Introduction to the `tidyverse`

Ellen Bledsoe

2022-09-22

2-dimensional Data and the `tidyverse`

What is the `tidyverse`?

Different programming languages have different syntax (language structure). The `tidyverse` is a package (more accurately, a set of packages) offered in R that all have similar goals and a unified syntax designed to work particularly well with 2-dimensional data.

Until now, all of the coding we have done is in the original R language, which is often called "base R." The syntax in the `tidyverse` is often pretty different from base R. Both are useful, and many people often combine them, which is why we start with base R.

Explore the tidyverse

If you want to learn more about the `tidyverse`, head over to www.tidyverse.org and browse the site. Below is a brief summary of *some* of the packages I think you might find the most useful.

- `tidyr`: creating data that is consistent in form/shape
- `dplyr`: creating data that is clean, easily wrangled, and summarized
- `ggplot2`: publication-worthy plots using The Grammar of Graphics
- `tibble`: data frames but better!
- `readr`: fast and friendly ways to read data into R
- `stringr`: easy manipulation of strings (character data)
- `lubridate`: easy manipulation of time and date values

More resources:

- RStudio Cheatsheets
  - specifically the `dplyr` and `tidyr` cheat sheets, but there are many more!
- Data Carpentry lesson
- Effectively using the tidyverse

Practice with the `tidyverse`

Set Up

I have already set up these RStudio Cloud projects so that the `tidyverse` is installed. Each time we want to use the `tidyverse`, however, we need to "load" it into our workspace. We do this with the `library()` function.

We are also going use the `penguins` dataset from the `palmerpenguin` package. We will need to load that package, as well.

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.7
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.1.1     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(palmerpenguins)
```

When you load the `tidyverse` library, you'll see some "conflicts." Don't panic! That's normal. Those conflicts are telling us that certain functions in `dplyr` are now overriding some functions in base R with the same name.

Penguins

Let's remind ourselves what the `penguins` dataset looks like.

```
head(penguins)
```

```
## # A tibble: 6 x 8
##   species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g sex
##   <fct>   <fct>           <dbl>         <dbl>            <int>       <int> <fct>
## 1 Adelie  Torge~           39.1          18.7              181        3750 male
## 2 Adelie  Torge~           39.5          17.4              186        3800 fema~
## 3 Adelie  Torge~           40.3          18                195        3250 fema~
## 4 Adelie  Torge~           NA            NA                 NA          NA <NA>
## 5 Adelie  Torge~           36.7          19.3              193        3450 fema~
## 6 Adelie  Torge~           39.3          20.6              190        3650 male
## # ... with 1 more variable: year <int>
```

```
colnames(penguins)
```

```
## [1] "species"           "island"            "bill_length_mm"
## [4] "bill_depth_mm"     "flipper_length_mm" "body_mass_g"
## [7] "sex"               "year"
```

```
str(penguins)
```

```
## tibble [344 x 8] (S3: tbl_df/tbl/data.frame)
##  $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ bill_length_mm   : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
##  $ bill_depth_mm    : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
##  $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
##  $ body_mass_g      : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
##  $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
##  $ year             : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

```
view(penguins)

# glimpse() is from the dplyr package
glimpse(penguins)
```

```
## Rows: 344
## Columns: 8
## $ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, Adel~
## $ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torgerse~
## $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, ~
## $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, ~
## $ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186~
## $ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, ~
## $ sex               <fct> male, female, female, NA, female, male, female, male~
## $ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007~
```

The `tidyverse` converts 2D data into something called a tibble! For our intents and purposes, it is basically the same as a data frame (and I'll probably call it a data frame, in reality).

Useful functions from `dplyr`

`dplyr` is one of the most useful packages in the `tidyverse`. I use it pretty much every single time I code. If we break down the package name `dplyr`, we get d-ply-r. The "d" stands for data (and data frame), "ply" means to work with and push the data into a format with which we can work, and the "r" means that we are doing this in R.

`select()`ing columns

Let's use our first function, `select()`. Select allows us to pick out specific columns from our data. You can use names or their position in the data frame.

