.edu/ngrevsta/wp-content/uploads/sites/416/2021/02/houses-for-sale.txt"
Houses <- read.csv(myURL, header = TRUE, sep = "\t")
head(Houses)
```

```
##    price lot_size waterfront age land_value construction air_cond fuel heat
## 1 132500     0.09          0  42      50000            0        0    3    4
## 2 181115     0.92          0   0      22300            0        0    2    3
## 3 109000     0.19          0 133       7300            0        0    2    3
## 4 155000     0.41          0  13      18700            0        0    2    2
## 5  86060     0.11          0   0      15000            1        1    2    2
## 6 120000     0.68          0  31      14000            0        0    2    2
##   sewer living_area pct_college bedrooms fireplaces bathrooms rooms
## 1     2         906          35        2          1       1.0     5
## 2     2        1953          51        3          0       2.5     6
## 3     3        1944          51        4          1       1.0     8
## 4     2        1944          51        3          1       1.5     5
## 5     3         840          51        2          0       1.0     3
## 6     2        1152          22        4          1       1.0     8
```

Specifying `sep = "\t"` in `read.csv` indicates that the columns in **houses-for-sale.txt** are separate by tabs.

We'll use a *subset* of the variables, namely `fuel`, `heat`, `sewer`, and `construction`:

```
Houses_small <- select(Houses, fuel, heat, sewer, construction)
```

```r
head(Houses_small)
```

```
##   fuel heat sewer construction
## 1    3    4     2            0
## 2    2    3     2            0
## 3    2    3     3            0
## 4    2    2     2            0
## 5    2    2     3            1
## 6    2    2     2            0
```

To *recode* `fuel` and `sewer` from *integers* to `"character"` values (`"gas"`, `"electric"`, `"oil"`, `"none"`, `"private"`, and `"public"`), we first create a ***codebook*** data frame (called `Translations` below) that can be used to translate the *integers* to `"character"`:

```r
myURL <-
  "http://sites.msudenver.edu/ngrevsta/wp-content/uploads/sites/416/2021/02/house_codes.txt"
Translations <- read.csv(myURL,
                         header = TRUE,
                         stringsAsFactors = FALSE,
                         sep = "\t")

Translations
```

```
##    code system_type   meaning
## 1     0   new_const        no
## 2     1   new_const       yes
## 3     1  sewer_type      none
## 4     2  sewer_type   private
## 5     3  sewer_type    public
## 6     0 central_air        no
## 7     1 central_air       yes
## 8     2   fuel_type       gas
## 9     3   fuel_type  electric
## 10    4   fuel_type       oil
## 11    2   heat_type   hot air
## 12    3   heat_type hot water
## 13    4   heat_type  electric
```

The same information can also be presented in a wide format:

```r
Codes <- Translations %>% pivot_wider(names_from = system_type,
                                      values_from = meaning,
                                      values_fill = list(meaning = "invalid"))
Codes
```

```
## # A tibble: 5 x 6
##    code new_const sewer_type central_air fuel_type heat_type
##   <int> <chr>     <chr>      <chr>       <chr>     <chr>
## 1     0 no        invalid    no          invalid   invalid
## 2     1 yes       none       yes         invalid   invalid
## 3     2 invalid   private    invalid     gas       hot air
## 4     3 invalid   public     invalid     electric  hot water
## 5     4 invalid   invalid    invalid     oil       electric
```

Specifying `values_fill = list(meaning = "invalid")` indicates that `"invalid"` should be used to fill `meaning` values that would otherwise be `NA` in the widened data frame.

Now we use `left_join()` to merge `Houses_small` with `Codes`, matching rows in `Codes` and `Houses_small` by the (integer) variables `code` and `fuel`.

```r
Houses_small <- left_join(x = Houses_small,
                          y = select(Codes, code, fuel_type), # Join using only code and
                                                              # fuel_type.
                          by = c(fuel = "code"))              # fuel gets matched with
                                                              # code.

head(Houses_small)

##   fuel heat sewer construction fuel_type
## 1    3    4     2            0  electric
## 2    2    3     2            0       gas
## 3    2    3     3            0       gas
## 4    2    2     2            0       gas
## 5    2    2     3            1       gas
## 6    2    2     2            0       gas
```

Now we'll do the same thing, but this time matching rows in `Codes` and `Houses_small` by the (integer) variables `code` and `sewer`.

