BIOE286 Lab B

Introduction to Data Manipulation and More Advanced Plots in R

One of the most frequent things you will do in R is manipulate data. In this lab, we'll teach you how to use base and a package called `dplyr` to manipulate aspects of data frames, including selecting columns, filtering for certain rows, adding new columns, and summarizing data across groups. Mastering these skills will not only make you a better data analyst - it will make you a better biologist, too, as you can use data as a tool to design experiments and studies as well as catch mistakes or omissions in your datasets.

For various parts of this lab, we'll use code written by Calvin Munson for the SCRUBS R Working Group at UCSC.

Before we get started, I want to quickly teach you something that'll make your life easier in the long run. R scripts can have headers in them, which organizes your code nicely into an Outline. On a PC, Ctr+Shift+O (like the letter O, not the number 0) should bring up the Outline, or you can find a button to bring up the Outline right under the Run button in Rstudio. To make a single level header, type `# Header----------` before your code. You can put as many dashes as you want. And, you can make a subheader with 2 pound signs... `## Subeader----------`. Then you can easily go from one header to another using the outline. I'll often add a header to distinguish different plots in a manuscript, for example (`# Fig1 -----`), so that I can easily jump around between plots. Try out headers today!

1) Data manipulation

A. Set up

First, load packages from library and read in data from an online repository (requires an internet connection). Fun fact: you can use the `read.csv()` function to read in data from a URL, like a Github webpage.

For future labs, note that I am going to assume you have packages installed by skipping the `install.packages(...)` code and going straight for loading the packages using `library(...)`. If the `library(...)` line gives you an error message, it's probably because you don't have the packages installed yet. Go ahead and do that using `install.packages()` or go to the Packages tab in the lower right panel of R Studio and click "Install" to do it manually.

```
library(tidyverse)
library(readr)

urlRemote  = "https://raw.githubusercontent.com/"
pathGithub = "calvin-munson/R-DataScience-workshops/master/workshop2_enterthetidyverse/"
fileName   = "cereal.csv"

cereal=read.csv(paste0(urlRemote, pathGithub, fileName))
```

FYI, one of the annoying things about RMarkdown is that you can't line wrap easily (i.e., things like very long file names or URLs or working directories get cut off). To fix this above, I've split the URL into three useful chunks (the Github URL [`urlRemote`], the Github path [`pathGithub`] and the `fileName`), then pasted them together using the `paste0()` function within the `read.csv()` function. Ask us if this doesn't make sense!

B. Introducing the pipeline: %>%

So let's say that we want to string together multiple functions at once. This can be particularly useful when dealing with lots of data, and if we don't want to keep creating new data frames. The `%>%` operator, also known as a "pipeline", can help us with this! In RStudio the keyboard shortcut for the pipe operator `%>%` is Ctrl + Shift + M (Windows) or Cmd + Shift + M (Mac). To demonstrate, let's look at a vector of fruit names:

```
fruits = c("apple", "apple", "orange", "orange", "banana")
fruits
```

```
## [1] "apple"  "apple"  "orange" "orange" "banana"
```

How many fruits are there total?

```
length(fruits)
```

```
## [1] 5
```

Our goal is to get the number of unique fruits in this vector. We can tell that it is 3, but with larger and larger strings of data, this can be quite challenging! We can look at the unique fruits:

```
unique(fruits)
```

```
## [1] "apple"  "orange" "banana"
```

```
fruits2 = unique(fruits)
length(fruits2)
```

```
## [1] 3
```

This is one option. However with data frames, where there are multiple different arguments, this can get very very messy. So instead, we can use the `%>%` operator, which takes the data object and "pipes" it into the function that follows. For instance, `fruits %>% unique()` is the same as `unique(fruits)`.

Here is the way to do it with two different pipes. This takes the "fruits" object and feeds it into the unique() function. This produces an output, which the second pipe takes and feeds into the length() function. It saves us a step!

```
fruits %>%
  unique() %>%
  length()
```

```
## [1] 3
```

And most importantly, this works with dataframes, like the `cereal` dataframe we already loaded.

```
cereal %>%
  # This line takes column names
  colnames() %>%
  # This line calculates the number of column names
  length()
```

```
## [1] 16
```

For the record, this is the same as typing `length(colnames(cereal))` or even better, `ncol(cereal)`.

If you are curious, you can read more about the pipe operator here: https://r4ds.had.co.nz/pipes.html

C. Introducing select()

Select is a function from the dplyr package in the tidyverse, and allows us to "select" specific columns from a dataframe. It's easy as pie! Just tell it the names of the columns you want. The first "argument" to the function will be the dataframe in question, so use this format: `select(data, column1, column2, etc...)`. With %>% , this format becomes `data %>% select(column1, column2, etc...)`. Try it out!

FYI, in the rest of this lab, I'm going to put `head()` outside of a lot of these lines of code, because otherwise the RMarkdown PDF I give you will be like 70 pages because it will print the full output of the subsetted dataframes instead of just the top 5 rows.

```
head(
cereal %>%
  dplyr::select(name, calories, fiber)
)
```

```
##                         name calories fiber
## 1                  100% Bran       70  10.0
## 2          100% Natural Bran      120   2.0
## 3                    All-Bran       70   9.0
## 4 All-Bran with Extra Fiber       50  14.0
## 5             Almond Delight      110   1.0
## 6    Apple Cinnamon Cheerios      110   1.5
```

(in case we haven't yet covered this in a lab... lots of different R packages have a function called `select()` which means that one package might "mask" another package and break the code. with the `dplyr::` part of the code above, we're telling R to use the `select()` function from within the dplyr package)

Remember that the code above selects the column names but doesn't actually save that dataframe as anything. To actually store the new dataframe, you must rename it as a new data object like this:

```
cereal2 = cereal %>%
  dplyr::select(name, calories, fiber)

head(cereal2)
```

```
##                         name calories fiber
## 1                  100% Bran       70  10.0
## 2          100% Natural Bran      120   2.0
## 3                    All-Bran       70   9.0
## 4 All-Bran with Extra Fiber       50  14.0
## 5             Almond Delight      110   1.0
## 6    Apple Cinnamon Cheerios      110   1.5
```

This new frame has only the name of the cereal and the amount of calories and fiber it has. Importantly, notice that the original data frame, `cereal`, is completely unmodified! We can go back and use this data frame as we please.

Now, here's a new task for you. Go ahead and create a new data frame that has the name, manufacturer, and amount of sugar. Name the data frame something intuitive. Hint: first, figure out which columns you want to keep (using `colnames(...)`) and then use the `select()` function with a pipe, like above.

For the record, you can also tell select to EXCLUDE certain columns by using a minus sign in front of the column name like this:

```
head(
cereal %>%
  dplyr::select(-name, -mfr)
)
```

```
##   type calories protein fat sodium fiber carbo sugars potass vitamins shelf
## 1    C       70       4   1    130  10.0   5.0      6    280       25     3
## 2    C      120       3   5     15   2.0   8.0      8    135        0     3
## 3    C       70       4   1    260   9.0   7.0      5    320       25     3
## 4    C       50       4   0    140  14.0   8.0      0    330       25     3
## 5    C      110       2   2    200   1.0  14.0      8     -1       25     3
## 6    C      110       2   2    180   1.5  10.5     10     70       25     1
##   weight cups   rating
## 1      1 0.33 68.40297
## 2      1 1.00 33.98368
## 3      1 0.33 59.42551
## 4      1 0.50 93.70491
## 5      1 0.75 34.38484
## 6      1 0.75 29.50954
```

D. Introducing filter()

We can use the filter function in two different ways: To filter rows based on number values or to filter characters. For instance, the manufacturer column is filled with characters, while the nutritional columns are numeric.

Which cereals have greater than 12 grams of sugar?

```
cereal %>%
  filter(sugars > 12)
```

```
##                    name           mfr type calories protein fat sodium fiber
## 1           Apple Jacks      Kelloggs    C      110       2   0    125     1
## 2           Cocoa Puffs General_Mills    C      110       1   1    180     0
## 3         Count Chocula General_Mills    C      110       1   1    180     0
## 4           Froot Loops      Kelloggs    C      110       2   1    125     1
## 5          Golden Crisp          Post    C      100       2   0     45     0
## 6  Mueslix Crispy Blend      Kelloggs    C      160       3   2    150     3
## 7 Post Nat. Raisin Bran          Post    C      120       3   1    200     6
## 8                Smacks      Kelloggs    C      110       2   1     70     1
## 9     Total Raisin Bran General_Mills    C      140       3   1    190     4
##   carbo sugars potass vitamins shelf weight cups   rating
## 1    11     14     30       25     2   1.00 1.00 33.17409
## 2    12     13     55       25     2   1.00 1.00 22.73645
## 3    12     13     65       25     2   1.00 1.00 22.39651
## 4    11     13     30       25     2   1.00 1.00 32.20758
```

```
## 5    11      15      40        25      1    1.00 0.88 35.25244
## 6    17      13     160        25      3    1.50 0.67 30.31335
## 7    11      14     260        25      3    1.33 0.67 37.84059
## 8     9      15      40        25      2    1.00 0.75 31.23005
## 9    15      14     230       100      3    1.50 1.00 28.59278
```

How many cereals have greater than 12 grams of sugar? Try adding `%>% nrow()` to the end of your last line of code.

There is SO MUCH flexibility with the logical operators you use to filter. As an example, you could also do:

R Comparison	Interpretation	Example	Answer
==	"equal to"	1==2	FALSE
>	"greater than"	1>2	FALSE
>=	"greater than or equal to"	1>=2	FALSE
<	"less than"	1<2	TRUE
<=	"less than or equal to"	1<=2	TRUE
!=	"not equal to"	1!=2	TRUE

(NOTE: Exactly equal to is NOT just one single "=", because that operator is used to assign variables)

Subsetting conditions can be combined with & ("and") or | ("or"). You'll see this in later labs, but it becomes particularly handy for if/then/else statements (e.g., if the mean of a vector is less than 2, do something, otherwise do a different thing).