First, let's remind ourselves how we would accomplish this in base R.

```
penguins[, 1:3]
```

```
## # A tibble: 344 x 3
##    species island    bill_length_mm
##    <fct>   <fct>              <dbl>
```

```
##  1 Adelie  Torgersen           39.1
##  2 Adelie  Torgersen           39.5
##  3 Adelie  Torgersen           40.3
##  4 Adelie  Torgersen           NA
##  5 Adelie  Torgersen           36.7
##  6 Adelie  Torgersen           39.3
##  7 Adelie  Torgersen           38.9
##  8 Adelie  Torgersen           39.2
##  9 Adelie  Torgersen           34.1
## 10 Adelie  Torgersen           42
## # ... with 334 more rows
```

The `select()` function does the same thing but with more power (and, in my opinion, more easily). The first argument in the function is the data frame. Any following arguments are the columns we want to select.

```
# first argument is the data frame, then the columns
select(penguins, bill_length_mm)
```

```
## # A tibble: 344 x 1
##    bill_length_mm
##             <dbl>
##  1           39.1
##  2           39.5
##  3           40.3
##  4           NA
##  5           36.7
##  6           39.3
##  7           38.9
##  8           39.2
##  9           34.1
## 10           42
## # ... with 334 more rows
```

```
# multiple columns
select(penguins, species, island, bill_length_mm)
```

```
## # A tibble: 344 x 3
##    species island    bill_length_mm
##    <fct>   <fct>              <dbl>
##  1 Adelie  Torgersen           39.1
##  2 Adelie  Torgersen           39.5
##  3 Adelie  Torgersen           40.3
##  4 Adelie  Torgersen           NA
##  5 Adelie  Torgersen           36.7
##  6 Adelie  Torgersen           39.3
##  7 Adelie  Torgersen           38.9
##  8 Adelie  Torgersen           39.2
##  9 Adelie  Torgersen           34.1
## 10 Adelie  Torgersen           42
## # ... with 334 more rows
```

```
select(penguins, species:bill_length_mm)
```

```
## # A tibble: 344 x 3
##    species island    bill_length_mm
##    <fct>   <fct>              <dbl>
##  1 Adelie  Torgersen           39.1
##  2 Adelie  Torgersen           39.5
##  3 Adelie  Torgersen           40.3
##  4 Adelie  Torgersen           NA
##  5 Adelie  Torgersen           36.7
##  6 Adelie  Torgersen           39.3
##  7 Adelie  Torgersen           38.9
##  8 Adelie  Torgersen           39.2
##  9 Adelie  Torgersen           34.1
## 10 Adelie  Torgersen           42
## # ... with 334 more rows
```

```
select(penguins, -island)
```

```
## # A tibble: 344 x 7
##    species bill_length_mm bill_depth_mm flipper_length_~ body_mass_g sex    year
##    <fct>            <dbl>         <dbl>            <int>       <int> <fct> <int>
##  1 Adelie            39.1          18.7              181        3750 male   2007
##  2 Adelie            39.5          17.4              186        3800 fema~  2007
##  3 Adelie            40.3          18                195        3250 fema~  2007
##  4 Adelie            NA            NA                 NA          NA <NA>   2007
##  5 Adelie            36.7          19.3              193        3450 fema~  2007
##  6 Adelie            39.3          20.6              190        3650 male   2007
##  7 Adelie            38.9          17.8              181        3625 fema~  2007
##  8 Adelie            39.2          19.6              195        4675 male   2007
##  9 Adelie            34.1          18.1              193        3475 <NA>   2007
## 10 Adelie            42            20.2              190        4250 <NA>   2007
## # ... with 334 more rows
```

You might have noticed that we haven't put any column names in quotations, unlike what we did with selecting columns by name in base R. This is one quirk of the `tidyverse` to which you will need to pay special attention. We *usually* will not need to put column names in quotations.

Let's practice! Write a line of code to select the columns with the following information from `penguins`: species, body mass, sex, and year.

```
select(penguins, body_mass_g, sex, year)
```

```
## # A tibble: 344 x 3
##    body_mass_g sex     year
##          <int> <fct>  <int>
##  1        3750 male    2007
##  2        3800 female  2007
##  3        3250 female  2007
##  4          NA <NA>    2007
##  5        3450 female  2007
```

```
##  6        3650 male    2007
##  7        3625 female  2007
##  8        4675 male    2007
##  9        3475 <NA>    2007
## 10        4250 <NA>    2007
## # ... with 334 more rows
```

