Manipulating, analyzing and exporting data with tidyverse - Answers

This lesson is adapted from the Data Carpentry Ecology Lessons 2 and 3

This is an R Markdown Notebook. When you execute code within the notebook, the results appear beneath the code.

Execute code chunks by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Cmd+Shift+Enter*.

Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by pressing *Cmd+Option+I*.

When you save the notebook, a HTML file containing the code and output will be saved alongside it (click the *Preview* button or press *Cmd+Shift+K* to preview the HTML file).

The preview shows you a rendered HTML copy of the contents of the editor. Consequently, unlike *Knit*, *Preview* does not run any R code chunks. Instead, the output of the chunk when it was last run in the editor is displayed.

1. Presentation of the Survey Data

We are studying the species repartition and weight of animals caught in plots in our study area. The dataset is stored as a comma separated value (CSV) file. Each row holds information for a single animal, and the columns represent:

Column	Description
record_id	Unique id for the observation
month	month of observation
day	day of observation
year	year of observation
plot_id	ID of a particular plot
species_id	2-letter code
sex	sex of animal ("M", "F")
hindfoot_length	length of the hindfoot in mm
weight	weight of the animal in grams
genus	genus of animal
species	species of animal
taxon	e.g. Rodent, Reptile, Bird, Rabbit
plot_type	type of plot

We are going to use the R function `download.file()` to download the CSV file that contains the survey data from Figshare. Inside the download.file command, the first entry is a character string with the source URL ("https://ndownloader.figshare.com/files/2292169"). This source URL downloads a CSV file from figshare. The text after the comma ("data_raw/portal_data_joined.csv") is the destination of the file on your local machine. You'll need to have a folder on your machine called "data_raw" where you'll download the file. So this command downloads a file from Figshare, names it "portal_data_joined.csv" and adds it to a preexisting folder named "data_raw".

```r
download.file(url = "https://ndownloader.figshare.com/files/2292169",
              destfile = "data_raw/portal_data_joined.csv")
```

2. Data Manipulation using `dplyr` and `tidyr`

2.1 Loading thetidyverse package

`dplyr` is a package for making tabular data manipulation easier. It pairs nicely with **`tidyr`** which enables you to swiftly convert between different data formats for plotting and analysis.

Packages in R are basically sets of additional functions that let you do more stuff. The functions we've been using so far, like `length()` or `mean()`, come built into R; packages give you access to more of them. Before you use a package for the first time you need to install it on your machine, and then you should import it in every subsequent R session when you need it. You should already have installed the **`tidyverse`** package. This is an "umbrella-package" that installs several packages useful for data analysis which work together well such as **`tidyr`**, **`dplyr`**, **`ggplot2`**, **`tibble`**, etc.

The **`tidyverse`** package tries to address 3 common issues that arise when doing data analysis with some of the functions that come with R:

1. The results from a base R function sometimes depend on the type of data.
2. Using R expressions in a non standard way, which can be confusing for new learners.
3. Hidden arguments, having default operations that new learners are not aware of.

If we haven't already done so, we can type `install.packages("tidyverse")` straight into the console. In fact, it's better to write this in the console than in our script for any package, as there's no need to re-install packages every time we run the script.

Then, to load the package type:

```r
## load the tidyverse packages, incl. dplyr
library(tidyverse)
```

```
## Warning: package 'ggplot2' was built under R version 4.3.1

## Warning: package 'dplyr' was built under R version 4.3.1

## Warning: package 'stringr' was built under R version 4.3.1

## Warning: package 'lubridate' was built under R version 4.3.1

## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.4.4     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

2.2 What are `dplyr` and `tidyr`?

The package **`dplyr`** provides easy tools for the most common data manipulation tasks. It is built to work directly with data frames, with many common tasks optimized by being written in a compiled language (C++). An additional feature is the ability to work directly with data stored in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query are returned.

This addresses a common problem with R in that all operations are conducted in-memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can connect to a database of many hundreds of GB, conduct queries on it directly, and pull back into R only what you need for analysis.

The package **tidyr** addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data sets where we have one row per measurement. Sometimes we want a data frame where each measurement type has its own column, and rows are instead more aggregated groups - like plots or aquaria. Moving back and forth between these formats is non-trivial, and **tidyr** gives you tools for this and more sophisticated data manipulation.

To learn more about **dplyr** and **tidyr** after the workshop, you may want to check out this handy data transformation with **dplyr** cheatsheet and this one about **tidyr** data input.

We'll read in our data using the `read_csv()` function, from the tidyverse package **readr**.

```
surveys <- read_csv('data_raw/portal_data_joined.csv')
```

```
## Rows: 34786 Columns: 13
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (6): species_id, sex, genus, species, taxa, plot_type
## dbl (7): record_id, month, day, year, plot_id, hindfoot_length, weight
##
## 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.
```

You will see the message `Parsed with column specification`, followed by each column name and its data type. When you execute `read_csv` on a data file, it looks through the first 1000 rows of each column and guesses the data type for each column as it reads it into R. For example, in this dataset, `read_csv` reads `weight` as `col_double` (a numeric data type), and `species` as `col_character`. You have the option to specify the data type for a column manually by using the `col_types` argument in `read_csv`.

You can inspect the structure of the data:

```
## inspect the data
str(surveys)
```

```
## spc_tbl_ [34,786 x 13] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ record_id      : num [1:34786] 1 72 224 266 349 363 435 506 588 661 ...
##  $ month          : num [1:34786] 7 8 9 10 11 11 12 1 2 3 ...
##  $ day            : num [1:34786] 16 19 13 16 12 12 10 8 18 11 ...
##  $ year           : num [1:34786] 1977 1977 1977 1977 1977 ...
##  $ plot_id        : num [1:34786] 2 2 2 2 2 2 2 2 2 2 ...
##  $ species_id     : chr [1:34786] "NL" "NL" "NL" "NL" ...
##  $ sex            : chr [1:34786] "M" "M" NA NA ...
##  $ hindfoot_length: num [1:34786] 32 31 NA NA NA NA NA NA NA NA ...
##  $ weight         : num [1:34786] NA NA NA NA NA NA NA NA 218 NA ...
##  $ genus          : chr [1:34786] "Neotoma" "Neotoma" "Neotoma" "Neotoma" ...
##  $ species        : chr [1:34786] "albigula" "albigula" "albigula" "albigula" ...
##  $ taxa           : chr [1:34786] "Rodent" "Rodent" "Rodent" "Rodent" ...
##  $ plot_type      : chr [1:34786] "Control" "Control" "Control" "Control" ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   record_id = col_double(),
##   ..   month = col_double(),
##   ..   day = col_double(),
##   ..   year = col_double(),
##   ..   plot_id = col_double(),
##   ..   species_id = col_character(),
##   ..   sex = col_character(),
##   ..   hindfoot_length = col_double(),
##   ..   weight = col_double(),
##   ..   genus = col_character(),
##   ..   species = col_character(),
##   ..   taxa = col_character(),
##   ..   plot_type = col_character()
##   .. )
##  - attr(*, "problems")=<externalptr>
```

You can also preview the data:

```
## preview the data
View(surveys)
```

Notice that the class of the data is a `tbl_df`. This is referred to as a "tibble", the data structure is very similar to a data frame. Tibbles tweak some of the behaviors of the data frame objects we introduced in the previous lesson. For our purposes the only differences between a tibb;e and a dataframe are that:

1. In addition to displaying the data type of each column under its name, it only prints the first few rows of data and only as many columns as fit on one screen.
2. Columns of class `character` are never converted into factors.

3. Common `dplyr` functions

We're now going to learn some of the most common **`dplyr`** functions:

- `select()`: subset columns
- `filter()`: subset rows on conditions
- `mutate()`: create new columns by using information from other columns
- `group_by()` and `summarise()`: create summary statistics on grouped data
- `arrange()`: sort results
- `count()`: count discrete values

3.1 Selecting columns and filtering rows

To select columns of a data frame, use `select()`. The first argument to this function is the data frame (`surveys`), and the subsequent arguments are the columns to keep.

```
select(surveys, plot_id, species_id, weight)
```

```
## # A tibble: 34,786 x 3
##    plot_id species_id weight
##      <dbl> <chr>       <dbl>
##  1       2 NL             NA
##  2       2 NL             NA
##  3       2 NL             NA
##  4       2 NL             NA
##  5       2 NL             NA
##  6       2 NL             NA
##  7       2 NL             NA
##  8       2 NL             NA
##  9       2 NL            218
## 10       2 NL             NA
## # i 34,776 more rows
```

To select all columns *except* certain ones, put a "-" in front of the variable to exclude it.

```
select(surveys, -record_id, -species_id)
```

```
## # A tibble: 34,786 x 11
##    month   day  year plot_id sex   hindfoot_length weight genus   species  taxa
##    <dbl> <dbl> <dbl>   <dbl> <chr>           <dbl>  <dbl> <chr>   <chr>    <chr>
##  1     7    16  1977       2 M                  32     NA Neotoma albigula Rode~
##  2     8    19  1977       2 M                  31     NA Neotoma albigula Rode~
##  3     9    13  1977       2 <NA>               NA     NA Neotoma albigula Rode~
##  4    10    16  1977       2 <NA>               NA     NA Neotoma albigula Rode~
##  5    11    12  1977       2 <NA>               NA     NA Neotoma albigula Rode~
##  6    11    12  1977       2 <NA>               NA     NA Neotoma albigula Rode~
##  7    12    10  1977       2 <NA>               NA     NA Neotoma albigula Rode~
##  8     1     8  1978       2 <NA>               NA     NA Neotoma albigula Rode~
##  9     2    18  1978       2 M                  NA    218 Neotoma albigula Rode~
## 10     3    11  1978       2 <NA>               NA     NA Neotoma albigula Rode~
## # i 34,776 more rows
## # i 1 more variable: plot_type <chr>
```

This will select all the variables in `surveys` except `record_id` and `species_id`.

To choose rows based on a specific criterion, use `filter()`:

```
filter(surveys, year == 1995)
```

```
## # A tibble: 1,180 x 13
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     22314     6     7  1995       2 NL         M                  34     NA
##  2     22728     9    23  1995       2 NL         F                  32    165
##  3     22899    10    28  1995       2 NL         F                  32    171
##  4     23032    12     2  1995       2 NL         F                  33     NA
##  5     22003     1    11  1995       2 DM         M                  37     41
##  6     22042     2     4  1995       2 DM         F                  36     45
##  7     22044     2     4  1995       2 DM         M                  37     46
##  8     22105     3     4  1995       2 DM         F                  37     49
##  9     22109     3     4  1995       2 DM         M                  37     46
## 10     22168     4     1  1995       2 DM         M                  36     48
## # i 1,170 more rows
## # i 4 more variables: genus <chr>, species <chr>, taxa <chr>, plot_type <chr>
```

3.2 Pipes

What if you want to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes.

With intermediate steps, you create a temporary data frame and use that as input to the next function, like this:

```
surveys2 <- filter(surveys, weight < 5)
surveys_sml <- select(surveys2, species_id, sex, weight)
```

This is readable, but can clutter up your workspace with lots of objects that you have to name individually. With multiple steps, that can be hard to keep track of.

You can also nest functions (i.e. one function inside of another), like this:

```
surveys_sml <- select(filter(surveys, weight < 5), species_id, sex, weight)
```

This is handy, but can be difficult to read if too many functions are nested, as R evaluates the expression from the inside out (in this case, filtering, then selecting).

The last option, *pipes*, let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. Pipes in R look like `%>%` and are made available via the **`magrittr`** package, installed automatically with **`dplyr`**. If you use RStudio, you can type the pipe with Ctrl + Shift + M if you have a PC or Cmd + Shift + M if you have a Mac.