For example, let's figure out which cereals are manufactured by Kelloggs?

```
cereal %>%
  filter(mfr == "Kelloggs")
```

```
##                           name      mfr type calories protein fat sodium fiber
## 1                     All-Bran Kelloggs    C       70       4   1    260     9
## 2    All-Bran with Extra Fiber Kelloggs    C       50       4   0    140    14
## 3                   Apple Jacks Kelloggs    C      110       2   0    125     1
## 4                   Corn Flakes Kelloggs    C      100       2   0    290     1
## 5                     Corn Pops Kelloggs    C      110       1   0     90     1
## 6           Cracklin' Oat Bran Kelloggs    C      110       3   3    140     4
## 7                      Crispix Kelloggs    C      110       2   0    220     1
## 8                   Froot Loops Kelloggs    C      110       2   1    125     1
## 9                Frosted Flakes Kelloggs    C      110       1   0    200     1
## 10         Frosted Mini-Wheats Kelloggs    C      100       3   0      0     3
## 11               Fruitful Bran Kelloggs    C      120       3   0    240     5
## 12 Just Right Crunchy  Nuggets Kelloggs    C      110       2   1    170     1
## 13      Just Right Fruit & Nut Kelloggs    C      140       3   1    170     2
## 14        Mueslix Crispy Blend Kelloggs    C      160       3   2    150     3
## 15            Nut&Honey Crunch Kelloggs    C      120       2   1    190     0
## 16   Nutri-Grain Almond-Raisin Kelloggs    C      140       3   2    220     3
## 17           Nutri-grain Wheat Kelloggs    C       90       3   0    170     3
## 18                 Product 19 Kelloggs    C      100       3   0    320     1
## 19                Raisin Bran Kelloggs    C      120       3   1    210     5
## 20              Raisin Squares Kelloggs    C       90       2   0      0     2
## 21               Rice Krispies Kelloggs    C      110       2   0    290     0
## 22                     Smacks Kelloggs    C      110       2   1     70     1
```

```
## 23                  Special K Kelloggs    C      110       6   0    230     1
##    carbo sugars potass vitamins shelf weight cups   rating
## 1      7      5    320       25     3   1.00 0.33 59.42551
## 2      8      0    330       25     3   1.00 0.50 93.70491
## 3     11     14     30       25     2   1.00 1.00 33.17409
## 4     21      2     35       25     1   1.00 1.00 45.86332
## 5     13     12     20       25     2   1.00 1.00 35.78279
## 6     10      7    160       25     3   1.00 0.50 40.44877
## 7     21      3     30       25     3   1.00 1.00 46.89564
## 8     11     13     30       25     2   1.00 1.00 32.20758
## 9     14     11     25       25     1   1.00 0.75 31.43597
## 10    14      7    100       25     2   1.00 0.80 58.34514
## 11    14     12    190       25     3   1.33 0.67 41.01549
## 12    17      6     60      100     3   1.00 1.00 36.52368
## 13    20      9     95      100     3   1.30 0.75 36.47151
## 14    17     13    160       25     3   1.50 0.67 30.31335
## 15    15      9     40       25     2   1.00 0.67 29.92429
## 16    21      7    130       25     3   1.33 0.67 40.69232
## 17    18      2     90       25     3   1.00 1.00 59.64284
## 18    20      3     45      100     3   1.00 1.00 41.50354
## 19    14     12    240       25     2   1.33 0.75 39.25920
## 20    15      6    110       25     3   1.00 0.50 55.33314
## 21    22      3     35       25     1   1.00 1.00 40.56016
## 22     9     15     40       25     2   1.00 0.75 31.23005
## 23    16      3     55       25     1   1.00 1.00 53.13132
```

NOTE: Remember the quotation marks, since it's a character! Also remember that R is case-sensitive, so for example, `mfr=="kelloggs"` would not have worked (well, the code would run, but it would return 0 rows because it wouldn't match anything).

Now we will look at how the %>% operator can be particularly helpful when we want to do multiple steps of data manipulation. Say we want to select certain columns and THEN filter that data. We COULD do it this way:

```
cereal2 = cereal %>%
  dplyr::select(name, sugars, protein)

cereal3 = cereal2 %>%
  filter(sugars > 12)
```

What is the drawback of this? It clutters up our working environment, adding unnecessarily many data frames. So instead, let's try it with piping:

```
cereal %>%
  dplyr::select(name, sugars, protein) %>%
  filter(sugars > 12)
```

```
##                   name sugars protein
## 1          Apple Jacks     14       2
## 2          Cocoa Puffs     13       1
## 3        Count Chocula     13       1
## 4          Froot Loops     13       2
## 5         Golden Crisp     15       2
```

```
## 6  Mueslix Crispy Blend     13       3
## 7 Post Nat. Raisin Bran     14       3
## 8                Smacks     15       2
## 9     Total Raisin Bran     14       3
```

This is a nice and TIDY (ha ha, get it?) way to run these functions; it progressively shows the steps we took, and we don't have to keep writing the name of the dataframe over and over again.

Challenge: starting with cereal, create a dataframe that has the name of the cereal and the amount of fiber, fat, and sodium per serving, but *only* for cereals that have more than 120 calories. This is tricky! Think about the order in which you do things here.

```
cereal %>%
  filter(calories > 120) %>%
  dplyr::select(name, fiber, fat, sodium)
```

```
##                                name fiber fat sodium
## 1                           Basic 4   2.0   2    210
## 2            Just Right Fruit & Nut   2.0   1    170
## 3  Muesli Raisins; Dates; & Almonds   3.0   3     95
## 4 Muesli Raisins; Peaches; & Pecans   3.0   3    150
## 5              Mueslix Crispy Blend   3.0   2    150
## 6         Nutri-Grain Almond-Raisin   3.0   2    220
## 7              Oatmeal Raisin Crisp   1.5   2    170
## 8                 Total Raisin Bran   4.0   1    190
```

You have to use the `filter()` function BEFORE the `select()` function, since `select()` gets rid of the calories column. You can't filter something that isn't there!

E. Introducing mutate()

The mutate() function adds a new column based upon calculations you provide. It applies these calculations to each row. Let's select only a couple columns to make this easier:

```
cereal_carbs = cereal %>%
  dplyr::select(name, carbo, sugars)

head(cereal_carbs)
```

```
##                        name carbo sugars
## 1                 100% Bran   5.0      6
## 2         100% Natural Bran   8.0      8
## 3                   All-Bran   7.0      5
## 4 All-Bran with Extra Fiber   8.0      0
## 5            Almond Delight  14.0      8
## 6   Apple Cinnamon Cheerios  10.5     10
```

Now let's say we want to multiply every value in the sugars column by two, and add those new data into a column called `sugars_total`:

```
head(
  cereal_carbs %>%
  mutate(sugars_total = sugars*2)
)
```

```
##                         name carbo sugars sugars_total
## 1                  100% Bran   5.0      6           12
## 2          100% Natural Bran   8.0      8           16
## 3                   All-Bran   7.0      5           10
## 4 All-Bran with Extra Fiber   8.0      0            0
## 5             Almond Delight  14.0      8           16
## 6   Apple Cinnamon Cheerios  10.5     10           20
```

If you give the new variable the same name as an existing variable, mutate() will OVERRIDE that old variable... so be careful! Remember that it is generally best coding practice to make new variables (or columns) instead of overriding other ones.

```
head(
cereal_carbs %>%
  mutate(sugars = sugars*2)
)
```

```
##                         name carbo sugars
## 1                  100% Bran   5.0     12
## 2          100% Natural Bran   8.0     16
## 3                   All-Bran   7.0     10
## 4 All-Bran with Extra Fiber   8.0      0
## 5             Almond Delight  14.0     16
## 6   Apple Cinnamon Cheerios  10.5     20
```

Now let's say you always eat cereal with milk, which has 5 grams of sugar per serving

```
head(
cereal_carbs %>%
  mutate(sugars_with_milk = sugars + 5)
)
```

```
##                         name carbo sugars sugars_with_milk
## 1                  100% Bran   5.0      6               11
## 2          100% Natural Bran   8.0      8               13
## 3                   All-Bran   7.0      5               10
## 4 All-Bran with Extra Fiber   8.0      0                5
## 5             Almond Delight  14.0      8               13
## 6   Apple Cinnamon Cheerios  10.5     10               15
```

We can add the two columns together to calculate how many total carbs you eat with your bowl of cereal by adding the two columns together:

```
head(
cereal_carbs %>%
  mutate(total_carbs = carbo + sugars)
)
```

```
##                         name carbo sugars total_carbs
## 1                  100% Bran   5.0      6        11.0
## 2          100% Natural Bran   8.0      8        16.0
## 3                    All-Bran   7.0      5        12.0
## 4 All-Bran with Extra Fiber   8.0      0         8.0
## 5             Almond Delight  14.0      8        22.0
## 6   Apple Cinnamon Cheerios  10.5     10        20.5
```

If this seems like it is getting a little complicated, just keep in mind that you can do all of these data manipulations in base R, using things like: `cereal_carbs$total_carbs=cereal_carbs$carbo+5`... but.... yuck.

One other cool thing you can do is add multiple new columns at once!

```
head(
cereal_carbs %>%
  mutate(total_carbs = carbo + sugars,
         total_with_milk = total_carbs + 5)
)
```

```
##                         name carbo sugars total_carbs total_with_milk
## 1                  100% Bran   5.0      6        11.0            16.0
## 2          100% Natural Bran   8.0      8        16.0            21.0
## 3                    All-Bran   7.0      5        12.0            17.0
## 4 All-Bran with Extra Fiber   8.0      0         8.0            13.0
## 5             Almond Delight  14.0      8        22.0            27.0
## 6   Apple Cinnamon Cheerios  10.5     10        20.5            25.5
```

F. Introducing group_by() and summarize()

Say we want to look at the average sugar content per cereal type, but grouped by manufacturer... How would we do that?

Conceptually, what we would want to do is to take the values in the sugar column that share the same value in the manufacturer column (e.g. Quaker_Oats, Kelloggs) and average them. We can accomplish this using the very handy functions `group_by()` and `summarise()`. `group_by()` tells R that you want to assign "groups" within your data. R then knows that all observations for a given column belong to that group. Let's try it out by using `mfr` (manufacturer) as a grouping variable.

So let's check out the original data again, after grouping by `mfr`:

```
head(
cereal %>%
  group_by(mfr)
)
```

```
## # A tibble: 6 x 16
## # Groups:   mfr [5]
##   name       mfr   type  calories protein   fat sodium fiber carbo sugars potass
##   <chr>      <chr> <chr>    <int>   <int> <int>  <int> <dbl> <dbl>  <int>  <int>
## 1 100% Bran  Nabi~ C           70       4     1    130  10     5        6    280
## 2 100% Natu~ Quak~ C          120       3     5     15   2     8        8    135
## 3 All-Bran   Kell~ C           70       4     1    260   9     7        5    320
```

```
## 4 All-Bran ~ Kell~ C           50       4     0    140  14     8        0    330
## 5 Almond De~ Rals~ C          110       2     2    200   1    14        8     -1
## 6 Apple Cin~ Gene~ C          110       2     2    180   1.5  10.5     10     70
## # i 5 more variables: vitamins <int>, shelf <int>, weight <dbl>, cups <dbl>,
## #   rating <dbl>
```

What changed?? Well, with the data, nothing!!! Nothing has been modified. But notice at the top of the data frame, it tells you what column is being used as the group, as well as how many unique values there are. This tells us how many unique groups exist in our data based on the column mfr.