```r
Houses_small <- left_join(x = Houses_small,
                          y = select(Codes, code, sewer_type), # Join using only code and
                                                               # sewer_type.
                          by = c(sewer = "code"))              # sewer gets matched with
                                                               # code.
```

Here's the resulting data set, with *recoded* `fuel` and `sewer` variables (named `fuel_type` and `sewer_type`) along with their original integer versions:

```r
head(Houses_small)

##   fuel heat sewer construction fuel_type sewer_type
## 1    3    4     2            0  electric    private
## 2    2    3     2            0       gas    private
## 3    2    3     3            0       gas     public
## 4    2    2     2            0       gas    private
## 5    2    2     3            1       gas     public
## 6    2    2     2            0       gas    private
```

Note that the sequence of two commands above (first *recoding* `fuel` then `sewer`) could've been done in a single command using the **pipe operator** %>%:

```r
Houses_small <- Houses_small %>%
   left_join(Codes %>%
             select(code, fuel_type), by = c(fuel = "code")) %>%
   left_join(Codes %>%
             select(code, sewer_type), by = c(sewer = "code"))
```

Section 6.5 Exercises

Exercise 9 Create the `Houses_small` data frame:

```r
myURL <-
  "http://sites.msudenver.edu/ngrevsta/wp-content/uploads/sites/416/2021/02/houses-for-sale.txt"

Houses <- read.csv(myURL, header = TRUE, sep = "\t")

Houses_small <- select(Houses, fuel, heat, sewer, construction)
```

Also, create the `Codes` *codebook* data frame:

```
myURL <-
  "http://sites.msudenver.edu/ngrevsta/wp-content/uploads/sites/416/2021/02/house_codes.txt"
Translations <- read.csv(myURL,
                         header = TRUE,
                         stringsAsFactors = FALSE,
                         sep = "\t")

Codes <- Translations %>% pivot_wider(names_from = system_type,
                                      values_from = meaning,
                                      values_fill = list(meaning = "invalid"))
```

Now *recode* `heat` from integers to `"character"` values (`"hot air"`, `"hot water"`, and `"electric"`) by using `left_join()` to merge `Houses_small` with `Codes`, matching rows in `Codes` and `Houses_small` by the (integer) variables `code` and `heat`. Report your R command(s).

6.5.2 From Strings (`"character"`) to Numbers

- Sometimes a **numeric** vector will inadvertently be read into R as `"character"`, and we need to convert it to numeric.

  One way to do this is with the `as.numeric()` function. Another is with the function `parse_number()` from the `"readr"` package.

  Other times, we may need to go the other way, i.e. convert from numeric to `"character"`, which can be done using `as.character()`.

```
as.numeric()          # Convert "character" to numeric, automatically converting
                      # non-numeric characters to NA.
parse_number()        # Convert "character" to numeric, automatically converting
                      # non-numeric characters to NA.
as.character()        # Convert numeric to "character".
```

- For example, here `y` is `"character"`:

```
my_data <- data.frame(Name = c("Joe", "Kim", "Al", "Don", "Ann"),
                      y = c("2", "5", "6", "1", "7"),
                      stringsAsFactors = FALSE)
```

```
str(my_data)
```

```
## data.frame: 5 obs. of  2 variables:
##  $ Name: chr  "Joe" "Kim" "Al" "Don" ...
##  $ y   : chr  "2" "5" "6" "1" ...
```

  We change `y` to numeric using `mutate()` and `as.numeric()` by typing:

```
my_data <- mutate(.data = my_data, y = as.numeric(y))
```

  and now `y` is **numeric** as desired:

```
str(my_data)
```

```
## data.frame: 5 obs. of  2 variables:
##  $ Name: chr  "Joe" "Kim" "Al" "Don" ...
##  $ y   : num  2 5 6 1 7
```

Above, we could've used `parse_number()` instead of `as.numeric()`.

- Both `as.numeric()` and `parse_number()` automatically convert non-numeric characters to NA.

  For example, suppose `"Unknown"` appeared in the 2nd position of `y`:

```
my_data <- data.frame(Name = c("Joe", "Kim", "Al", "Don", "Ann"),
                      y = c("2", "Unknown", "6", "1", "7"),
                      stringsAsFactors = FALSE)
```

  When we change `y` to numeric using `mutate()` and either `as.numeric()` or `parse_number()`, the `"Unknown"` gets converted to `NA` (and a warning message is printed):