```
surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)
```

```
## # A tibble: 17 x 3
##    species_id sex   weight
##    <chr>      <chr>  <dbl>
##  1 PF         F          4
##  2 PF         F          4
##  3 PF         M          4
##  4 RM         F          4
##  5 RM         M          4
##  6 PF         <NA>       4
##  7 PP         M          4
##  8 RM         M          4
##  9 RM         M          4
## 10 RM         M          4
## 11 PF         M          4
## 12 PF         F          4
## 13 RM         M          4
## 14 RM         M          4
## 15 RM         F          4
## 16 RM         M          4
## 17 RM         M          4
```

In the above code, we use the pipe to send the `surveys` dataset first through `filter()` to keep rows where `weight` is less than 5, then through `select()` to keep only the `species_id`, `sex`, and `weight` columns. Since `%>%` takes the object on its left and passes it as the first argument to the function on its right, we don't need to explicitly include the data frame as an argument to the `filter()` and `select()` functions any more.

Some may find it helpful to read the pipe like the word "then". For instance, in the above example, we took the data frame `surveys`, *then* we `filter`ed for rows with `weight < 5`, *then* we `select`ed columns `species_id`, `sex`, and `weight`. The **`dplyr`** functions by themselves are somewhat simple, but by combining them into linear workflows with the pipe, we can accomplish more complex manipulations of data frames.

If we want to create a new object with this smaller version of the data, we can assign it a new name:

```
surveys_sml <- surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)

surveys_sml
```

```
## # A tibble: 17 x 3
##    species_id sex   weight
##    <chr>      <chr>  <dbl>
##  1 PF         F          4
##  2 PF         F          4
##  3 PF         M          4
##  4 RM         F          4
##  5 RM         M          4
##  6 PF         <NA>       4
##  7 PP         M          4
##  8 RM         M          4
##  9 RM         M          4
## 10 RM         M          4
## 11 PF         M          4
## 12 PF         F          4
## 13 RM         M          4
## 14 RM         M          4
## 15 RM         F          4
## 16 RM         M          4
## 17 RM         M          4
```

Note that the final data frame is the leftmost part of this expression.

3.3 Mutate

Frequently you'll want to create new columns based on the values in existing columns, for example to do unit conversions, or to find the ratio of values in two columns. For this we'll use `mutate()`.

To create a new column of weight in kg:

```
surveys %>%
  mutate(weight_kg = weight / 1000)
```

```
## # A tibble: 34,786 x 14
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1         1     7    16  1977       2 NL         M                  32     NA
##  2        72     8    19  1977       2 NL         M                  31     NA
##  3       224     9    13  1977       2 NL         <NA>               NA     NA
##  4       266    10    16  1977       2 NL         <NA>               NA     NA
##  5       349    11    12  1977       2 NL         <NA>               NA     NA
##  6       363    11    12  1977       2 NL         <NA>               NA     NA
##  7       435    12    10  1977       2 NL         <NA>               NA     NA
##  8       506     1     8  1978       2 NL         <NA>               NA     NA
##  9       588     2    18  1978       2 NL         M                  NA    218
## 10       661     3    11  1978       2 NL         <NA>               NA     NA
## # i 34,776 more rows
## # i 5 more variables: genus <chr>, species <chr>, taxa <chr>, plot_type <chr>,
## #   weight_kg <dbl>
```

You can also create a second new column based on the first new column within the same call of `mutate()`:

```
surveys %>%
  mutate(weight_kg = weight / 1000,
         weight_lb = weight_kg * 2.2)
```

```
## # A tibble: 34,786 x 15
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1         1     7    16  1977       2 NL         M                  32     NA
##  2        72     8    19  1977       2 NL         M                  31     NA
##  3       224     9    13  1977       2 NL         <NA>               NA     NA
##  4       266    10    16  1977       2 NL         <NA>               NA     NA
##  5       349    11    12  1977       2 NL         <NA>               NA     NA
##  6       363    11    12  1977       2 NL         <NA>               NA     NA
##  7       435    12    10  1977       2 NL         <NA>               NA     NA
##  8       506     1     8  1978       2 NL         <NA>               NA     NA
##  9       588     2    18  1978       2 NL         M                    NA    218
## 10       661     3    11  1978       2 NL         <NA>                 NA     NA
## # i 34,776 more rows
## # i 6 more variables: genus <chr>, species <chr>, taxa <chr>, plot_type <chr>,
## #   weight_kg <dbl>, weight_lb <dbl>
```

If you just want to see the first few rows, you can use a pipe to view the `head()` of the data. (Pipes work with non-**`dplyr`** functions, too, as long as the **`dplyr`** or `magrittr` package is loaded).

```
surveys %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

```
## # A tibble: 6 x 14
##   record_id month   day  year plot_id species_id sex   hindfoot_length weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 1         1     7    16  1977       2 NL         M                  32     NA
## 2        72     8    19  1977       2 NL         M                  31     NA
## 3       224     9    13  1977       2 NL         <NA>               NA     NA
## 4       266    10    16  1977       2 NL         <NA>               NA     NA
## 5       349    11    12  1977       2 NL         <NA>               NA     NA
## 6       363    11    12  1977       2 NL         <NA>               NA     NA
## # i 5 more variables: genus <chr>, species <chr>, taxa <chr>, plot_type <chr>,
## #   weight_kg <dbl>
```

The first few rows of the output are full of `NA`s, so if we wanted to remove those we could insert a `filter()` in the chain:

```
surveys %>%
  filter(!is.na(weight)) %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

```
## # A tibble: 6 x 14
##   record_id month   day  year plot_id species_id sex   hindfoot_length weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 1       588     2    18  1978       2 NL         M                  NA    218
## 2       845     5     6  1978       2 NL         M                  32    204
## 3       990     6     9  1978       2 NL         M                  NA    200
## 4      1164     8     5  1978       2 NL         M                  34    199
## 5      1261     9     4  1978       2 NL         M                  32    197
## 6      1453    11     5  1978       2 NL         M                  NA    218
## # i 5 more variables: genus <chr>, species <chr>, taxa <chr>, plot_type <chr>,
## #   weight_kg <dbl>
```

`is.na()` is a function that determines whether something is an `NA`. The `!` symbol negates the result, so we're asking for every row where weight *is not* an `NA`.

3.4 Challenges for `dplyr` functions

Challenge #1 Using pipes, subset the `surveys` data to include animals collected before 1995 and retain only the columns `year`, `sex`, and `weight`.