The real functionality of `group_by()` comes in once we use another function to modify this data. Let's try it by using the `summarise()` function (`summarize()` spelled with a z also works). Summarise does what we just talked about: it takes a data table, typically a grouped data table, and summarises a current column to create a new column based on a function that you provide.

Let's summarise the "sugars" column by calculating the average grams of sugars for each cereal manufacturer.

```
cereal %>%
  group_by(mfr) %>%
  summarise(mean_sugars = mean(sugars))
```

```
## # A tibble: 7 x 2
##   mfr                          mean_sugars
##   <chr>                              <dbl>
## 1 American_Home_Food_Products         3
## 2 General_Mills                       7.95
## 3 Kelloggs                            7.57
## 4 Nabisco                             1.83
## 5 Post                                8.78
## 6 Quaker_Oats                         5.25
## 7 Ralston_Purina                      6.12
```

One thing that will become important later.... the `mean()` calculation won't work if you have NA values in your dataframes. Try it out for yourself by making a vector that includes an NA value and then trying to calculate the mean of that vector.

```
mean(c(2,5,900,NA,60))
```

```
## [1] NA
```

The code returns NA. Instead, try adding `na.rm=TRUE` inside the `mean()` function, which says "remove the NA values before taking the mean":

```
mean(c(2,5,900,NA,60),na.rm=TRUE)
```

```
## [1] 241.75
```

That worked! You don't need that now (if the cereal dataset had NAs, your summarize function above would have looked like `summarise(mean_sugars = mean(sugars, na.rm=TRUE))`), but will for the elephant example later!

Just for fun, I want to show you how to do these calculations in Base... without dplyr.

Remember, we want to calculate the average sugar content for each cereal manufacturer. We can use the `aggregate()` function, but first let's make sure the data are in the appropriate format.

```
str(cereal$mfr)
```

```
##  chr [1:77] "Nabisco" "Quaker_Oats" "Kelloggs" "Kelloggs" "Ralston_Purina" ...
```

```
str(cereal$sugars)
```

```
##  int [1:77] 6 8 5 0 8 10 14 8 6 5 ...
```

Looks like `mfr` is a character and `sugars` is an integer Great! Let's use the aggregate function. The aggregate function format is something like `aggregate(y~x+z,dat=...,FUN=...)`. FYI I use `...` as a placeholder. For example, your code might look like this:

```
out=aggregate(sugars~mfr,data=cereal,FUN="mean")
out
```

```
##                          mfr   sugars
## 1 American_Home_Food_Products 3.000000
## 2               General_Mills 7.954545
## 3                    Kelloggs 7.565217
## 4                     Nabisco 1.833333
## 5                        Post 8.777778
## 6                 Quaker_Oats 5.250000
## 7              Ralston_Purina 6.125000
```

In normal words, this line of code says "calculate the mean sugar content of cereal made by each manufacturer using the dataset `cereal`". Do the numbers in this dataset align with the numbers from the tidyverse solution? They should :)

You can always export aggregated data from R into excel or another file format using the `write.csv()` function which requires a few inputs: the data you want to export (out), the filename you want to use INCLUDING THE FILE EXTENSION (e.g., MeanSugarByCerealMfr.csv), and row.names=FALSE. If you don't add row.names=FALSE, the default is to create a new column with sequential row numbers (1,2,3, etc) which is really annoying. For example, if you wanted to export this aggregated dataset you would type `write.csv(out,file="MeanSugarByCerealMfr.csv",row.names=FALSE)`. Go ahead and try it; there should now be a new .csv file in your working directory called "MeanSugarByCerealMfr".

OK. Back to Tidyverse. To summarise another column, we just add another argument separated by a comma. Here, in addition to calculating average sugar, we can ALSO calculate average calories:

```
cereal %>%
  group_by(mfr) %>%
  summarise(mean_sugars = mean(sugars),
            mean_calories = mean(calories))
```

```
## # A tibble: 7 x 3
##   mfr                          mean_sugars mean_calories
##   <chr>                              <dbl>         <dbl>
## 1 American_Home_Food_Products         3            100
## 2 General_Mills                       7.95         111.
## 3 Kelloggs                            7.57         109.
## 4 Nabisco                             1.83          86.7
## 5 Post                                8.78         109.
## 6 Quaker_Oats                         5.25          95
## 7 Ralston_Purina                      6.12         115
```

G. Examples using pivot_longer() and pivot_wider()

Pivoting functions allow us to pivot data from long to wide format and back again. Check out the help pages for these functions; they are super useful!

Pivot_longer() takes certain columns that you name and pivots then into their own column, and tosses their values into a second column:

```
cereal_long = cereal %>%
  pivot_longer(cols = c(-name, -mfr, -type),
               names_to = "nutrient",
               values_to = "value")

head(cereal_long)
```

```
## # A tibble: 6 x 5
##   name      mfr     type  nutrient value
##   <chr>     <chr>   <chr> <chr>    <dbl>
## 1 100% Bran Nabisco C     calories    70
## 2 100% Bran Nabisco C     protein      4
## 3 100% Bran Nabisco C     fat          1
## 4 100% Bran Nabisco C     sodium     130
## 5 100% Bran Nabisco C     fiber       10
## 6 100% Bran Nabisco C     carbo        5
```

As a tiny exercise, try to write out in plain English what the chunk of code above does.

```
cereal_wide = cereal_long %>%
  pivot_wider(names_from = "nutrient",
              values_from = "value")
head(cereal_wide)
```

```
## # A tibble: 6 x 16
##   name       mfr   type  calories protein   fat sodium fiber carbo sugars potass
##   <chr>      <chr> <chr>    <dbl>   <dbl> <dbl>  <dbl> <dbl> <dbl>  <dbl>  <dbl>
## 1 100% Bran  Nabi~ C           70       4     1    130  10     5         6    280
## 2 100% Natu~ Quak~ C          120       3     5     15   2     8         8    135
## 3 All-Bran   Kell~ C           70       4     1    260   9     7         5    320
## 4 All-Bran ~ Kell~ C           50       4     0    140  14     8         0    330
## 5 Almond De~ Rals~ C          110       2     2    200   1    14         8     -1
## 6 Apple Cin~ Gene~ C          110       2     2    180   1.5  10.5      10     70
## # i 5 more variables: vitamins <dbl>, shelf <dbl>, weight <dbl>, cups <dbl>,
## #   rating <dbl>
```

Again, try to write out in plain English what that code chunk does. Notice that this pivots the table back to its original format!

2) Basic plotting

Let's get familiar with integrating the data manipulation techniques you just learned into basic plots! Open the file `ourworld.csv`.

```
rm(list=ls()) #fresh start!
dat=read.csv("ourworld.csv")
```

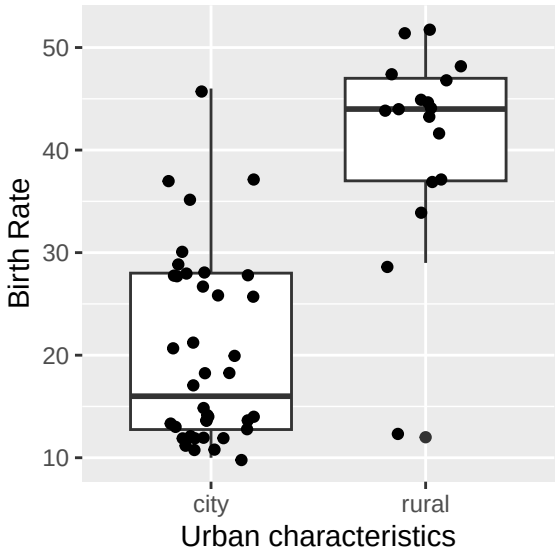
This is a file that describes demographic and other attributes of countries of the world. Check out the columns in the dataset to see which parameters you might want to plot:

```
colnames(dat)
```

```
##  [1] "Country"  "Pop_1983" "Pop_1986" "Pop_1990" "Pop_2020" "Urban"
##  [7] "Birth_82" "Birth_Rt" "Death_82" "Death_Rt" "Babymt82" "Babymort"
## [13] "Life_Exp" "Gnp_82"   "Gnp_86"   "Gdp_Cap"  "Log_Gdp"  "Educ_84"
## [19] "Educ"     "Health84" "Health"   "Mil_84"   "Mil"      "Govern"
## [25] "Leader"   "Gov"      "Gnp"      "B_To_D82" "Urban.1"  "Lifeexpm"
## [31] "Lifeexpf" "Literacy" "Group"    "B_To_D"   "Group.1"  "Gdp"
## [37] "Lat"      "Lon"      "Mcdonald"
```

Assume you want to look at the relationship between birth rate `Birth_Rt` and whether a country is urban or rural `Urban.1`. Because birth rate is continuous and urban is a character (check for yourself using `str()` on each column), you'll want to use a boxplot or barplot.

```
ggplot(data=dat,aes(x=Urban.1,y=Birth_Rt))+
        geom_boxplot()+geom_jitter(width=0.2)+
        labs(x="Urban characteristics",y="Birth Rate")
```



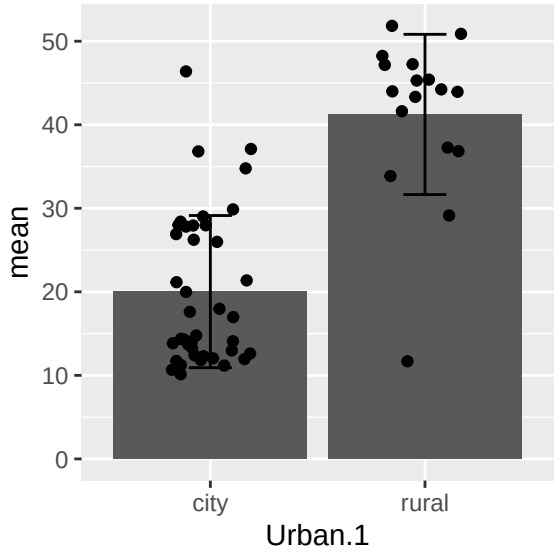
Barplots (especially with error bars) are pretty tough to make using base R. The `ggplot` package makes it much easier, but first you need to calculate the mean and standard deviation of each group:

```
out=dat %>%
  group_by(Urban.1) %>%
  summarize(mean=mean(Birth_Rt),
            sd=sd(Birth_Rt))
out
```

```
## # A tibble: 2 x 3
##   Urban.1  mean    sd
##   <chr>   <dbl> <dbl>
## 1 city     20.0  9.11
## 2 rural    41.2  9.59
```

Now would be a good time to check that `out` is what you think it is (your code worked properly)! Then, make the plot (don't forget to overlay the raw datapoints from the `dat` dataframe):

```
#make a bar graph with error bars
library(ggplot2)
ggplot(data=out, aes(x=Urban.1, y=mean)) +
    geom_bar(stat="identity", position=position_dodge()) +
    geom_jitter(data=dat,aes(x=Urban.1,y=Birth_Rt),width=0.2)+
    geom_errorbar(aes(ymin=mean-sd, ymax=mean+sd), width=.2)
```



See how the default y label ("mean") is totally uninformative? Go ahead and change that using the code I gave you in the last lab (or Google "how to change y label of ggplot"). Make sure your new y-label includes information about what the error bars represent (standard deviation, abbreviated SD).