`filter()`ing rows

`filter()` allows you filter rows by certain conditions. Recall that we did this a bit with base R.

```
# base R
penguins[penguins$island == "Torgersen", ]
```

```
## # A tibble: 52 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
##  4 Adelie  Torgersen           NA            NA                  NA          NA
##  5 Adelie  Torgersen           36.7          19.3               193        3450
##  6 Adelie  Torgersen           39.3          20.6               190        3650
##  7 Adelie  Torgersen           38.9          17.8               181        3625
##  8 Adelie  Torgersen           39.2          19.6               195        4675
##  9 Adelie  Torgersen           34.1          18.1               193        3475
## 10 Adelie  Torgersen           42            20.2               190        4250
## # ... with 42 more rows, and 2 more variables: sex <fct>, year <int>
```

The code above is, in my opinion, a bit unwieldy. Filter feels more intuitive. We still need the double equal signs, though!

```
# filter
filter(penguins, island == "Torgersen")
```

```
## # A tibble: 52 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
##  4 Adelie  Torgersen           NA            NA                  NA          NA
##  5 Adelie  Torgersen           36.7          19.3               193        3450
##  6 Adelie  Torgersen           39.3          20.6               190        3650
##  7 Adelie  Torgersen           38.9          17.8               181        3625
##  8 Adelie  Torgersen           39.2          19.6               195        4675
##  9 Adelie  Torgersen           34.1          18.1               193        3475
## 10 Adelie  Torgersen           42            20.2               190        4250
## # ... with 42 more rows, and 2 more variables: sex <fct>, year <int>
```

```
# easy to write multiple conditions and to chain stuff together
filter(penguins, island == "Torgersen" & year > 2008)
```

```
## # A tibble: 16 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           38.6          17                188        2900
##  2 Adelie  Torgersen           37.3          20.5              199        3775
##  3 Adelie  Torgersen           35.7          17                189        3350
##  4 Adelie  Torgersen           41.1          18.6              189        3325
##  5 Adelie  Torgersen           36.2          17.2              187        3150
##  6 Adelie  Torgersen           37.7          19.8              198        3500
##  7 Adelie  Torgersen           40.2          17                176        3450
##  8 Adelie  Torgersen           41.4          18.5              202        3875
##  9 Adelie  Torgersen           35.2          15.9              186        3050
## 10 Adelie  Torgersen           40.6          19                199        4000
## 11 Adelie  Torgersen           38.8          17.6              191        3275
## 12 Adelie  Torgersen           41.5          18.3              195        4300
## 13 Adelie  Torgersen           39            17.1              191        3050
## 14 Adelie  Torgersen           44.1          18                210        4000
## 15 Adelie  Torgersen           38.5          17.9              190        3325
## 16 Adelie  Torgersen           43.1          19.2              197        3500
## # ... with 2 more variables: sex <fct>, year <int>
```

```
# worth noting here that we haven't saved any of this. We need to write to a new object.
torg2008 <- filter(penguins, island == "Torgersen" & year >= 2008)
```

Let's practice using `select()` and `filter()` Using the `penguins` data frame, write a small set of code that does the following:

1. Slims down the full data frame to one that contains the species, bill length, and sex. Assign this to an object called `slim`.
2. Filters the data for only male penguins with bills greater than 20 mm in length.
3. Name this new data frame `male20`

```
slim <- select(penguins, species, bill_length_mm, sex)
male20 <- filter(slim, sex == "male", bill_length_mm > 20)
```

The pipe %>%

You can use the pipe operator to chain `tidyverse` functions together. You can think of the pipe as automatically sending the output from the first line (left side of the pipe) into the next line as the input (right side of the pipe).

This is helpful for a lot of reasons, including:

1. removing the clutter of creating a lot of intermediate objects in your work space, which reduces the chance of errors caused by using the wrong input object
2. makes things more human-readable (in addition to computer-readable)

The shortcut for the pipe is `Ctrl + Shift + M` (Windows) or `Cmd + Shift + M` (Mac).