```
my_data <- mutate(.data = my_data, y = as.numeric(y))

## Warning in mask$eval_all_mutate(quo):  NAs introduced by coercion
```

```
my_data

##   Name  y
## 1  Joe  2
## 2  Kim NA
## 3   Al  6
## 4  Don  1
## 5  Ann  7
```

- To go the other way (from *numeric* to `"character"`), use `as.character()`.

Section 6.5 Exercises

Exercise 10 Here's a data frame:

```
x <- data.frame(Name = c("Joe", "Lucy", "Tom", "Sally"),
                NumberChildren = c("2", "1", "0", "3"),
                stringsAsFactors = FALSE)
```

Note that `NumberChildren` is `"character"`:

```
str(x)
```

Write one or more commands using `mutate()` and either `as.numeric()` or `parse_number()` that convert the `NumberChildren` column of `x` to *numeric*. Check your answer using `str()`. Report your R command(s).

Exercise 11 Here's a modified version of the data frame from Exercise 10, with "Unknown" in the 2nd position of `NumberChildren`:

```
x <- data.frame(Name = c("Joe", "Lucy", "Tom", "Sally"),
                NumberChildren = c("2", "Unknown", "0", "3"),
                stringsAsFactors = FALSE)
```

Both `as.numeric()` and `parse_number()` automatically convert non-numeric characters to NA.

What happens to the value in the 2nd position of `NumberChildren` when you type the following:

```r
x <- mutate(x, NumberChildren = as.numeric(NumberChildren))
x
```

6.5.3 Dates

- Often **dates** end up being stored as `"character"` values in a data frame.

  When this is the case, R doesn't recognize the inherent ordering in the dates (e.g. `"16 December 2019"` should come *after* `"29 October 2019"`).

  It's preferable in this case to convert the variable to an object of class `"Date"`. R recognizes the ordering in objects that belong to the `"Date"` class.

- The `"lubridate"` package has several functions that are useful for working with date/time variables.

  The functions below convert dates stored as `"character"` vectors to `"Date"` objects.

  ```
  ymd()      # Converts "character" (year, month, day) to a "Date" object
  mdy()      # Converts "character" (month, day, year) to a "Date" object
  dmy()      # Converts "character" (day, month, year) to a "Date" object
  ymd_hms()  # Converts "character" (year, month, day, hour, minute,
             # second) to a "Date" object
  ```

  They can also be used to convert `"character"` vectors to so-called **POSIXct** objects.

  For a complete list of the functions in the `"lubridate"` package, type:

  ```r
  help(package = lubridate)
  ```

- Here are some examples:

  ```r
  library(lubridate)
  ```

  ```r
  class("12/18/73")

  ## [1] "character"

  myDate <- mdy("12/18/73")
  myDate

  ## [1] "1973-12-18"

  class(myDate)

  ## [1] "Date"
  ```

  ```r
  myDates <- mdy(c("12/18/73", "12/19/73", "12/20/73"))
  myDates

  ## [1] "1973-12-18" "1973-12-19" "1973-12-20"

  class(myDates)

  ## [1] "Date"
  ```

- Internally, `"Date"` objects are stored in R as *numerical* values – the number of *days* **since 01-01-1970** (the so-called **UNIX epoch**):

```
as.numeric(mdy("01-01-1970"))

## [1] 0

as.numeric(mdy("01-02-1970"))

## [1] 1

as.numeric(mdy("01-01-1971"))