Let's say we want to use standard error instead. We basically need to calculate standard error first and then use it for the plot (FYI I'm taking the dplyr code from above and simply adding two lines).

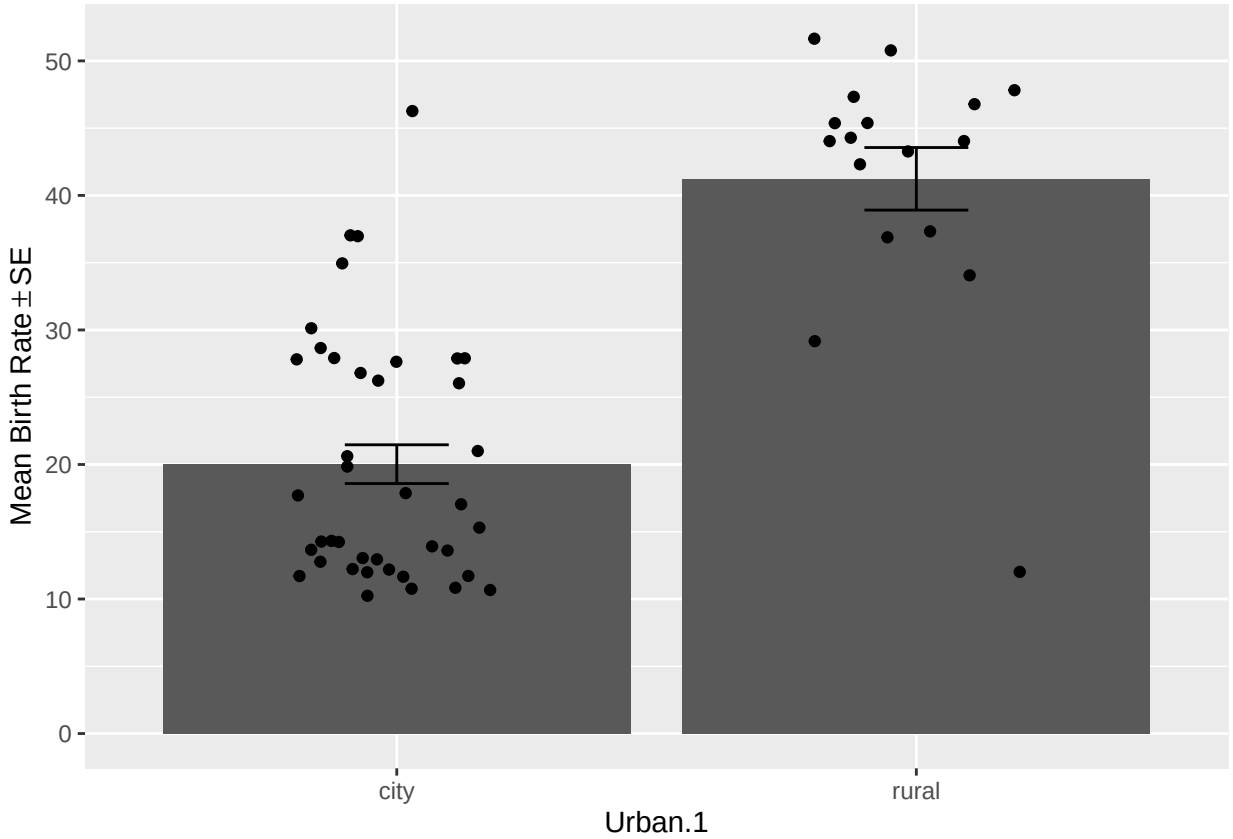
```
#first calculate the sample size of each group
out=dat %>%
  group_by(Urban.1) %>%
  summarize(mean=mean(Birth_Rt),
            sd=sd(Birth_Rt),
            n=n(),
            se=sd/sqrt(n))
out
```

```
## # A tibble: 2 x 5
##   Urban.1  mean    sd     n    se
##   <chr>   <dbl> <dbl> <int> <dbl>
## 1 city     20.0  9.11    40  1.44
## 2 rural    41.2  9.59    17  2.33
```

One quick thing I want to point out about using piping to summarize mean, sd, etc. Do you see how we specify that we want `mean()` and `sd()` to use the "Birth_Rt" column, whereas we don't specify a column for the `n()` calculation? This is because all of the columns have the same length (e.g., the length of the Birth_Rt column is the same as any other column). So we don't need to specify `n(Birth_Rt)`. In the new plot, don't forget to specify that the error bars are now SE instead of SD!

The easy way to label the y axis of this plot would be `labs(y="Mean Birth Rate +- SE)` but that's not very pretty. Try using the `expression()` function as I've done down below. To make the +- sign look fancy, you just include `%+-%` within the expression function. Google "Axis labels in R plots using expression() command" to see what other options you have for making fancy axis labels - like exponents, subscripts, arrows, etc.

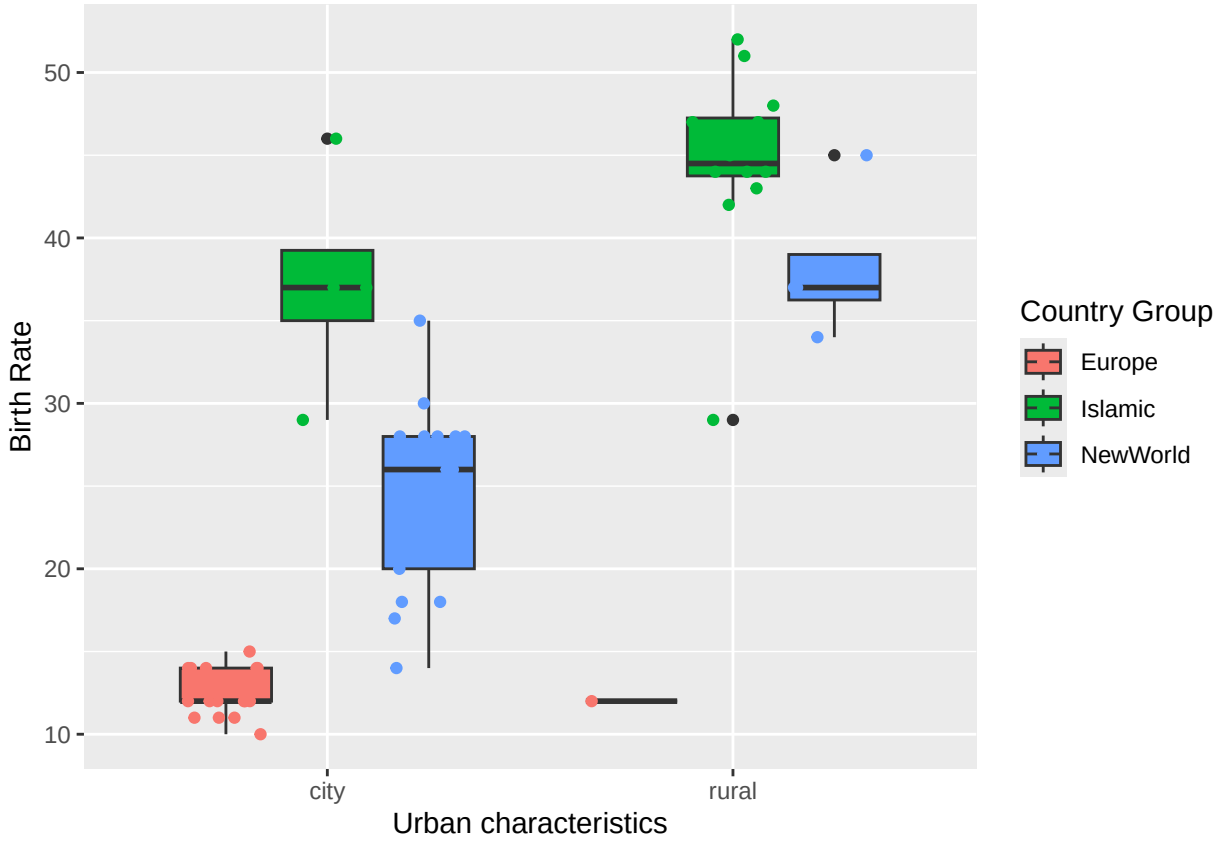
```
#make a bar graph with error bars
ggplot(data=out, aes(x=Urban.1, y=mean)) +
    geom_bar(stat="identity", position=position_dodge()) +
    geom_errorbar(aes(ymin=mean-se, ymax=mean+se), width=.2)+
    geom_jitter(data=dat,aes(x=Urban.1,y=Birth_Rt),width=0.2)+
    labs(y=expression("Mean Birth Rate" %+-% "SE"))
```



Play around with boxplots and barplots using a new variable of your choice. For example, try adding in `Group` as an explanatory variable. Group is a character with three levels: Europe, Islamic, NewWorld which you can find out using `str()` or `summary()`. Pro trip, try using `table()` to see how many cases there are of each `Group`. For the boxplot, you can add the interaction between `Urban.1` and `Group` by using one (let's say, `Urban.1`) as the x variable and one (let's say, `Group`), as the fill variable:

```
ggplot(data=dat,aes(x=Urban.1,y=Birth_Rt,fill=Group))+
        geom_boxplot()+
        geom_point(aes(col=Group),position=position_jitterdodge(jitter.width = .2))+
```