Let's recreate the `male20` dataframe that we created above; this time, however, we will use the pipe!

```
male20 <- penguins %>%
  select(species, bill_length_mm, sex) %>%
  filter(sex == "male", bill_length_mm > 20)
```

Group Challenge

Using pipes, subset the `penguins` data to include only Adelie penguins from the year 2007. Your final data frame should only have the species, sex, and year columns.

```
penguins %>%
  filter(species == "Adelie", year == 2007) %>%
  select(species, sex, year)
```

```
## # A tibble: 50 x 3
##    species sex     year
##    <fct>   <fct>  <int>
##  1 Adelie  male    2007
##  2 Adelie  female  2007
##  3 Adelie  female  2007
##  4 Adelie  <NA>    2007
##  5 Adelie  female  2007
##  6 Adelie  male    2007
##  7 Adelie  female  2007
##  8 Adelie  male    2007
##  9 Adelie  <NA>    2007
## 10 Adelie  <NA>    2007
## # ... with 40 more rows
```

More useful functions!

Creating new variables with `mutate()`

Sometimes our data doesn't have our data in exactly the format we want. For example, we might want our temperature data in Fahrenheit instead of Celsius or our millimeter measurements in centimeters.

The `tidyverse` has a function called `mutate()` that lets us create a new column. Often, we want to apply a function to the entire column or perform some type of calculation. The `mutate()` function allows us to do this.

```
penguins %>%
  mutate(bill_length_cm = bill_length_mm / 10)
```

```
## # A tibble: 344 x 9
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
```

```
##  4 Adelie  Torgersen           NA            NA                  NA          NA
##  5 Adelie  Torgersen           36.7          19.3               193        3450
##  6 Adelie  Torgersen           39.3          20.6               190        3650
##  7 Adelie  Torgersen           38.9          17.8               181        3625
##  8 Adelie  Torgersen           39.2          19.6               195        4675
##  9 Adelie  Torgersen           34.1          18.1               193        3475
## 10 Adelie  Torgersen           42            20.2               190        4250
## # ... with 334 more rows, and 3 more variables: sex <fct>, year <int>,
## #   bill_length_cm <dbl>
```

```
# remember, we would have to create a new object to save this new column!
```

Understanding data through `summarize()`

Like we have talked about in previous classes, some of the best ways for us to understand our data is through what we call summary statistics such as the mean, standard deviation, minimums, maximums, etc.

Fortunately, the `tidyverse` has a handy-dandy function to make this easy to do with data frames.

```
# first attempt at mean and sd of body mass
penguins %>%
  summarise(mean_body_mass = mean(body_mass_g),
            sd_body_mass = sd(body_mass_g))
```

```
## # A tibble: 1 x 2
##   mean_body_mass sd_body_mass
##            <dbl>        <dbl>
## 1             NA           NA
```

Wait a second! Those are some weird values!

It is important to note that if any of the values in the column that you are trying to summarize are missing, you might get some wonky values, like you did above. Fortunately, `mean()` and `sd()` and many other functions have an argument to remove the missing values: `na.rm = TRUE`

```
penguins %>%
  summarise(mean_body_mass = mean(body_mass_g, na.rm = TRUE),
            sd_body_mass = sd(body_mass_g, na.rm = TRUE))
```

```
## # A tibble: 1 x 2
##   mean_body_mass sd_body_mass
##            <dbl>        <dbl>
## 1          4202.         802.
```

Split, Apply, Combine with `group_by()`

One common way we analyze data is through something we call the "split, apply, combine" approach. This means that we:

- *split* data up into groups via some type of categorization
- *apply* some type of analysis to each group independently and

- *combine* the data back together

The `group_by()` function lets us do this. It is most often used in combination with `mutate()` or `summarize()`.

For example, we can use this method to calculate the mean body mass for males and females of each species instead of the overall mean of the entire dataset

```
penguins %>%
  group_by(species, sex) %>%
  summarise(mean_body_mass = mean(body_mass_g, na.rm = TRUE),
            sd_body_mass = sd(body_mass_g, na.rm = TRUE))
```

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

```
## # A tibble: 8 x 4
## # Groups:   species [3]
##   species   sex    mean_body_mass sd_body_mass
##   <fct>     <fct>           <dbl>        <dbl>
## 1 Adelie    female          3369.         269.
## 2 Adelie    male            4043.         347.
## 3 Adelie    <NA>            3540          477.
## 4 Chinstrap female          3527.         285.
## 5 Chinstrap male            3939.         362.
## 6 Gentoo    female          4680.         282.
## 7 Gentoo    male            5485.         313.
## 8 Gentoo    <NA>            4588.         338.
```

Let's practice!