```
surveys %>%
  filter(year < 1995) %>%
  select(year, sex, weight)
```

```
## # A tibble: 21,486 x 3
##     year sex   weight
##    <dbl> <chr>  <dbl>
##  1  1977 M         NA
##  2  1977 M         NA
##  3  1977 <NA>      NA
##  4  1977 <NA>      NA
##  5  1977 <NA>      NA
##  6  1977 <NA>      NA
##  7  1977 <NA>      NA
##  8  1978 <NA>      NA
##  9  1978 M        218
## 10  1978 <NA>      NA
## # i 21,476 more rows
```

Challenge #2 Create a new data frame from the `surveys` data that meets the following criteria: contains only the `species_id` column and a new column called `hindfoot_cm` containing the `hindfoot_length` values converted to centimeters. In this `hindfoot_cm` column, there are no `NA`s and all values are less than 3.

Hint: think about how the commands should be ordered to produce this data frame!

```
surveys_hindfoot_cm <- surveys %>%
  filter(!is.na(hindfoot_length)) %>%
  mutate(hindfoot_cm = hindfoot_length / 10) %>%
  filter(hindfoot_cm < 3) %>%
  select(species_id, hindfoot_cm)
```

4. Split-apply-combine data analysis and the `summarise()` function

Many data analysis tasks can be approached using the *split-apply-combine* paradigm: split the data into groups, apply some analysis to each group, and then combine the results. **`dplyr`** makes this very easy through the use of the `group_by()` function.

4.1 The `summarise()` function

`group_by()` is often used together with `summarise()`, which collapses each group into a single-row summary of that group. `group_by()` takes as arguments the column names that contain the **categorical** variables for which you want to calculate the summary statistics. So to compute the mean `weight` by sex:

```
surveys %>%
  group_by(sex) %>%
  summarise(mean_weight = mean(weight, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   sex   mean_weight
##   <chr>       <dbl>
## 1 F            42.2
## 2 M            43.0
## 3 <NA>         64.7
```

You may also have noticed that the output from these calls doesn't run off the screen anymore. It's one of the advantages of `tbl_df` over data frame.

You can also group by multiple columns:

```
surveys %>%
  group_by(sex, species_id) %>%
  summarise(mean_weight = mean(weight, na.rm = TRUE)) %>%
  tail()
```

```
## `summarise()` has grouped output by 'sex'. You can override using the `.groups`
## argument.
```

```
## # A tibble: 6 x 3
## # Groups:   sex [1]
##   sex   species_id mean_weight
##   <chr> <chr>            <dbl>
## 1 <NA>  SU                 NaN
## 2 <NA>  UL                 NaN
## 3 <NA>  UP                 NaN
## 4 <NA>  UR                 NaN
## 5 <NA>  US                 NaN
## 6 <NA>  ZL                 NaN
```

Here, we used `tail()` to look at the last six rows of our summary. Before, we had used `head()` to look at the first six rows. We can see that the `sex` column contains `NA` values because some animals had escaped before their sex and body weights could be determined. The resulting `mean_weight` column does not contain `NA` but `NaN` (which refers to "Not a Number") because `mean()` was called on a vector of `NA` values while at the same time setting `na.rm = TRUE`. To avoid this, we can remove the missing values for weight before we attempt to calculate the summary statistics on weight. Because the missing values are removed first, we can omit `na.rm = TRUE` when computing the mean:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarise(mean_weight = mean(weight))
```

```
## `summarise()` has grouped output by 'sex'. You can override using the `.groups`
## argument.
```

```
## # A tibble: 64 x 3
## # Groups:   sex [3]
##    sex   species_id mean_weight
##    <chr> <chr>            <dbl>
##  1 F     BA                9.16
##  2 F     DM               41.6
##  3 F     DO               48.5
##  4 F     DS              118.
##  5 F     NL              154.
##  6 F     OL               31.1
##  7 F     OT               24.8
##  8 F     OX               21
##  9 F     PB               30.2
## 10 F     PE               22.8
## # i 54 more rows
```

Once the data are grouped, you can also summarise multiple variables at the same time (and not necessarily on the same variable). For instance, we could add a column indicating the minimum weight for each species for each sex:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarise(mean_weight = mean(weight),
            min_weight = min(weight))
```

```
## `summarise()` has grouped output by 'sex'. You can override using the `.groups`
## argument.
```

```
## # A tibble: 64 x 4
## # Groups:   sex [3]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 F     BA                9.16          6
##  2 F     DM               41.6          10
##  3 F     DO               48.5          12
##  4 F     DS              118.           45
##  5 F     NL              154.           32
##  6 F     OL               31.1          10
##  7 F     OT               24.8           5
##  8 F     OX               21            20
##  9 F     PB               30.2          12
## 10 F     PE               22.8          11
## # i 54 more rows
```

It is sometimes useful to rearrange the result of a query to inspect the values. For instance, we can sort on `min_weight` to put the lighter species first:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarise(mean_weight = mean(weight),
            min_weight = min(weight)) %>%
  arrange(min_weight)
```

```
## `summarise()` has grouped output by 'sex'. You can override using the `.groups`
## argument.
```

```
## # A tibble: 64 x 4
## # Groups:   sex [3]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 F     PF                7.97          4
##  2 F     RM               11.1           4
##  3 M     PF                7.89          4
##  4 M     PP               17.2           4
##  5 M     RM               10.1           4
##  6 <NA>  PF                6             4
##  7 F     OT               24.8           5
##  8 F     PP               17.2           5
##  9 F     BA                9.16          6
## 10 M     BA                7.36          6
## # i 54 more rows
```

To sort in descending order, we need to add the `desc()` function. If we want to sort the results by decreasing order of mean weight:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarise(mean_weight = mean(weight),
            min_weight = min(weight)) %>%
  arrange(desc(mean_weight))
```

```
## `summarise()` has grouped output by 'sex'. You can override using the `.groups`
## argument.
```

```
## # A tibble: 64 x 4
## # Groups:   sex [3]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 <NA>  NL               168.          83
##  2 M     NL               166.          30
##  3 F     NL               154.          32
##  4 M     SS               130          130
##  5 <NA>  SH               130          130
##  6 M     DS               122.          12
##  7 <NA>  DS               120           78
##  8 F     DS               118.          45
##  9 F     SH                78.8         30
## 10 F     SF                69           46
## # i 54 more rows
```

4.2 Counting

When working with data, we often want to know the number of observations found for each factor or combination of factors. For this task, **dplyr** provides `count()`. For example, if we wanted to count the number of rows of data for each sex, we would do:

```
surveys %>%
    count(sex)
```

```
## # A tibble: 3 x 2
##   sex       n
##   <chr> <int>
## 1 F     15690
## 2 M     17348
## 3 <NA>   1748
```

The `count()` function is shorthand for something we've already seen: grouping by a variable, and summarizing it by counting the number of observations in that group. In other words, `surveys %>% count()` is equivalent to:

```
surveys %>%
    group_by(sex) %>%
    summarise(count = n())
```

```
## # A tibble: 3 x 2
##   sex   count
##   <chr> <int>
## 1 F     15690
## 2 M     17348
## 3 <NA>   1748
```

For convenience, `count()` provides the `sort` argument:

```
surveys %>%
    count(sex, sort = TRUE)
```

```
## # A tibble: 3 x 2
##   sex       n
##   <chr> <int>
## 1 M     17348
## 2 F     15690
## 3 <NA>   1748
```

Previous example shows the use of `count()` to count the number of rows/observations for *one* factor (i.e., `sex`).

If we wanted to count *combination of factors*, such as `sex` and `species`, we would specify the first and the second factor as the arguments of `count()`:

```
surveys %>%
  count(sex, species)
```

```
## # A tibble: 81 x 3
##    sex   species         n
##    <chr> <chr>       <int>
##  1 F     albigula      675
##  2 F     baileyi      1646
##  3 F     eremicus      579
##  4 F     flavus        757
##  5 F     fulvescens     57
##  6 F     fulviventer    17
##  7 F     hispidus       99
##  8 F     leucogaster   475
##  9 F     leucopus       16
## 10 F     maniculatus   382
## # i 71 more rows
```

With the above code, we can proceed with `arrange()` to sort the table according to a number of criteria so that we have a better comparison.

For instance, we might want to arrange the table above in (i) an alphabetical order of the levels of the species and (ii) in descending order of the count:

```
surveys %>%
  count(sex, species) %>%
  arrange(species, desc(n))
```

```
## # A tibble: 81 x 3
##    sex   species             n
##    <chr> <chr>           <int>
##  1 F     albigula          675
##  2 M     albigula          502
##  3 <NA>  albigula           75
##  4 <NA>  audubonii          75
##  5 F     baileyi          1646
##  6 M     baileyi          1216
##  7 <NA>  baileyi            29
##  8 <NA>  bilineata         303
##  9 <NA>  brunneicapillus    50
## 10 <NA>  chlorurus          39
## # i 71 more rows
```

From the table above, we may learn that, for instance, there are 75 observations of the *albigula* species that are not specified for its sex (i.e. `NA`).

4.3 Challenges for Split-apply-combine

Challenge #3 How many animals were caught in each `plot_type` surveyed?

```
surveys %>%
  count(plot_type)
```

```
## # A tibble: 5 x 2
##   plot_type                     n
##   <chr>                     <int>
## 1 Control                   15611
## 2 Long-term Krat Exclosure   5118
## 3 Rodent Exclosure           4233
## 4 Short-term Krat Exclosure  5906
## 5 Spectab exclosure          3918
```

Challenge #4 Use `group_by()` and `summarise()` to find the mean, min, and max hindfoot length for each species (using `species_id`). Also add the number of observations (hint: see `?n`).

```
surveys %>%
  filter(!is.na(hindfoot_length)) %>%
  group_by(species_id) %>%
  summarise(
    mean_hindfoot_length = mean(hindfoot_length),
    min_hindfoot_length = min(hindfoot_length),
    max_hindfoot_length = max(hindfoot_length),
    n = n()
    )
```

```
## # A tibble: 25 x 5
##    species_id mean_hindfoot_length min_hindfoot_length max_hindfoot_length     n
##    <chr>                     <dbl>               <dbl>               <dbl> <int>
##  1 AH                         33                    31                  35     2
##  2 BA                         13                     6                  16    45
##  3 DM                         36.0                  16                  50  9972
##  4 DO                         35.6                  26                  64  2887
##  5 DS                         49.9                  39                  58  2132
##  6 NL                         32.3                  21                  70  1074
##  7 OL                         20.5                  12                  39   920
##  8 OT                         20.3                  13                  50  2139
##  9 OX                         19.1                  13                  21     8
## 10 PB                         26.1                   2                  47  2864
## # i 15 more rows
```

Challenge #5 What was the heaviest animal measured in each year? Return the columns `year`, `genus`, `species_id`, and `weight`.

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(year) %>%
  filter(weight == max(weight)) %>%
  select(year, genus, species, weight) %>%
  arrange(year)
```

```
## # A tibble: 27 x 4
## # Groups:   year [26]
##      year genus     species     weight
##     <dbl> <chr>     <chr>        <dbl>
##   1  1977 Dipodomys spectabilis    149
##   2  1978 Neotoma   albigula       232
##   3  1978 Neotoma   albigula       232
##   4  1979 Neotoma   albigula       274
##   5  1980 Neotoma   albigula       243
##   6  1981 Neotoma   albigula       264
##   7  1982 Neotoma   albigula       252
##   8  1983 Neotoma   albigula       256
##   9  1984 Neotoma   albigula       259
##  10  1985 Neotoma   albigula       225
## # i 17 more rows
```

5. Exporting data

Now that you have learned how to use **`dplyr`** to extract information from or summarise your raw data, you may want to export these new data sets to share them with your collaborators or for archival.

Similar to the `read_csv()` function used for reading CSV files into R, there is a `write_csv()` function that generates CSV files from data frames.

Before using `write_csv()`, we are going to create a new folder, `data`, in our working directory that will store this generated dataset. We don't want to write generated datasets in the same directory as our raw data. It's good practice to keep them separate. The `data_raw` folder should only contain the raw, unaltered data, and should be left alone to make sure we don't delete or modify it. In contrast, our script will generate the contents of the `data` directory, so even if the files it contains are deleted, we can always re-generate them.

In preparation for our next lesson on plotting, we are going to prepare a cleaned up version of the data set that doesn't include any missing data.

We will start by removing observations of animals for which `weight` and `hindfoot_length` are missing, or the `sex` has not been determined.