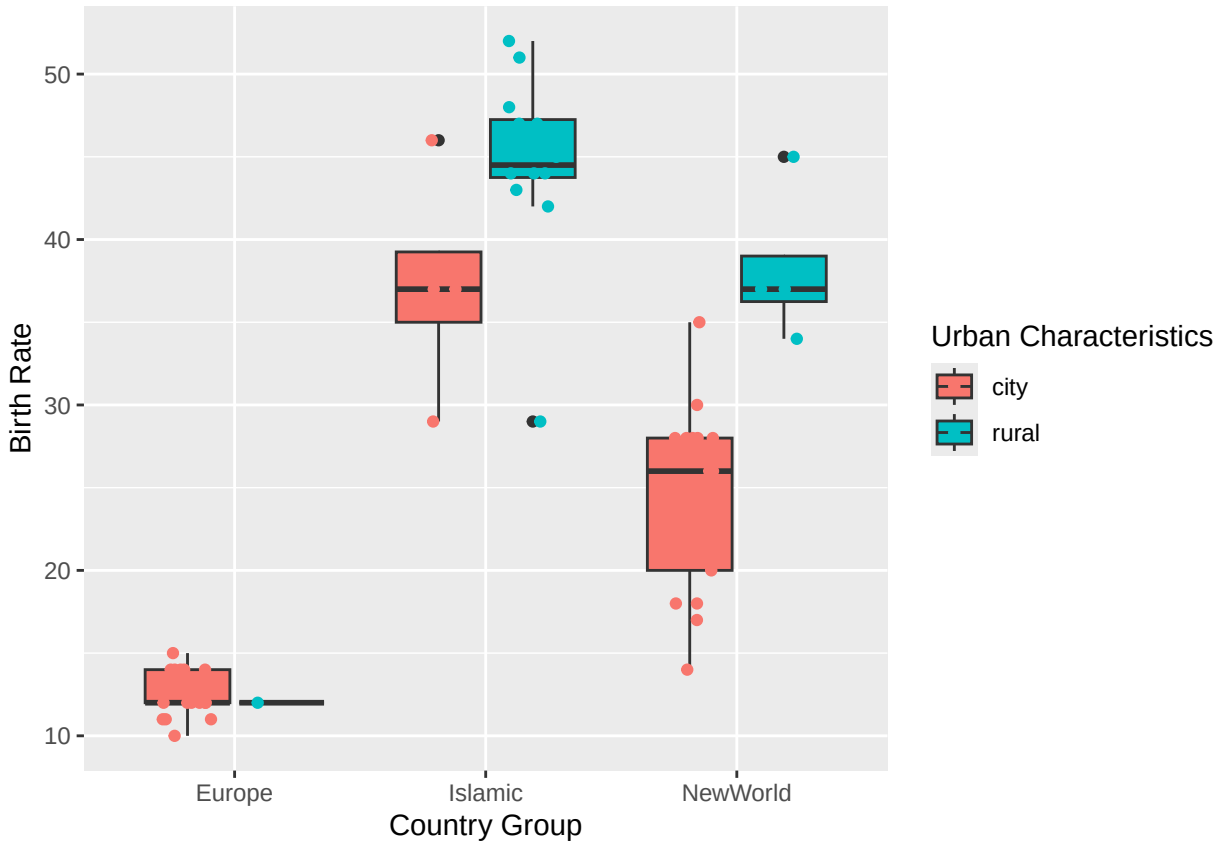
```
        labs(x="Urban characteristics",
             y="Birth Rate",
             col="Country Group",
             fill="Country Group")
```



PS: Don't worry too much about this, but I had to use `position=position_jitterdodge(jitter.width = .2)` within `geom_point()` instead of `geom_jitter()` because the latter is garbage with groups. Dumb ggplot weirdnesses.

Keep in mind that the order of x values matters. If instead you use `Group` as the x variable and `Urban.1` as the fill, the plot will look very different (make sure to switch the `col` specification in `geom_point()` to `Urban.1` as well!):

```
p1=ggplot(data=dat,aes(x=Group,y=Birth_Rt,fill=Urban.1))+
        geom_boxplot()+
        geom_point(aes(col=Urban.1),position=position_jitterdodge(jitter.width=.2))+
        labs(x="Country Group",y="Birth Rate",
             col="Urban Characteristics",
             fill="Urban Characteristics")
p1
```

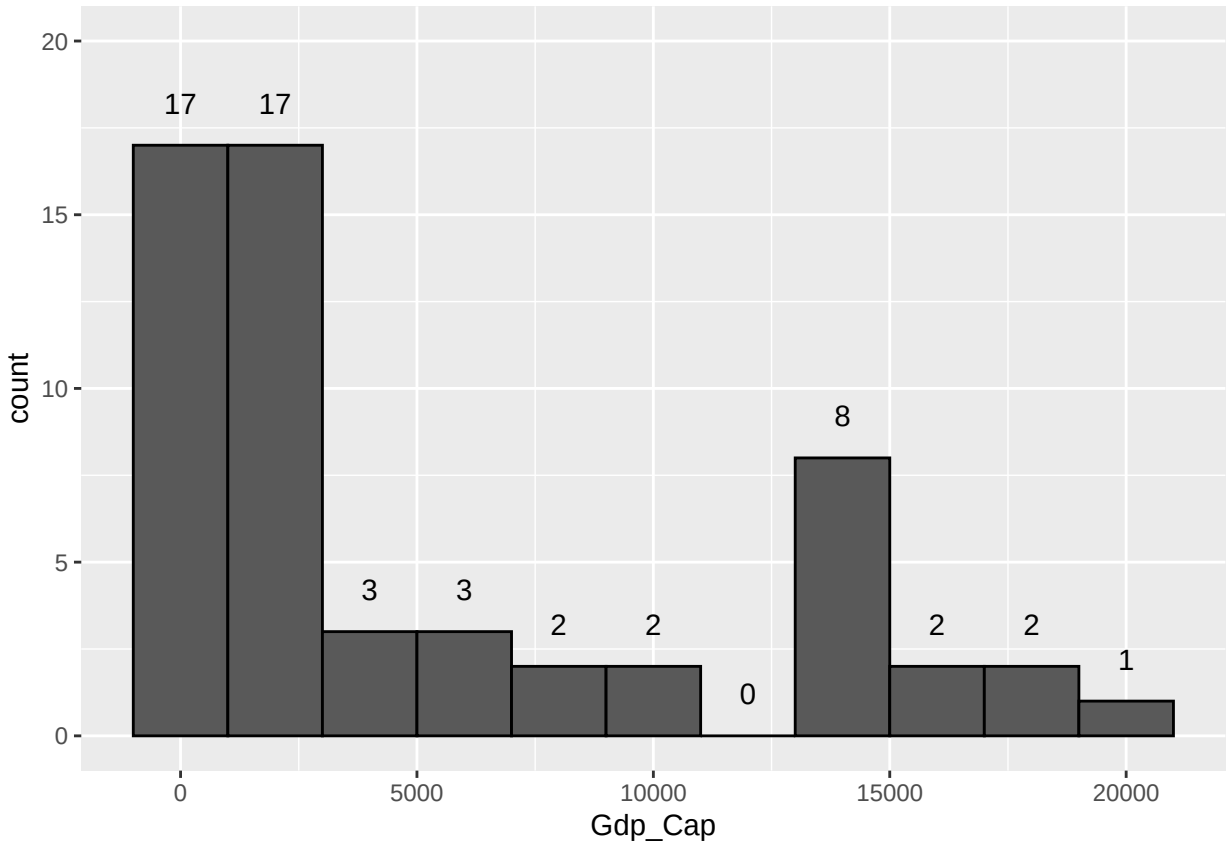


One thing I want to point out.... there are two ways to make ggplots. One is to just make the ggplot (the code line starts with `ggplot(...)`), and the other is to save the ggplot as something (for instance, the code line starts with `p1=ggplot(...)`) and then the plot prints when you type `p1` into the source code or console. This latter option is what I typically use, and is what I did in the last plot above.

So of the last two plots we made, which should we use? Well, the order of your variables will depend on the point you want to make..... To practice, think about which plot shows that Islamic countries have higher birth rates than European countries, and think about which plot shows that Rural countries tend to have higher birthrates than Urban countries. Let us know if you have questions!

Let's make a histogram of `Gdp_Cap` and label each bar with the count. We'll call this plot `p2`. (If you want, check out the help function for `after_stat()` to see what I use it in the label function below.

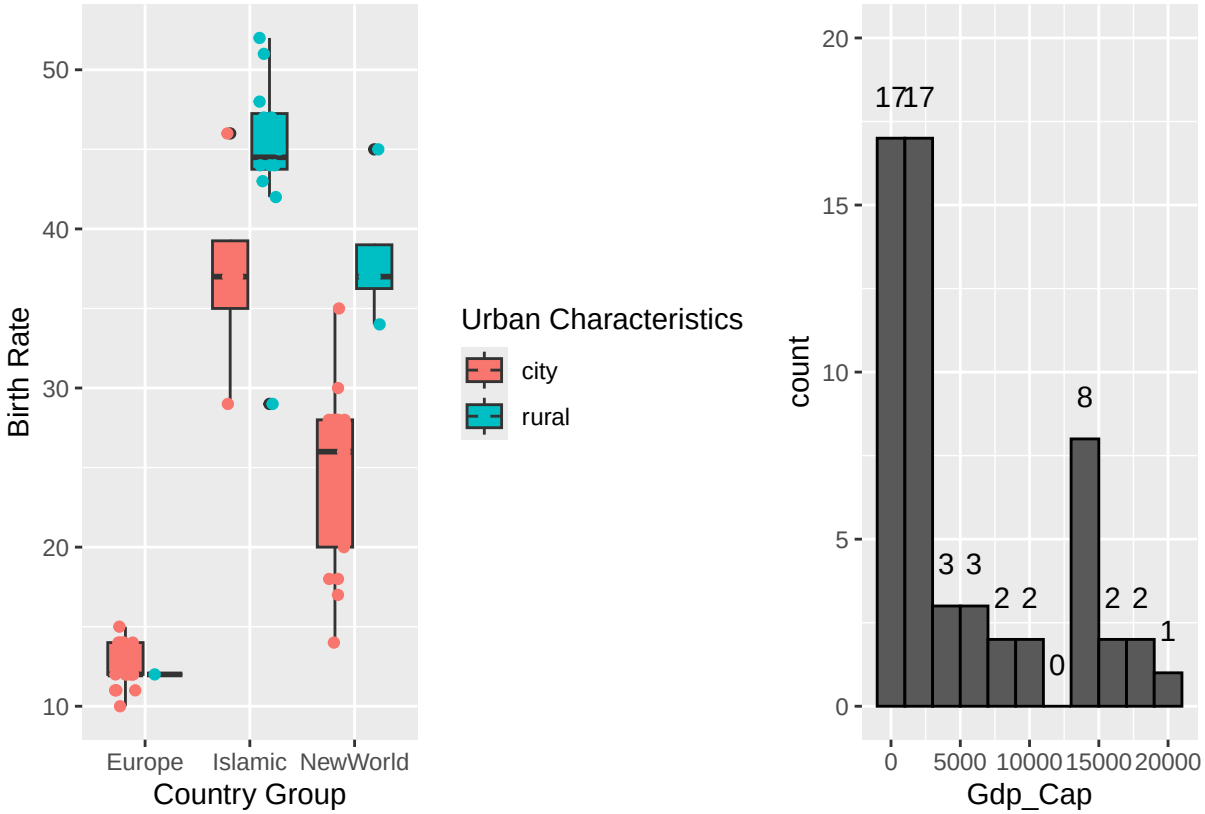
```
p2=ggplot(data=dat,aes(x=Gdp_Cap))+
  geom_histogram(binwidth=2000,col="black")+
    stat_bin(binwidth=2000, geom="text", aes(label=after_stat(count)), vjust=-1.5)+
  ylim(0,20)
p2
```



(FYI I had to change the y-axis limits to `(0,20)` so that the label on the tallest bar would fit within the plot).