Practice using the combination of `group_by()` and `summarize()` to calculate the minimum (`min()`) and maximum (`max()`) average flipper length per island. Save this data frame as `flipper_min_max`

```
flipper_min_max <- penguins %>%
  group_by(island) %>%
  summarize(min_flipper = min(flipper_length_mm, na.rm = TRUE),
            max_flipper = max(flipper_length_mm, na.rm = TRUE))
flipper_min_max
```

```
## # A tibble: 3 x 3
##   island    min_flipper max_flipper
##   <fct>           <int>       <int>
## 1 Biscoe            172         231
## 2 Dream             178         212
## 3 Torgersen         176         210
```

Already accomplished this task? Try to figure out how you can keep the "species" column in the final data frame.

```
penguins %>%
  group_by(island, species) %>%
  summarize(min_flipper = min(flipper_length_mm, na.rm = TRUE),
            max_flipper = max(flipper_length_mm, na.rm = TRUE))
```

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

```
## # A tibble: 5 x 4
## # Groups:   island [3]
##   island    species   min_flipper max_flipper
##   <fct>     <fct>           <int>       <int>
## 1 Biscoe    Adelie            172         203
## 2 Biscoe    Gentoo            203         231
## 3 Dream     Adelie            178         208
## 4 Dream     Chinstrap         178         212
## 5 Torgersen Adelie            176         210
```