Because we are interested in plotting how species abundances have changed through time, we are also going to remove observations for rare species (i.e., that have been observed less than 50 times). We will do this in two steps: first we are going to create a data set that counts how often each species has been observed, and filter out the rare species; then, we will extract only the observations for these more common species:

```
### Create the dataset for exporting:
##  Start by removing observations for which the `species_id`, `weight`,
##  `hindfoot_length`, or `sex` data are missing:
surveys_complete <- surveys %>%
  filter(species_id != "",        # remove missing species_id
         !is.na(weight),                 # remove missing weight
         !is.na(hindfoot_length),        # remove missing hindfoot_length
         sex != "")                      # remove missing sex

##  Now remove rare species in two steps. First, make a list of species which appear at least 50 times
species_counts <- surveys_complete %>%
  count(species_id) %>%
  filter(n >= 50) %>%
  select(species_id)

##  Second, keep only those species:
surveys_complete <- surveys_complete %>%
  filter(species_id %in% species_counts$species_id)
```

To make sure that everyone has the same data set, check that `surveys_complete` has 30463 rows and 13 columns by typing `dim(surveys_complete)`.

```r
dim(surveys_complete)
```

```
## [1] 30463    13
```

Now that our data set is ready, we can save it as a CSV file in our `data` folder.

```r
write_csv(surveys_complete, path = "data/surveys_complete.csv")
```

```
## Warning: The `path` argument of `write_csv()` is deprecated as of readr 1.4.0.
## i Please use the `file` argument instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```r
if (!dir.exists("data")) dir.create("data")
write_csv(surveys_complete, path = "data/surveys_complete.csv")
```

6. Pivoting with `pivot_longer()` and `pivot_wider()`

There are three rules defining a "Tidy" dataset:

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

In `surveys`, the rows of `surveys` contain the values of variables associated with each record, e.g., the weight or sex of each animal. What if instead of comparing records, we wanted to compare the different mean weight of each genus between plots? (Ignoring `plot_type` for simplicity).

We'd need to create a new table where each row is comprised of values of variables associated with each plot. In practical terms this means the values in `genus` would become the names of columns and the cells would contain the values of the mean weight observed on each plot.

Having created a new table, it is then straightforward to explore the relationship between the weight of different genera within, and between, the plots. The key point here is that we have **pivoted** the data according to the observations of interest: average genus weight per plot instead of recordings per date.

The inverse procedure would be to pivot column names into values of a variable.

We can do both these of operations with two `tidyr` functions, namely `pivot_wider()` and `pivot_longer()`.

6.1 Pivot Wider

`pivot_wider()` takes three principal arguments:

1. the data
2. *names_from*: the column whose values will become new column names.

3. *values_from*: the column whose values will fill the new columns.

Further arguments include `fill` which if set fills in missing values with the specified value.

Let's use `pivot_wider()` to transform `surveys` to find the mean weight of each genus in each plot over the entire survey period.

We first use `filter()`, `group_by()` and `summarise()` to filter our observations and variables of interest, and create a new variable for the `mean_weight`.

```r
surveys_gw <- surveys %>%
  filter(!is.na(weight)) %>%
  group_by(plot_id, genus) %>%
  summarise(mean_weight = mean(weight))
```

```
## `summarise()` has grouped output by 'plot_id'. You can override using the
## `.groups` argument.
```

```r
str(surveys_gw)
```

```
## gropd_df [196 x 3] (S3: grouped_df/tbl_df/tbl/data.frame)
##  $ plot_id    : num [1:196] 1 1 1 1 1 1 1 1 2 2 ...
##  $ genus      : chr [1:196] "Baiomys" "Chaetodipus" "Dipodomys" "Neotoma" ...
##  $ mean_weight: num [1:196] 7 22.2 60.2 156.2 27.7 ...
##  - attr(*, "groups")= tibble [24 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ plot_id: num [1:24] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ .rows  : list<int> [1:24]
##   .. ..$ : int [1:8] 1 2 3 4 5 6 7 8
##   .. ..$ : int [1:9] 9 10 11 12 13 14 15 16 17
##   .. ..$ : int [1:9] 18 19 20 21 22 23 24 25 26
##   .. ..$ : int [1:8] 27 28 29 30 31 32 33 34
##   .. ..$ : int [1:9] 35 36 37 38 39 40 41 42 43
##   .. ..$ : int [1:8] 44 45 46 47 48 49 50 51
##   .. ..$ : int [1:7] 52 53 54 55 56 57 58
##   .. ..$ : int [1:7] 59 60 61 62 63 64 65
##   .. ..$ : int [1:8] 66 67 68 69 70 71 72 73
##   .. ..$ : int [1:7] 74 75 76 77 78 79 80
##   .. ..$ : int [1:8] 81 82 83 84 85 86 87 88
##   .. ..$ : int [1:8] 89 90 91 92 93 94 95 96
##   .. ..$ : int [1:8] 97 98 99 100 101 102 103 104
##   .. ..$ : int [1:8] 105 106 107 108 109 110 111 112
##   .. ..$ : int [1:8] 113 114 115 116 117 118 119 120
##   .. ..$ : int [1:7] 121 122 123 124 125 126 127
##   .. ..$ : int [1:8] 128 129 130 131 132 133 134 135
##   .. ..$ : int [1:9] 136 137 138 139 140 141 142 143 144
##   .. ..$ : int [1:9] 145 146 147 148 149 150 151 152 153
##   .. ..$ : int [1:10] 154 155 156 157 158 159 160 161 162 163
##   .. ..$ : int [1:9] 164 165 166 167 168 169 170 171 172
##   .. ..$ : int [1:8] 173 174 175 176 177 178 179 180
##   .. ..$ : int [1:8] 181 182 183 184 185 186 187 188
##   .. ..$ : int [1:8] 189 190 191 192 193 194 195 196
##   .. ..@ ptype: int(0)
##   ..- attr(*, ".drop")= logi TRUE
```

This yields `surveys_gw` where the observations for each plot are spread across multiple rows, 196 observations of 3 variables.