## [1] 365
```

This allows for subtraction to find the elapsed number of days between two dates:

```
myDate1 <- mdy("12-20-1973")
myDate2 <- mdy("01-15-1974")
myDate2 - myDate1

## Time difference of 26 days
```

- It also allows for a vector of dates to be used as the x variable in a plot.

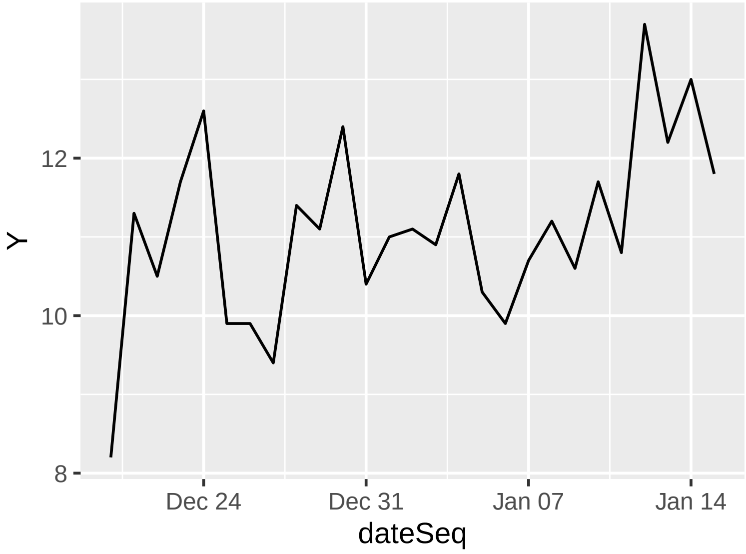
```
dateSeq <- seq(from = mdy("12-20-1973"), to = mdy("01-15-1974"), by = "days")
y <- c(8.2, 11.3, 10.5, 11.7, 12.6,  9.9, 9.9, 9.4, 11.4, 11.1, 12.4,
       10.4, 11.0, 11.1, 10.9, 11.8, 10.3, 9.9, 10.7, 11.2, 10.6, 11.7,
       10.8, 13.7, 12.2, 13.0, 11.8)
myData <- data.frame(Date = dateSeq, Y = y)
head(myData)

##         Date    Y
## 1 1973-12-20  8.2
## 2 1973-12-21 11.3
## 3 1973-12-22 10.5
## 4 1973-12-23 11.7
## 5 1973-12-24 12.6
## 6 1973-12-25  9.9

str(myData)

## 'data.frame': 27 obs. of  2 variables:
##  $ Date: Date, format: "1973-12-20" "1973-12-21" ...
##  $ Y   : num  8.2 11.3 10.5 11.7 12.6 9.9 9.9 9.4 11.4 11.1 ...
```

```
ggplot(data = myData, mapping = aes(x = dateSeq, y = Y)) +
  geom_line()
```



- Specific components of "Date" objects can be extracted using the following functions.

```
day()     # Get the day of the month from a "Date" object
mday()    # Same as day()
wday()    # Get the day of the week from a "Date" object
yday()    # Get the day of the year from a "Date" object
week()    # Get the week of the year from a "Date" object
```

- The "Date" class of objects (from the "lubridate" package) is most useful for dates that don't include the time of day.
- For **timestamp** data (also called **datetime** data), i.e. data that includes time of day (e.g. hour, minute, second), in which the **time zone** is important, the "POSIXct" and "POSIXlt" classes of objects are useful.

  The "POSIXct" and "POSIXlt" classes can generally be treated the same, but internally they're stored differently.

  "POSIXct" objects are stored as *numerical* values – the number of *seconds* **since 01-01-1970**. "POSIXlt" objects are stored as a *list* of year, month, day, hour, etc. "character" values.

Section 6.5 Exercises

Exercise 12 The functions ymd(), mdy(), etc. (from the "lubridate" package) recognize "character" dates in a variety of formats, and in each case covert from "character" to the "Date" class.

```
library(lubridate)
```

Guess what each of the following commands returns, then check your answers.

a) `mdy("Dec 18, 1973")`

b) `mdy("December 18, 1973")`

c) `mdy("12/18/1973")`

d) `mdy("12/18/73")`

e)
```r
mdy("12-18-1973")
```

f)
```r
mdy("12-18-73")
```

Exercise 13 Be careful when using `ymd()`, `mdy()`, etc. with `"character"` dates for which the century isn't given. Does `mdy()` interpret `"11/14/23"` as referring to the year 2023 or 1923? Try it.