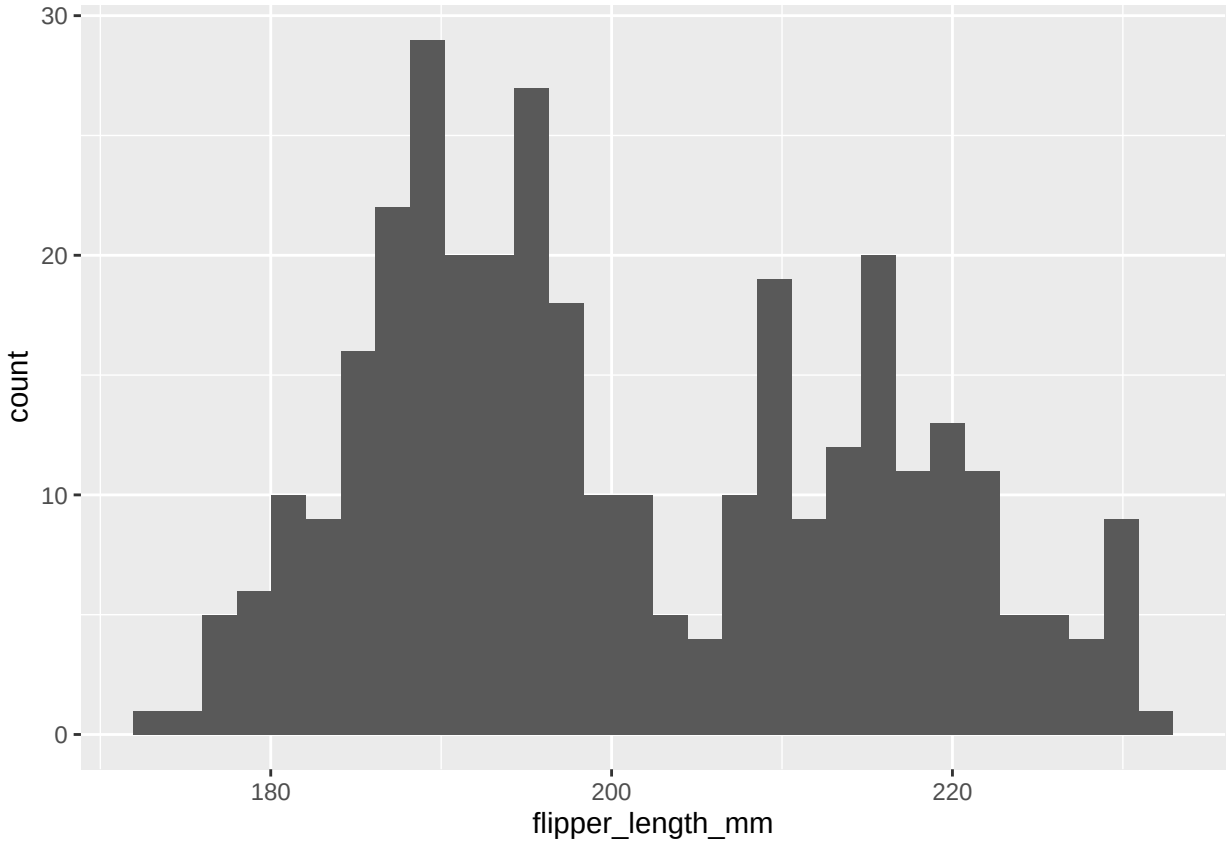
Making Plots with ggplot2

One of my favorite parts of using the `tidyverse` is making plots with the `ggplot2` package. We aren't going to delve into how this works too much right now because we don't end up making too many plots in this course, but I wanted to include some code for making a histogram to give you an idea of how this works.

```
ggplot(penguins, aes(x = flipper_length_mm)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 2 rows containing non-finite values (stat_bin).
```



Here's what is happening in the code above.

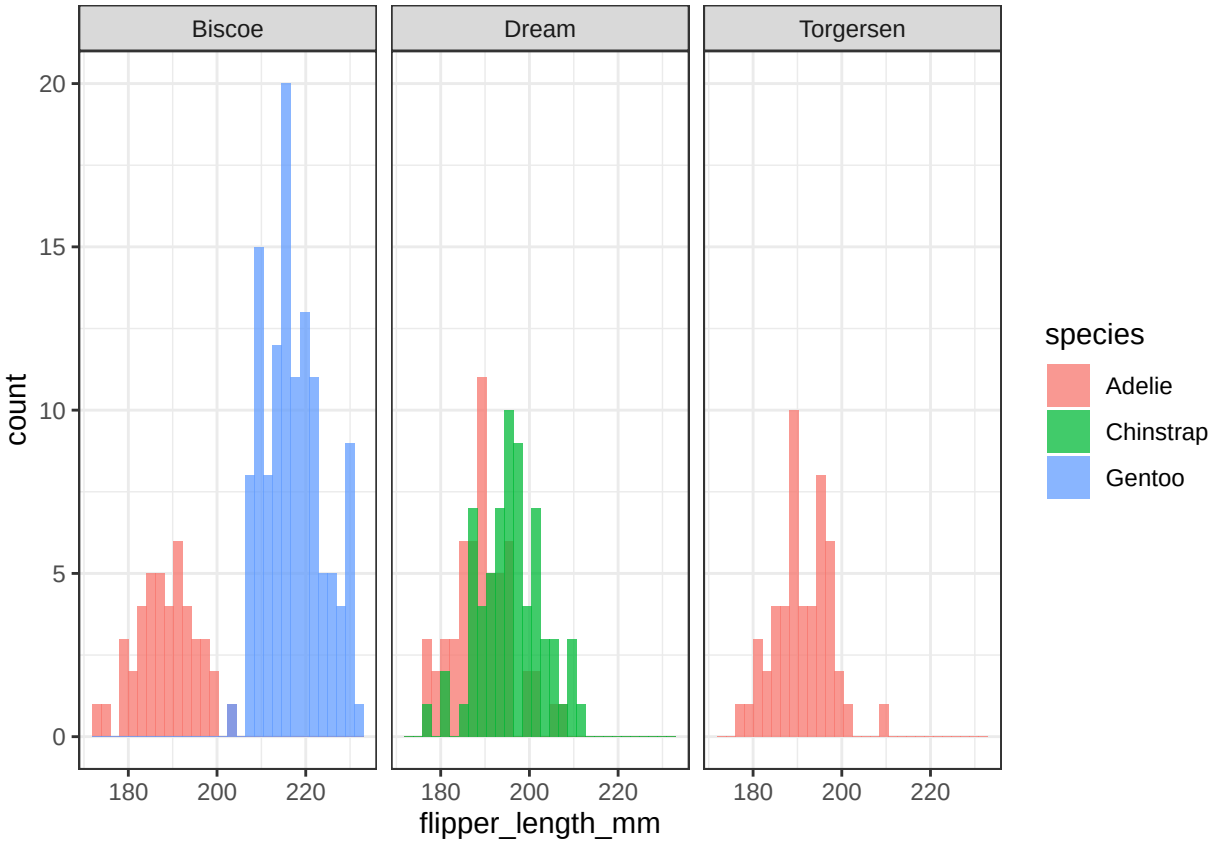
- `ggplot()` is the function to make a plot
- after telling the function what data to use (`penguins`), we need to tell the function the "mappings", or which variables (columns) should be associated with which axes or other elements of the plot. Here, I've assigned the "flipper_length_mm" column to the x-axis. When plotting a histogram, `ggplot()` automatically knows that the y-axis will be frequency, so we do not need to specify
- we use the `+` symbol to add a new "layer" to the plot
- here, I've added a histogram layer with the `geom_histogram()` function, telling `ggplot2` to make a histogram plot