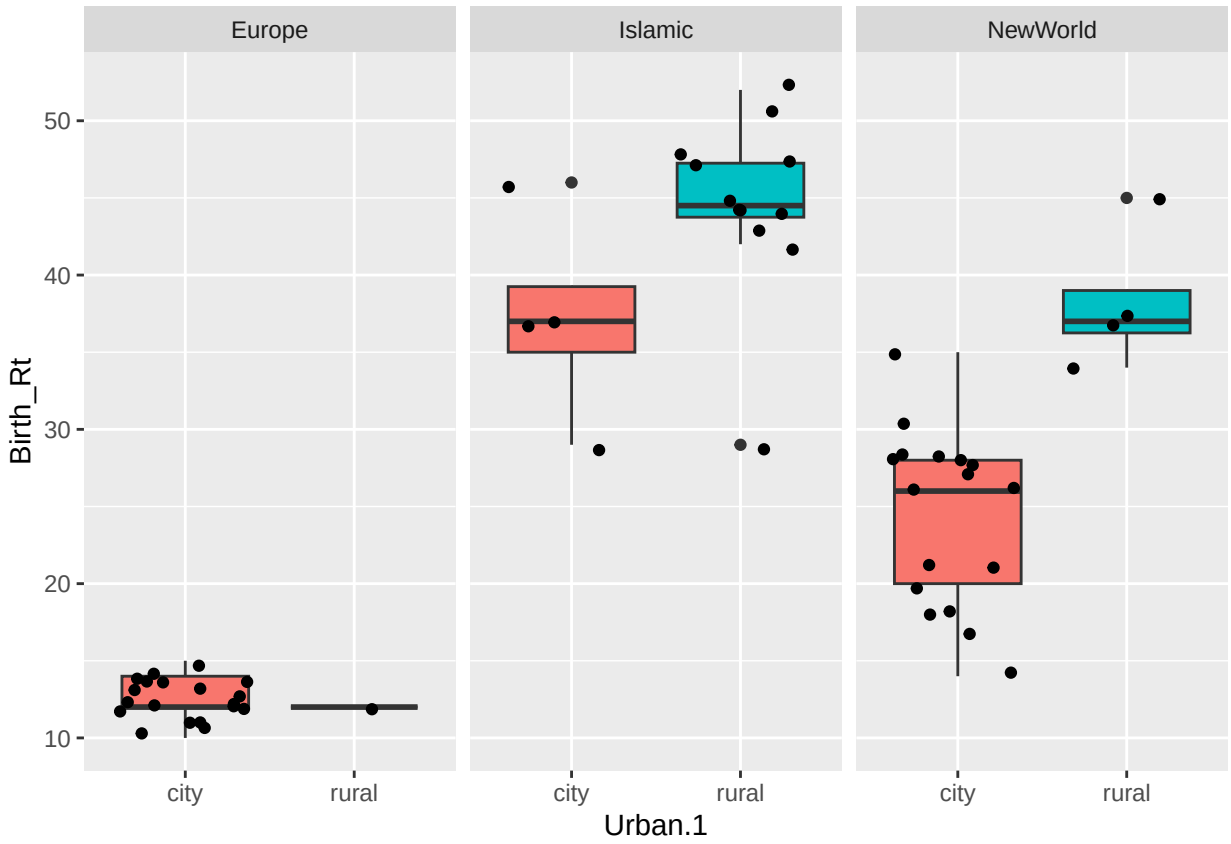
One of the cool things you can do with saved plots is make multiple panels within the same plot using the patchwork package like this:

```
library(patchwork)
p1+p2
```



One other thing you might consider for making that left panel (the boxplot) a little easier to interpret is using `facet_wrap()` on the `Group` variable, like below. Note that I've chosen to hide the legend because it is redundant to the "city" and "rural" labels on the x-axis... try commenting out (adding a `#` at the beginning of the line) the `theme(...)` part of the code below so you can see what the legend would have looked like. Hiding the ggplot legend is something I do pretty frequently - a neat trick :).

```
ggplot(dat, aes(x=Urban.1, y=Birth_Rt,fill=Urban.1))+
  geom_boxplot()+geom_jitter()+facet_wrap(.~Group)+
  theme(legend.position="none")
```



Fun fact, it is also possible to pull out the actual data behind each plot so you can see exactly what is going on. Let's say we want to know the minimum and maximum values of the histogram's (`p1`) first bin, exactly how many values are in the first bin, and which values they are. There's a function called `ggplot_build()` that does this:

```
p2nums = ggplot_build(p2)
head(p2nums$data[[1]])
```

```
##    y count     x  xmin  xmax      density    ncount  ndensity flipped_aes PANEL
## 1 17    17     0 -1000  1000 1.491228e-04 1.0000000 1.0000000       FALSE     1
## 2 17    17  2000  1000  3000 1.491228e-04 1.0000000 1.0000000       FALSE     1
## 3  3     3  4000  3000  5000 2.631579e-05 0.1764706 0.1764706       FALSE     1
## 4  3     3  6000  5000  7000 2.631579e-05 0.1764706 0.1764706       FALSE     1
## 5  2     2  8000  7000  9000 1.754386e-05 0.1176471 0.1176471       FALSE     1
## 6  2     2 10000  9000 11000 1.754386e-05 0.1176471 0.1176471       FALSE     1
##   group ymin ymax colour   fill linewidth linetype alpha
## 1    -1    0   17  black grey35       0.5        1    NA
## 2    -1    0   17  black grey35       0.5        1    NA
## 3    -1    0    3  black grey35       0.5        1    NA
## 4    -1    0    3  black grey35       0.5        1    NA
## 5    -1    0    2  black grey35       0.5        1    NA
## 6    -1    0    2  black grey35       0.5        1    NA
```

We can see that the first bin (first row of the table above) contains values between `Gdp_Cap=-1000` (xmin) and `Gdp_Cap=1000` (xmax), and that there are 17 values in that bin (count). Does that match the histogram figure above? It should :)

For fun, let's do some data manipulation to pull out all of the raw data points that fall within that first bin. First, let's try using the `subset()` function in base R.

```
subset(dat,Gdp_Cap>-1000&Gdp_Cap<1000)
```

```
##        Country Pop_1983 Pop_1986   Pop_1990 Pop_2020 Urban Birth_82 Birth_Rt
## 21      Gambia      0.7      0.8   0.848147      1.7    18       49       48
## 23    Pakistan     94.7    101.9 114.649408    194.4    28       43       43
## 24  Bangladesh     95.9    104.1 118.433064    245.0    11       49       42
## 25    Ethiopia     33.8     43.9  51.666624     79.2    14       47       45
## 26      Guinea      5.4      6.2   7.269240     15.0    19       47       47
## 28     Senegal      6.1      6.9   7.713851     15.9    34       48       44
## 29        Mali      7.5      7.9   8.142373     21.6    17       46       51
## 31     Somalia      5.3      7.8   8.424269     12.8    30       47       47
## 32 Afghanistan     17.2     15.4  15.862293     32.2    16       48       44
## 33       Sudan     20.0     22.9  24.971806     51.3    21       47       44
## 36       Yemen      6.2      6.3   7.160981     13.9    12       48       52
## 39     Bolivia      6.0      6.4   6.706854     17.0    45       42       35
## 46 DominicanR.      6.4      6.4   7.240793     13.1    52       35       28
## 48     Ecuador      8.8      9.6  10.506668     23.4    44       41       30
## 52       Haiti      6.3      5.9   6.142141     14.1    26       36       45
## 53    Honduras      4.1      4.6   5.259699     12.0    37       44       37
## 55        Peru     19.2     20.2  21.905604     48.8    65       37       28
##    Death_82 Death_Rt Babymt82 Babymort Life_Exp Gnp_82 Gnp_86  Gdp_Cap  Log_Gdp
## 21       28       18      193      140       35    360    230 229.9130 2.361563
## 23       15       14      120      110       50    380    350 376.8009 2.576112
## 24       18       14      133      136       48    140    160 173.9379 2.240394
## 25       23       15      142      116       43    140    120 127.7420 2.106334
## 26       19       22      147      147       40    310     NA 343.9149 2.536451
## 28       18       14      141       87       43    490    420 648.1847 2.811699
## 29       22       21      137      116       42    180    180 238.2598 2.377051
## 31       21       15      143      125       43    290    280 201.7979 2.304917
## 32       23       18      205      154       37     NA     NA 189.1278 2.276755
## 33       17       14      118      107       48    440    320 340.3839 2.531969
## 36       21       17      154      129       44    500    550 768.0512 2.885390
## 39       16       13      124      125       51    570    600 685.8655 2.836239
## 46        9        7       64       62       63   1330    710 704.3427 2.847784
## 48        9        7       70       61       64   1350   1160 932.7410 2.969761
## 52       14       16      108      107       53    300    330 390.7432 2.591891
## 53       10        7       82       62       60    660    740 836.5498 2.922492
## 55       12        8       99       67       59   1310   1090 862.7930 2.935907
##      Educ_84      Educ   Health84     Health    Mil_84       Mil Govern
## 21 14.285714 10.000000 10.0000000  6.2500000        NA  2.500000      1
## 23  7.201690  7.654563  1.5628300  0.6967615 23.220697 23.267910      6
## 24  2.815433  3.554275  0.5943691  0.9798271  2.554745  2.372719      6
## 25  4.171598  4.943052  1.9526627  1.1845103 10.798817 10.136674      5
## 26 11.851852  9.193548  4.2592593  3.0645161        NA  9.193548      5
## 28 20.163934 19.710145  5.4098361  4.7826087 12.245902  9.855072      1
## 29  5.333333  5.189873  2.5333333  1.1392405 10.173333  3.924051      3
## 31  5.471698 11.923077  2.0754717  0.3846154 11.830189  8.846154      3
## 32  4.941860  1.000000  1.6860465         NA 10.116279        NA      3
## 33 20.650000 15.414847  1.0000000  0.7860262  9.000000 22.794760      1
## 36 47.258065 40.476190 11.4516129  8.5714286 97.419355 54.444444      5
## 39 21.000000 13.281250  6.6666667  2.1875000 38.333333 13.437500      1
```

```
## 46 20.468750 13.281250 15.1562500 11.7187500 15.031250 11.562500      1
## 48 62.045455 40.104167 18.0681818 12.0833333 24.659091 18.125000      1
## 52  3.333333  4.576271  3.0158730  3.2203390  3.619048  5.593220      6
## 53 29.268293 36.956522 12.1951220 19.3478261 43.170732 43.913043      6
## 55 30.937500 18.217822 11.7187500 11.5346535 62.604167 74.405941      1
##      Leader       Gov Gnp B_To_D82 Urban.1 Lifeexpm Lifeexpf Literacy    Group
## 21  Islamic Democracy   U 1.750000   rural       46       50     25.1  Islamic
## 23  Islamic  Military   U 2.866667   rural       56       57     26.0  Islamic
## 24  Islamic  Military   U 2.722222   rural       54       53     29.0  Islamic
## 25  Islamic  Military   U 2.043478   rural       49       52     55.2  Islamic
## 26  Islamic  Military   U 2.473684   rural       40       44     20.0  Islamic
## 28  Islamic Democracy   U 2.666667   rural       53       56     28.1  Islamic
## 29  Islamic  OneParty   U 2.090909   rural       45       47     18.0  Islamic
## 31  Islamic  OneParty   U 2.238095   rural       53       54     11.6  Islamic
## 32  Islamic  OneParty     2.086957   rural       47       46     12.0  Islamic
## 33  Islamic Democracy   U 2.764706   rural       51       55     31.0  Islamic
## 36  Islamic  Military   U 2.285714   rural       48       49     15.0  Islamic
## 39 Catholic Democracy   U 2.625000    city       52       56     63.0 NewWorld
## 46 Catholic Democracy   D 3.888889    city       65       69     74.0 NewWorld
## 48 Catholic Democracy   D 4.555556    city       64       68     85.0 NewWorld
## 52 Catholic  Military   U 2.571429   rural       52       55     23.0 NewWorld
## 53 Catholic  Military   U 4.400000   rural       64       67     56.0 NewWorld
## 55 Catholic Democracy   D 3.083333    city       62       66     80.0 NewWorld
##      B_To_D Group.1      Gdp Lat Lon Mcdonald
## 21 2.666667       2 Emerging  -4  14       NA
## 23 3.071429       2 Emerging  22 -79       NA
## 24 3.000000       2 Emerging  35  33       NA
## 25 3.000000       2 Emerging  49  16       NA
## 26 2.136364       2 Emerging  56  10       NA
## 28 3.142857       2 Emerging  52  13       NA
## 29 2.428571       2 Emerging   0 -78       NA
## 31 3.133333       2 Emerging   9  38       NA
## 32 2.444444       2 Emerging -18 178       NA
## 33 3.142857       2 Emerging  66  25       NA
## 36 3.058824       2 Emerging  13 -15       NA
## 39 2.692308       3 Emerging  14 -90       NA
## 46 4.000000       3 Emerging  53  -7       NA
## 48 4.285714       3 Emerging   7  -5       NA
## 52 2.812500       3 Emerging   6  -9       NA
## 53 5.285714       3 Emerging  25  20       NA
## 55 3.500000       3 Emerging   3 101       NA
```