```r
mdy("11/14/23")
```

Exercise 14 How many elapsed days are there between January 15, 2007 (`"1/15/07"`) and October 4, 2019 (`"10/4/19"`)?

Exercise 15 Guess what each of the following commands does, then check your answers.

a)
```r
seq(from = mdy("12-20-1993"), to = mdy("01-15-2004"),
    by = "days")
```

b)
```r
seq(from = mdy("12-20-1993"), to = mdy("01-15-2004"),
    by = "weeks")
```

c)
```r
seq(from = mdy("12-20-1993"), to = mdy("01-15-2004"),
    by = "years")
```

Exercise 16 Here's a data frame:

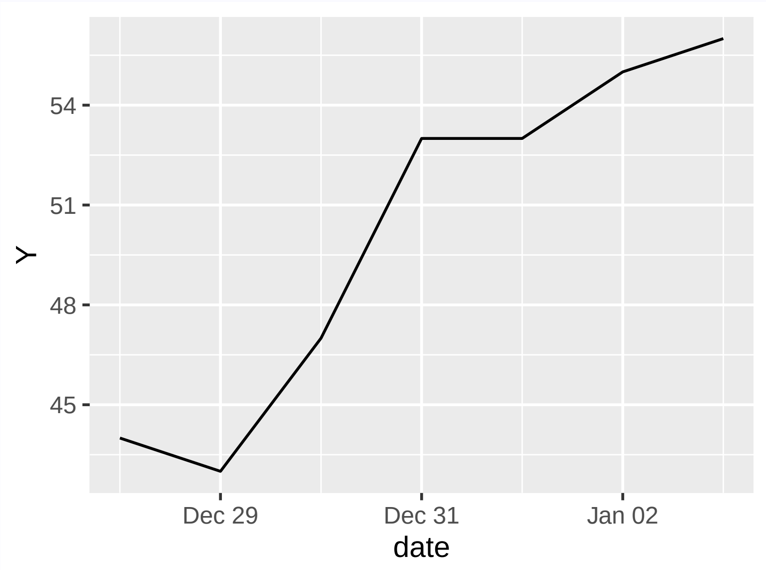
```r
my.data <- data.frame(Date = c("12/28/2017",  "12/29/2017", "12/30/2017",
                               "12/31/2017", "1/1/2018", "1/2/2018", "1/3/2018"),
                      Y = c(44, 43, 47, 53, 53, 55, 56))
```

a) Why doesn't the following plot command work?

```r
library(ggplot2)
```

```r
ggplot(data = my.data, mapping = aes(x = Date, y = Y)) +
  geom_line()
```

b) How can you use `mutate()` (from the `"dplyr"` package) and `mdy()` to fix the problem? Do it and report your R commands. You should end up with this:



7 Iteration (7)

7.1 Iteration Using `for()` and `nest_by()` with `mutate()`

- Sometimes we need to ***iterate*** (repeatedly execute) a set of R commands, each time using a different set of input values.
- We've seen (Class Notes 1) that many of R's built-in functions are **vectorized**, meaning the computation they perform is **iterated** over elements of a *vector*.
- ***Looping*** is another way of **iterating** R commands. Loops are usually implemented using:

```
for()      # Iterate a set of statements a specified number of times
```

- A special case of **iteration** is applying the same function repeatedly, each time over a different **column** (variable) of a data frame, as the `summarize()` function does.

  The following function (from the `"dplyr"` package) is useful in this regard.

```
nest_by()     # Similar to group_by(), but instead of storing the group structure,
              # in the metadata it is made explicit in the data, giving each
              # group a single row along with a list-column of data frames that
              # contain all the data for that group.
```

7.1.1 Iteration Using a `for()` Loop

- As a simple (but not very useful) example, the following sequence of **five `print()` commands** prints the squares of the numbers $1, 2, ..., 5$ to the console (output not shown):

```
print(1^2)
print(2^2)
print(3^2)
print(4^2)
print(5^2)
```

  We can achieve the **same result** more succinctly using a `for()` loop by typing:

```
for(i in 1:5) {             # i takes the values 1, 2, 3, 4, 5 in succession.
  print(i^2)                # The print() statement is executed 5 times,
}                           # once for each value of i.

## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

  Above, `i` takes the values 1, 2, 3, 4, 5 in succession, and the command `print(i^2)` is executed **five times**, once for each value of `i`.

- The body of the `for()` loop above contained just one statement, `print(i)`. More generally, it can contain a whole set of statements within curly brackets `{ }`.

  The general form of a `for()` loop is:

```
for(var in seq) {
    statement1
    statement2
```

```
        .
        .
        .
      statementq
}
```

where `seq` is a vector, usually of the form `1:n`, and `var` (whose name you're free to change) takes values `seq[1]`, `seq[2]`, . . ., `seq[length(seq)]` (normally `1`, `2`, . . ., `n`) sequentially, each time triggering another iteration of the loop during which `statements 1` through `q` are executed.