Using `spread()` to key on `genus` with values from `mean_weight` this becomes 24 observations of 11 variables, one row for each plot.

```r
surveys_wide <- surveys_gw |>
  pivot_wider(names_from = "genus",
              values_from = "mean_weight")

str(surveys_wide)
```

```
## gropd_df [24 x 11] (S3: grouped_df/tbl_df/tbl/data.frame)
##  $ plot_id        : num [1:24] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Baiomys        : num [1:24] 7 6 8.61 NA 7.75 ...
##  $ Chaetodipus    : num [1:24] 22.2 25.1 24.6 23 18 ...
##  $ Dipodomys      : num [1:24] 60.2 55.7 52 57.5 51.1 ...
##  $ Neotoma        : num [1:24] 156 169 158 164 190 ...
##  $ Onychomys      : num [1:24] 27.7 26.9 26 28.1 27 ...
##  $ Perognathus    : num [1:24] 9.62 6.95 7.51 7.82 8.66 ...
##  $ Peromyscus     : num [1:24] 22.2 22.3 21.4 22.6 21.2 ...
##  $ Reithrodontomys: num [1:24] 11.4 10.7 10.5 10.3 11.2 ...
##  $ Sigmodon       : num [1:24] NA 70.9 65.6 82 82.7 ...
##  $ Spermophilus   : num [1:24] NA NA NA NA NA NA NA NA NA NA ...
##  - attr(*, "groups")= tibble [24 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ plot_id: num [1:24] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ .rows  : list<int> [1:24]
##   .. ..$ : int 1
##   .. ..$ : int 2
##   .. ..$ : int 3
##   .. ..$ : int 4
##   .. ..$ : int 5
##   .. ..$ : int 6
##   .. ..$ : int 7
##   .. ..$ : int 8
##   .. ..$ : int 9
##   .. ..$ : int 10
##   .. ..$ : int 11
##   .. ..$ : int 12
##   .. ..$ : int 13
##   .. ..$ : int 14
##   .. ..$ : int 15
##   .. ..$ : int 16
##   .. ..$ : int 17
##   .. ..$ : int 18
##   .. ..$ : int 19
##   .. ..$ : int 20
##   .. ..$ : int 21
##   .. ..$ : int 22
##   .. ..$ : int 23
##   .. ..$ : int 24
##   .. ..@ ptype: int(0)
##   ..- attr(*, ".drop")= logi TRUE
```

We could now plot comparisons between the weight of genera in different plots, although we may wish to fill in the missing values first.

```
surveys_gw |>
  pivot_wider(names_from = "genus",
              values_from = "mean_weight",
              values_fill = 0)
```

```
## # A tibble: 24 x 11
## # Groups:   plot_id [24]
##    plot_id Baiomys Chaetodipus Dipodomys Neotoma Onychomys Perognathus
##      <dbl>   <dbl>       <dbl>     <dbl>   <dbl>     <dbl>       <dbl>
##  1       1    7           22.2      60.2    156.      27.7        9.62
##  2        2    6            25.1      55.7    169.      26.9        6.95
##  3        3    8.61         24.6      52.0    158.      26.0        7.51
##  4        4    0            23.0      57.5    164.      28.1        7.82
##  5        5    7.75         18.0      51.1    190.      27.0        8.66
##  6        6    0            24.9      58.6    180.      25.9        7.81
##  7        7    0            19.9      57.4    170.      23.6        7
##  8        8    0            20.5      59.4    134.      25.9        7.06
##  9        9    0            18.9      57.5    162.      27.5        7.37
## 10       10    0            22.3      51.8    190       28.7        0
## # i 14 more rows
## # i 4 more variables: Peromyscus <dbl>, Reithrodontomys <dbl>, Sigmodon <dbl>,
## #   Spermophilus <dbl>
```

6.2 Pivot Longer

The converse situation could occur if we had been provided with data in then form of `surveys_spread`, where the genus names are column names, but we wish to treat them as values of a genus variable instead.

In this situation we are gathering the column names and turning them into a pair of new variables. One variable represents the column names as values in a new column, and the other variable contains the values previously found across all the original columns.

`pivot_longer()` takes four principal arguments:

1. the data
2. *cols*: the columns that we want to pivot into longer format.
3. *names_to*: the name of the new column for storing what were previously column names.
4. *values_to*: the name of another new column for storing the values that were in previously found in the *cols*.

To recreate `surveys_gw` from `surveys_spread` we would create a new column called `genus` and another called `mean_weight` and use all columns except `plot_id` for the key variable. Here we exclude `plot_id` from being `pivoted_longer()`.

```
surveys_long <- surveys_wide |>
  pivot_longer(cols = -plot_id,
               names_to = "genus",
               values_to = "mean_weight")

str(surveys_long)
```

```
## gropd_df [240 x 3] (S3: grouped_df/tbl_df/tbl/data.frame)
##  $ plot_id    : num [1:240] 1 1 1 1 1 1 1 1 1 1 ...
##  $ genus      : chr [1:240] "Baiomys" "Chaetodipus" "Dipodomys" "Neotoma" ...
##  $ mean_weight: num [1:240] 7 22.2 60.2 156.2 27.7 ...
##  - attr(*, "groups")= tibble [24 x 2] (S3: tbl_df/tbl/data.frame)
##   ..$ plot_id: num [1:24] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ .rows  : list<int> [1:24]
##   .. ..$ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ : int [1:10] 11 12 13 14 15 16 17 18 19 20
##   .. ..$ : int [1:10] 21 22 23 24 25 26 27 28 29 30
##   .. ..$ : int [1:10] 31 32 33 34 35 36 37 38 39 40
##   .. ..$ : int [1:10] 41 42 43 44 45 46 47 48 49 50
##   .. ..$ : int [1:10] 51 52 53 54 55 56 57 58 59 60
##   .. ..$ : int [1:10] 61 62 63 64 65 66 67 68 69 70
##   .. ..$ : int [1:10] 71 72 73 74 75 76 77 78 79 80
##   .. ..$ : int [1:10] 81 82 83 84 85 86 87 88 89 90
##   .. ..$ : int [1:10] 91 92 93 94 95 96 97 98 99 100
##   .. ..$ : int [1:10] 101 102 103 104 105 106 107 108 109 110
##   .. ..$ : int [1:10] 111 112 113 114 115 116 117 118 119 120
##   .. ..$ : int [1:10] 121 122 123 124 125 126 127 128 129 130
##   .. ..$ : int [1:10] 131 132 133 134 135 136 137 138 139 140
##   .. ..$ : int [1:10] 141 142 143 144 145 146 147 148 149 150
##   .. ..$ : int [1:10] 151 152 153 154 155 156 157 158 159 160
##   .. ..$ : int [1:10] 161 162 163 164 165 166 167 168 169 170
##   .. ..$ : int [1:10] 171 172 173 174 175 176 177 178 179 180
##   .. ..$ : int [1:10] 181 182 183 184 185 186 187 188 189 190
##   .. ..$ : int [1:10] 191 192 193 194 195 196 197 198 199 200
##   .. ..$ : int [1:10] 201 202 203 204 205 206 207 208 209 210
##   .. ..$ : int [1:10] 211 212 213 214 215 216 217 218 219 220
##   .. ..$ : int [1:10] 221 222 223 224 225 226 227 228 229 230
##   .. ..$ : int [1:10] 231 232 233 234 235 236 237 238 239 240
##   .. ..@ ptype: int(0)
##   ..- attr(*, ".drop")= logi TRUE
```