In plain English, this says, pull out the rows of raw data where `Gdp_Cap` is greater than -1,000 AND less than 1,000.

How many rows are there? Now we can try the tidyverse/dplyr option with piping:

```
dat %>% filter(Gdp_Cap>-1000&Gdp_Cap<1000)
```

```
##        Country Pop_1983 Pop_1986   Pop_1990 Pop_2020 Urban Birth_82 Birth_Rt
## 1       Gambia      0.7      0.8   0.848147      1.7    18       49       48
## 2     Pakistan     94.7    101.9 114.649408    194.4    28       43       43
## 3   Bangladesh     95.9    104.1 118.433064    245.0    11       49       42
## 4     Ethiopia     33.8     43.9  51.666624     79.2    14       47       45
```

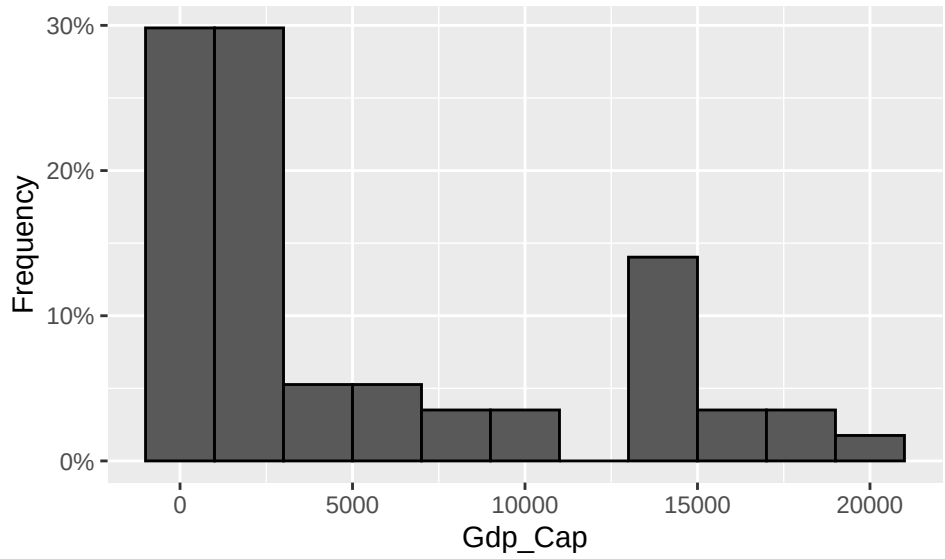
```
## 5       Guinea      5.4      6.2   7.269240     15.0    19       47       47
## 6      Senegal      6.1      6.9   7.713851     15.9    34       48       44
## 7         Mali      7.5      7.9   8.142373     21.6    17       46       51
## 8      Somalia      5.3      7.8   8.424269     12.8    30       47       47
## 9  Afghanistan     17.2     15.4  15.862293     32.2    16       48       44
## 10       Sudan     20.0     22.9  24.971806     51.3    21       47       44
## 11       Yemen      6.2      6.3   7.160981     13.9    12       48       52
## 12     Bolivia      6.0      6.4   6.706854     17.0    45       42       35
## 13 DominicanR.      6.4      6.4   7.240793     13.1    52       35       28
## 14     Ecuador      8.8      9.6  10.506668     23.4    44       41       30
## 15       Haiti      6.3      5.9   6.142141     14.1    26       36       45
## 16    Honduras      4.1      4.6   5.259699     12.0    37       44       37
## 17        Peru     19.2     20.2  21.905604     48.8    65       37       28
##    Death_82 Death_Rt Babymt82 Babymort Life_Exp Gnp_82 Gnp_86  Gdp_Cap  Log_Gdp
## 1        28       18      193      140       35    360    230 229.9130 2.361563
## 2        15       14      120      110       50    380    350 376.8009 2.576112
## 3        18       14      133      136       48    140    160 173.9379 2.240394
## 4        23       15      142      116       43    140    120 127.7420 2.106334
## 5        19       22      147      147       40    310     NA 343.9149 2.536451
## 6        18       14      141       87       43    490    420 648.1847 2.811699
## 7        22       21      137      116       42    180    180 238.2598 2.377051
## 8        21       15      143      125       43    290    280 201.7979 2.304917
## 9        23       18      205      154       37     NA     NA 189.1278 2.276755
## 10       17       14      118      107       48    440    320 340.3839 2.531969
## 11       21       17      154      129       44    500    550 768.0512 2.885390
## 12       16       13      124      125       51    570    600 685.8655 2.836239
## 13        9        7       64       62       63   1330    710 704.3427 2.847784
## 14        9        7       70       61       64   1350   1160 932.7410 2.969761
## 15       14       16      108      107       53    300    330 390.7432 2.591891
## 16       10        7       82       62       60    660    740 836.5498 2.922492
## 17       12        8       99       67       59   1310   1090 862.7930 2.935907
##      Educ_84      Educ   Health84     Health    Mil_84       Mil Govern
## 1  14.285714 10.000000 10.0000000  6.2500000        NA  2.500000      1
## 2   7.201690  7.654563  1.5628300  0.6967615 23.220697 23.267910      6
## 3   2.815433  3.554275  0.5943691  0.9798271  2.554745  2.372719      6
## 4   4.171598  4.943052  1.9526627  1.1845103 10.798817 10.136674      5
## 5  11.851852  9.193548  4.2592593  3.0645161        NA  9.193548      5
## 6  20.163934 19.710145  5.4098361  4.7826087 12.245902  9.855072      1
## 7   5.333333  5.189873  2.5333333  1.1392405 10.173333  3.924051      3
## 8   5.471698 11.923077  2.0754717  0.3846154 11.830189  8.846154      3
## 9   4.941860  1.000000  1.6860465         NA 10.116279        NA      3
## 10 20.650000 15.414847  1.0000000  0.7860262  9.000000 22.794760      1
## 11 47.258065 40.476190 11.4516129  8.5714286 97.419355 54.444444      5
## 12 21.000000 13.281250  6.6666667  2.1875000 38.333333 13.437500      1
## 13 20.468750 13.281250 15.1562500 11.7187500 15.031250 11.562500      1
## 14 62.045455 40.104167 18.0681818 12.0833333 24.659091 18.125000      1
## 15  3.333333  4.576271  3.0158730  3.2203390  3.619048  5.593220      6
## 16 29.268293 36.956522 12.1951220 19.3478261 43.170732 43.913043      6
## 17 30.937500 18.217822 11.7187500 11.5346535 62.604167 74.405941      1
##      Leader       Gov Gnp B_To_D82 Urban.1 Lifeexpm Lifeexpf Literacy    Group
## 1   Islamic Democracy   U 1.750000   rural       46       50     25.1  Islamic
## 2   Islamic  Military   U 2.866667   rural       56       57     26.0  Islamic
## 3   Islamic  Military   U 2.722222   rural       54       53     29.0  Islamic
## 4   Islamic  Military   U 2.043478   rural       49       52     55.2  Islamic
```

```
## 5   Islamic  Military   U 2.473684   rural       40       44     20.0  Islamic
## 6   Islamic Democracy   U 2.666667   rural       53       56     28.1  Islamic
## 7   Islamic  OneParty   U 2.090909   rural       45       47     18.0  Islamic
## 8   Islamic  OneParty   U 2.238095   rural       53       54     11.6  Islamic
## 9   Islamic  OneParty     2.086957   rural       47       46     12.0  Islamic
## 10  Islamic Democracy   U 2.764706   rural       51       55     31.0  Islamic
## 11  Islamic  Military   U 2.285714   rural       48       49     15.0  Islamic
## 12 Catholic Democracy   U 2.625000    city       52       56     63.0 NewWorld
## 13 Catholic Democracy   D 3.888889    city       65       69     74.0 NewWorld
## 14 Catholic Democracy   D 4.555556    city       64       68     85.0 NewWorld
## 15 Catholic  Military   U 2.571429   rural       52       55     23.0 NewWorld
## 16 Catholic  Military   U 4.400000   rural       64       67     56.0 NewWorld
## 17 Catholic Democracy   D 3.083333    city       62       66     80.0 NewWorld
##      B_To_D Group.1      Gdp Lat Lon Mcdonald
## 1  2.666667       2 Emerging  -4  14       NA
## 2  3.071429       2 Emerging  22 -79       NA
## 3  3.000000       2 Emerging  35  33       NA
## 4  3.000000       2 Emerging  49  16       NA
## 5  2.136364       2 Emerging  56  10       NA
## 6  3.142857       2 Emerging  52  13       NA
## 7  2.428571       2 Emerging   0 -78       NA
## 8  3.133333       2 Emerging   9  38       NA
## 9  2.444444       2 Emerging -18 178       NA
## 10 3.142857       2 Emerging  66  25       NA
## 11 3.058824       2 Emerging  13 -15       NA
## 12 2.692308       3 Emerging  14 -90       NA
## 13 4.000000       3 Emerging  53  -7       NA
## 14 4.285714       3 Emerging   7  -5       NA
## 15 2.812500       3 Emerging   6  -9       NA
## 16 5.285714       3 Emerging  25  20       NA
## 17 3.500000       3 Emerging   3 101       NA
```