The `statements` usually involve the variable `var`.

Section 7.1 Exercises

Exercise 17 Guess how many times `"Good Sport"` will be printed to the screen in the following set of commands. Then check your answer.

```
for(i in 1:5) {
    print("Good Sport")
}
```

Exercise 18 The sequence of values we iterate over doesn't have to be of the form `1:n`. Guess what will be printed to the screen in the following set of commands. Then check your answer.

```
x <- c(2, 4, 6, 8)

for(i in x) {
    print(i^2)
}
```

Exercise 19 The sum of squares

$$\sum_{i=1}^{10} i^2 \quad = \quad 1^2 + 2^2 + \cdots + 10^2$$

can be computed using a `for()` loop by typing:

```
sum.sq <- 0

for(i in 1:10) {
    sum.sq <- sum.sq + i^2
}

sum.sq

## [1] 385
```

a) Why is it necessary to make the assignment `sum.sq <- 0` *before* entering the loop? What would happen if `sum.sq <- 0` wasn't there? Try it (after removing `sum.sq` from your Workspace if it's there). **Hint**: Notice `sum.sq` appears on *both* sides of the assignment statement in the loop, and R evaluates the *right* side first.

```
rm(sum.sq)

for(i in 1:10) {
    sum.sq <- sum.sq + i^2
}
```

b) What would happen if `sum.sq <- 0` was mistakenly placed *inside* the loop? Try it:

```
for(i in 1:10) {
    sum.sq <- 0
    sum.sq <- sum.sq + i^2
}

sum.sq
```

Exercise 20

a) What does the following loop do?

```
num.sq <- rep(NA, 10)      # Pre-allocate a 10-element vector

for(i in 1:10) {
    num.sq[i] <- i^2
}

num.sq
```

b) Loops are relatively **slow** to execute in R. It's advisable to *avoid* using loops when possible, and instead use the *vectorized* property of R's arithmetic operators and built-in functions or use one of the `apply()` functions (`apply()`, `sapply()`, etc.).

What does the following command do?

```
num.sq <- (1:10)^2
```

7.1.2 Iteration Over Groups Using `"dplyr"`'s `nest_by()`

- We'll work with the `sleepstudy` data set from the `"lme4"` package.

Data Set: `sleepstudy`

The `sleepstudy` data set (in the `"lme4"` package) contains data on the average reaction time per day for subjects in a sleep deprivation study (Belenky et al. 2003). On day 0 the subjects had their normal amount of sleep. Starting that night they were restricted to 3 hours of sleep per night. The response variable, `Reaction`, represents average reaction times in milliseconds (ms) on a series of tests given each `Day` to each `Subject`.

The three variables are:

`Reaction`	Average reaction time (milliseconds).
`Days`	Days into the study (0-9)
`Subject`	Subject ID number.

- Here are (some of) the data:

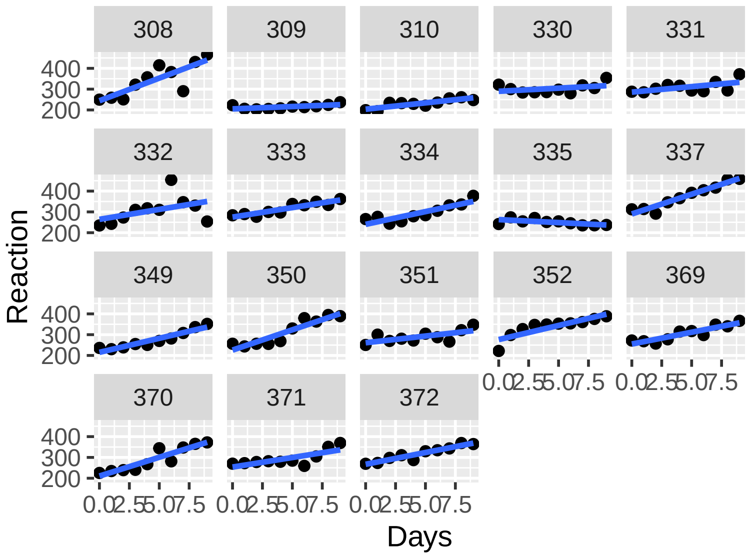
```
library(lme4)
head(sleepstudy)

##   Reaction Days Subject
## 1 249.5600    0     308
```

```
## 2 258.7047     1      308
## 3 250.8006     2      308
## 4 321.4398     3      308
## 5 356.8519     4      308
## 6 414.6901     5      308
```

- Here's a **faceted plot** of the data by `Subject`:

```
ggplot(data = sleepstudy, mapping = aes(x = Days, y = Reaction)) +
  facet_wrap(facets = ~ Subject) +
  geom_point() +
  geom_smooth(method = lm, se = FALSE)
```



The specification `method = lm` in `geom_smooth()` fits a "**linear model**", i.e. **straight line**, to the data.