Note that now the `NA` genera are included in the pivoted format.

Pivoting wider and then longer can be a useful way to balance out a dataset so that every replicate has the same composition.

We could also have used a specification for what columns to include. This can be useful if you have a large number of identifying columns, and it's easier to specify what to pivot than what to leave alone. And if the columns are directly adjacent, we don't even need to list them all out - just use the `:` operator!

```
surveys_wide |>
  pivot_longer(cols = Baiomys:Spermophilus,
               names_to = "genus",
               values_to = "mean_weight") |>
  head()
```

```
## # A tibble: 6 x 3
## # Groups:   plot_id [1]
##   plot_id genus       mean_weight
##     <dbl> <chr>             <dbl>
## 1       1 Baiomys            7
## 2       1 Chaetodipus       22.2
## 3       1 Dipodomys         60.2
## 4       1 Neotoma          156.
## 5       1 Onychomys         27.7
## 6       1 Perognathus        9.62
```

6.3 Challenges for Pivoting

Challenge #6 Pivot the `surveys` data frame with `year` as columns, `plot_id` as rows, and the number of genera per plot as the values. You will need to summarise before pivoting, and use the function `n_distinct()` to get the number of unique genera within a particular chunk of data. It's a powerful function! See `?n_distinct` for more.

```
surveys_wide_genera <- surveys %>%
  group_by(plot_id, year) %>%
  summarise(n_genera = n_distinct(genus)) %>%
  pivot_wider(names_from = "year",
              values_from = "n_genera")
```

```
## `summarise()` has grouped output by 'plot_id'. You can override using the
## `.groups` argument.
```

```r
head(surveys_wide_genera)
```

```
## # A tibble: 6 x 27
## # Groups:   plot_id [6]
##   plot_id `1977` `1978` `1979` `1980` `1981` `1982` `1983` `1984` `1985` `1986`
##     <dbl>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>
## 1       1      2      3      4      7      5      6      7      6      4      3
## 2       2      6      6      6      8      5      9      9      9      6      4
## 3       3      5      6      4      6      6      8     10     11      7      6
## 4       4      4      4      3      4      5      4      6      3      4      3
## 5       5      4      3      2      5      4      6      7      7      3      1
## 6       6      3      4      3      4      5      9      9      7      5      6
## # i 16 more variables: `1987` <int>, `1988` <int>, `1989` <int>, `1990` <int>,
## #   `1991` <int>, `1992` <int>, `1993` <int>, `1994` <int>, `1995` <int>,
## #   `1996` <int>, `1997` <int>, `1998` <int>, `1999` <int>, `2000` <int>,
## #   `2001` <int>, `2002` <int>
```

Challenge #7 Now take that data frame and `pivot_longer()` it back so each row is a unique `plot_id` by `year` combination.

```r
surveys_long_genera <- surveys_wide_genera %>%
  pivot_longer(cols = -plot_id,
               names_to = "year",
               values_to = "ngenera")

head(surveys_long_genera)
```

```
## # A tibble: 6 x 3
## # Groups:   plot_id [1]
##   plot_id year  ngenera
##     <dbl> <chr>   <int>
## 1       1 1977        2
## 2       1 1978        3
## 3       1 1979        4
## 4       1 1980        7
## 5       1 1981        5
## 6       1 1982        6
```

Challenge #8 The `surveys` data set has two measurement columns: `hindfoot_length` and `weight`. This makes it difficult to do things like look at the relationship between mean values of each measurement per year in different plot types. Let's walk through a common solution for this type of problem. First, use `pivot_longer()` to create a dataset where we have a key column called `measurement` and a `value` column that takes on the value of either `hindfoot_length` or `weight`. *Hint*: You'll need to specify which columns are being gathered.

```r
surveys_long <- surveys %>%
  pivot_longer(cols = c(hindfoot_length, weight), # Tidyselect syntax
               names_to = "measurement",
               values_to = "value")
```

Challenge #9 With this new data set, calculate the average of each `measurement` in each `year` for each different `plot_type`. Then `pivot_wider()` them into a data set with a column for `hindfoot_length` and `weight`. *Hint*: You only need to specify the key and value columns for `pivot_wider()`.

```
surveys_long %>%
  group_by(year, measurement, plot_type) %>%
  summarise(mean_value = mean(value, na.rm=TRUE)) %>%
  pivot_wider(names_from = "measurement",
              values_from = "mean_value")
```

```
## `summarise()` has grouped output by 'year', 'measurement'. You can override
## using the `.groups` argument.
```

```
## # A tibble: 130 x 4
## # Groups:   year [26]
##     year plot_type                 hindfoot_length weight
##    <dbl> <chr>                               <dbl>  <dbl>
##  1  1977 Control                              36.1   50.4
##  2  1977 Long-term Krat Exclosure             33.7   34.8
##  3  1977 Rodent Exclosure                     39.1   48.2
##  4  1977 Short-term Krat Exclosure            35.8   41.3
##  5  1977 Spectab exclosure                    37.2   47.1
##  6  1978 Control                              38.1   70.8
##  7  1978 Long-term Krat Exclosure             22.6   35.9
##  8  1978 Rodent Exclosure                     37.8   67.3
##  9  1978 Short-term Krat Exclosure            36.9   63.8
## 10  1978 Spectab exclosure                    42.3   80.1
## # i 120 more rows
```