This is definitely more complicated than making a basic histogram in base R, so why is it so great? Well, with just a few modifications, you can make beautiful, multi-layered plots that would be much more complicated to produce in base R. For example, check out resulting plot from the code below.

```
ggplot(penguins, aes(flipper_length_mm, fill = species)) +
  geom_histogram(position = "identity", alpha = 0.75) +
  facet_wrap( ~ island) +
  theme_bw()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 2 rows containing non-finite values (stat_bin).
```



A few other things to know

Below, I'm including some additional functions that you might find helpful! You *will NOT* be tested on these functions below, but they often come in handy, and we might use them down the road.

1. We can use the `is.na()` function to filter out `NA` values. The `!` tells R to do the opposite of what we've asked. Therefore, the code below effectively says the following:

- ask if each value in the sex column is an `NA` value or not and return a string of TRUE and FALSE
- instead of filtering out the TRUE values (meaning we would get all rows in which the value in the sex column *is* `NA`, adding the `!` symbol tells the filter function to pull all the FALSE values, meaning we pull all the rows in which the value in the sex column *is not* `NA`.

```
penguins %>%
  filter(!is.na(sex))
```

```
## # A tibble: 333 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
##  4 Adelie  Torgersen           36.7          19.3               193        3450
##  5 Adelie  Torgersen           39.3          20.6               190        3650
##  6 Adelie  Torgersen           38.9          17.8               181        3625
##  7 Adelie  Torgersen           39.2          19.6               195        4675
##  8 Adelie  Torgersen           41.1          17.6               182        3200
##  9 Adelie  Torgersen           38.6          21.2               191        3800
## 10 Adelie  Torgersen           34.6          21.1               198        4400
## # ... with 323 more rows, and 2 more variables: sex <fct>, year <int>
```

2. The `arrange()` functions lets us order our rows alphabetically or numerically.

```
penguins %>%
  arrange(body_mass_g)
```

```
## # A tibble: 344 x 8
##    species   island    bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
##    <fct>     <fct>              <dbl>         <dbl>            <int>       <int>
##  1 Chinstrap Dream               46.9          16.6              192        2700
##  2 Adelie    Biscoe              36.5          16.6              181        2850
##  3 Adelie    Biscoe              36.4          17.1              184        2850
##  4 Adelie    Biscoe              34.5          18.1              187        2900
##  5 Adelie    Dream               33.1          16.1              178        2900
##  6 Adelie    Torgersen           38.6          17                188        2900
##  7 Chinstrap Dream               43.2          16.6              187        2900
##  8 Adelie    Biscoe              37.9          18.6              193        2925
##  9 Adelie    Dream               37.5          18.9              179        2975
## 10 Adelie    Dream               37            16.9              185        3000
## # ... with 334 more rows, and 2 more variables: sex <fct>, year <int>
```

3. `count()` let's us find out how many observations we have.

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

```
## # A tibble: 8 x 3
##   species   sex        n
##   <fct>     <fct>  <int>
## 1 Adelie    female    73
## 2 Adelie    male      73
## 3 Adelie    <NA>       6
## 4 Chinstrap female    34
## 5 Chinstrap male      34
## 6 Gentoo    female    58
## 7 Gentoo    male      61
## 8 Gentoo    <NA>       5
```

4. Finally, `distinct()` allows us to find all the unique values in a column. This is usually best for categorical (character) columns.

```
penguins %>%
  distinct(species)
```

```
## # A tibble: 3 x 1
##   species
##   <fct>
## 1 Adelie
## 2 Gentoo
## 3 Chinstrap
```