- We can fit a "**linear model**" to a set of data using the built-in, `base` R `lm()` function, which also reports the **equation** of the fitted line.

```
lm()         # Carry out a linear regression analysis by fitting a
             # linear model to a data set.
summary()    # Summarize the results of the regression analysis.
```

To obtain the **equations** of the lines shown in the **faceted plot** above, we need to apply `lm()` separately to each `Subject`'s data.

We can do this using `"dplyr"`'s `nest_by()` function with the `sleepstudy` data, **grouped** by `Subject`, by typing:

```
by_subject <- nest_by(.data = sleepstudy, Subject)

head(by_subject)

## # A tibble: 6 x 2
## # Rowwise:  Subject
##   Subject                data
##   <fct>    <list<tibble[,2]>>
## 1 308                [10 x 2]
## 2 309                [10 x 2]
## 3 310                [10 x 2]
## 4 330                [10 x 2]
```

```
## 5 331                [10 x 2]
## 6 332                [10 x 2]
```

Above, `nest_by()` separates the `sleepstudy` into smaller data frames, each corresponding to a **Subject** in the `sleepstudy` data set.

Those **Subject**-specific data frames are stored ***nested*** within the larger `by_subject` data frame as elements of a ***list column*** named `data`.

Now we're ready to apply `lm()` separately to each **Subject**-specific data frame in the `data` column of `by_subject` using `mutate()` :

```
models <- mutate(.data = by_subject, mod = list(lm(Reaction ~ Days, data = data)))

head(models)

## # A tibble: 6 x 3
## # Rowwise:  Subject
##   Subject               data mod
##   <fct>   <list<tibble[,2]>> <list>
## 1 308                [10 x 2] <lm>
## 2 309                [10 x 2] <lm>
## 3 310                [10 x 2] <lm>
## 4 330                [10 x 2] <lm>
## 5 331                [10 x 2] <lm>
## 6 332                [10 x 2] <lm>
```

In `lm()`, the expression `Reaction ~ Days` (an R ***formula***) indicates that `Reaction` is the ***y*** **variable** and `Days` is the ***x*** **variable**.

The '`data = data`' says apply the `lm()` command separately to each **Subject**-specific data frame in the `data` column of `by_subject`.

`mutate()` returns a data frame (named `models` above). The first two columns will be the **group** labels (`Subject` numbers above), and **Subject**-specific data frames from `by_subject`. The third column (`mod` above) will be a *list-column* whose elements are the returned values of the function that's applied to the **groups** in `mutate()` (e.g. the the *lists* returned by `lm()` above).

We can look at the **equation** of the fitted line for, say, `Subject 330` (the 4th subject in the study) by typing:

```
models$mod[[4]]       # Gets the line for the 4th Subject.

##
## Call:
## lm(formula = Reaction ~ Days, data = data)
##
## Coefficients:
## (Intercept)         Days
##     289.685        3.008
```

The ***y-intercept*** is `289.685` and the ***slope*** is `3.008`, so the ***equation*** of the line is

$$Y = 289.685 + 3.008X,$$

and this is the line in the 4th facet of the faceted plot above.

- More examples on fitting linear models to each **group** in a grouped data frame can be found in the help file for `nest_by()`:

```
? nest_by
```

Section 7.1 Exercises

Exercise 21 Using the `sleepstudy` data (from the `"lme4"` package), use `mutate()` with `nest_by()` and `lm()` to fit lines separately to each `Subject`, with `Days` as the x variable and `Response` as the y variable by typing:

```
library(lme4)          # Contains the sleepstudy data set.

by_subject <- nest_by(.data = sleepstudy, Subject)
```

```
models <- mutate(.data = by_subject, mod = list(lm(Reaction ~ Days, data = data)))

models
```

What's the equation of the fitted line for `Subject 371` (the 17th subject in the study)?