Same result! (With both of those options, the resulting table clearly shows how many rows there are, but remember that you can put `nrow()` around those lines of code to get the exact value of the number of rows.)

Note that instead of the count of cases in each bin on the y-axis, we can plot *Density* (the proportion of cases in each bin) using:

```
p3=ggplot(data=dat,aes(x=Gdp_Cap))+
  geom_histogram(aes(y = after_stat(count/sum(count))), binwidth=2000,col="black")+
  labs(y="Frequency")+scale_y_continuous(labels = scales::percent)
p3
```



Here again I'm using the `after_stat()` function to calculate something (in this case, the frequency of each bin, which is calculated as the count of that bin divided by the sum of counts for all bins). I also changed the y-axis to "Freqency" and changed the y labels to percentages to make them a bit more pretty. Try commenting out that line of code (using `#` at the beginning of the line) to see what the original plot would have looked like.

The y-axis now shows the probability of occurrence for each bar value. What are the values? You can use the `ggplot_build()` function like we did above to figure out what the y values are:

```
p3nums = ggplot_build(p3)
head(p3nums$data[[1]])
```

```
##           y count     x  xmin  xmax      density    ncount  ndensity
## 1 0.29824561    17     0 -1000  1000 1.491228e-04 1.0000000 1.0000000
## 2 0.29824561    17  2000  1000  3000 1.491228e-04 1.0000000 1.0000000
## 3 0.05263158     3  4000  3000  5000 2.631579e-05 0.1764706 0.1764706
## 4 0.05263158     3  6000  5000  7000 2.631579e-05 0.1764706 0.1764706
## 5 0.03508772     2  8000  7000  9000 1.754386e-05 0.1176471 0.1176471
## 6 0.03508772     2 10000  9000 11000 1.754386e-05 0.1176471 0.1176471
##   flipped_aes PANEL group ymin       ymax colour   fill linewidth linetype
## 1       FALSE     1    -1    0 0.29824561  black grey35       0.5        1
## 2       FALSE     1    -1    0 0.29824561  black grey35       0.5        1
## 3       FALSE     1    -1    0 0.05263158  black grey35       0.5        1
## 4       FALSE     1    -1    0 0.05263158  black grey35       0.5        1
## 5       FALSE     1    -1    0 0.03508772  black grey35       0.5        1
## 6       FALSE     1    -1    0 0.03508772  black grey35       0.5        1
##   alpha
## 1    NA
## 2    NA
## 3    NA
## 4    NA
## 5    NA
## 6    NA
```

(e.g., 30% of `Gdp_Cap` cases, a total of 17, have values between -1,000 and 1,000).

3) Try out data manipulation and plotting on your own!

Welcome to this practice worksheet! We will be investigating some trends in the body size of different large mammals. The data for this worksheet is titled "animal_weights.csv".

I suggest trying to run through each section without looking at the code or the output. Once you feel you have answered the question or completed the task, you can check and see if your answer matches. If it does, then you can also check to see if the code matches up.

If you are stuck on a section, try looking at the output that is generated instead of going right to the code. Compare the output to the dataframe that you are starting with. What has changed? How can you write code to make those changes? Sometimes seeing the end product may jog your memory! And as always, *Google is your friend*!!! There are certain components of these tasks, particularly with graphing, that we have not yet taught you in class, so you will *need* to search around.

Have fun!

Questions:

```
## Call to tidyverse packages
library(tidyverse)

urlRemote  = "https://raw.githubusercontent.com/"
pathGithub = "calvin-munson/R-DataScience-workshops/master/practice_worksheets/animal_weights/"
fileName   = "animal_weight.csv"

animal_weights=read.csv(paste0(urlRemote, pathGithub, fileName))
```

1. Load the tidyverse packages from the library and read in the .csv file called "animal_weight.csv". Store it in as a data object called `animal_weights`.

2. Write code to: a) Look at the header of the data, b) find the number of columns in the data, and c) find the number of rows in the data.

3. Identify the unique species and unique age classes present in the dataset

4. Write code to calculate the mean weight of each species in the dataset. Hint: group_by() and summarise() are your friends

5. Calculate the same average weight, but this time for each species/age class combination in the dataset (for instance, what does the average elephant child weight? Elephant adult? Hippo child? etc)

6. Next, calculate both mean weight AND mean height in the same data frame for each species/age class combination

7. You may have noticed that there is also a column for the sex of the animal, but that the data only exists for the Hippos in our dataset. Create a new dataframe called hippo_stats, which a) only includes data from Hippos, and b) has the mean weight and height for each species, sex, and age combination in the dataset.

8. Calculate Body Mass Index (BMI) for each individual animal. In humans, BMI is calculated as:

$$BMI = kg/m^2$$

Where BMI is body mass index, kg is mass in kilograms, and m is height in meters.

Taking the original dataframe, create a new column that contains a calculated BMI for each individual animal (this is obviously totally meaningless in terms of actual data on these species!)

Hint: mutate() is your friend! Also, if you are stuck on how to square a value in R, check out our even better friend, Google!

9. Using ggplot, create a set of boxplots showing the distribution of weights for each species Hint: Think about what value you want on which axis

10. Create a more detailed boxplot, this time including the fill of the boxplot as the age class of the species Additionally, change a few of the features of the plot:

a) Add neat x and y axis labels
b) Add a plot title
c) Change the y-axis limits to include 0 as a minimum
d) Jitter the raw data points on top

Hint: Again, use Google if you're stuck on how to change these features. The website Stack Overflow is a great place to go.

```r
# a)
head(animal_weights)
```

2 ANSWER.

```
##    species sex age_class individual weight_kg height_m
## 1 Elephant         Child          A      8000      4.0
## 2 Elephant         Child          B      6000      3.0
## 3 Elephant         Adult          C     10000      7.0
## 4 Elephant         Adult          D      9000      6.5
## 5 Elephant         Child          E      7000      4.0
## 6 Elephant         Child          F      5500      3.0
```

```r
# b)
ncol(animal_weights)
```

```
## [1] 6
```

```r
# c)
nrow(animal_weights)
```

```
## [1] 24
```

3 ANSWER. Unique species:

```r
# Unique species:
unique(animal_weights$species)
```

```
## [1] "Elephant" "Hippo"    "Rhino"
```

Unique age classes:

```r
# Unique age classes:
unique(animal_weights$age_class)
```

```
## [1] "Child" "Adult"
```

Note: Using the dollar sign to select a column of a dataframe turns that column into a vector of values. You can also use a pipeline to pipe that vector into the function you want to use

```r
animal_weights$species %>% unique()
```

```
## [1] "Elephant" "Hippo"    "Rhino"
```

```
animal_weights %>%
  group_by(species) %>%
  summarise(mean_weight = mean(weight_kg, na.rm = TRUE))
```

4 ANSWER

```
## # A tibble: 3 x 2
##   species  mean_weight
##   <chr>          <dbl>
## 1 Elephant       7875
## 2 Hippo          1888.
## 3 Rhino          1781.
```

```
animal_weights %>%
  group_by(species, age_class) %>%
  summarise(mean_weight = mean(weight_kg, na.rm = TRUE))
```

5 ANSWER

```
## # A tibble: 6 x 3
## # Groups:   species [3]
##   species  age_class mean_weight
##   <chr>    <chr>           <dbl>
## 1 Elephant Adult            9125
## 2 Elephant Child            6625
## 3 Hippo    Adult            2300
## 4 Hippo    Child            1475
## 5 Rhino    Adult            2225
## 6 Rhino    Child            1338.
```

```
animal_weights %>%
  group_by(species, age_class) %>%
  summarise(mean_weight = mean(weight_kg, na.rm = TRUE),
            mean_height = mean(height_m, na.rm = TRUE))
```

6 ANSWER

```
## # A tibble: 6 x 4
## # Groups:   species [3]
##   species  age_class mean_weight mean_height
##   <chr>    <chr>           <dbl>       <dbl>
## 1 Elephant Adult            9125         6.6
## 2 Elephant Child            6625         3.5
## 3 Hippo    Adult            2300         1.48
## 4 Hippo    Child            1475         0.9
## 5 Rhino    Adult            2225         1.62
## 6 Rhino    Child            1338.        1.12
```

```r
hippo_stats <- animal_weights %>%
  # a) Filter the original dataframe to only include Hippos
  filter(species == "Hippo") %>%
  # b) Group by all 3 columns of interest and summarise
  group_by(species, sex, age_class) %>%
  summarise(mean_weight = mean(weight_kg, na.rm = TRUE),
            mean_height = mean(height_m, na.rm = TRUE))

hippo_stats
```

7 ANSWER

```
## # A tibble: 4 x 5
## # Groups:   species, sex [2]
##   species sex    age_class mean_weight mean_height
##   <chr>   <chr>  <chr>           <dbl>       <dbl>
## 1 Hippo   Female Adult            1500        1.4
## 2 Hippo   Female Child            1200        0.8
## 3 Hippo   Male   Adult            3100        1.55
## 4 Hippo   Male   Child            1750        1
```

```r
animal_weights %>%
  mutate(BMI = weight_kg/height_m^2)
```

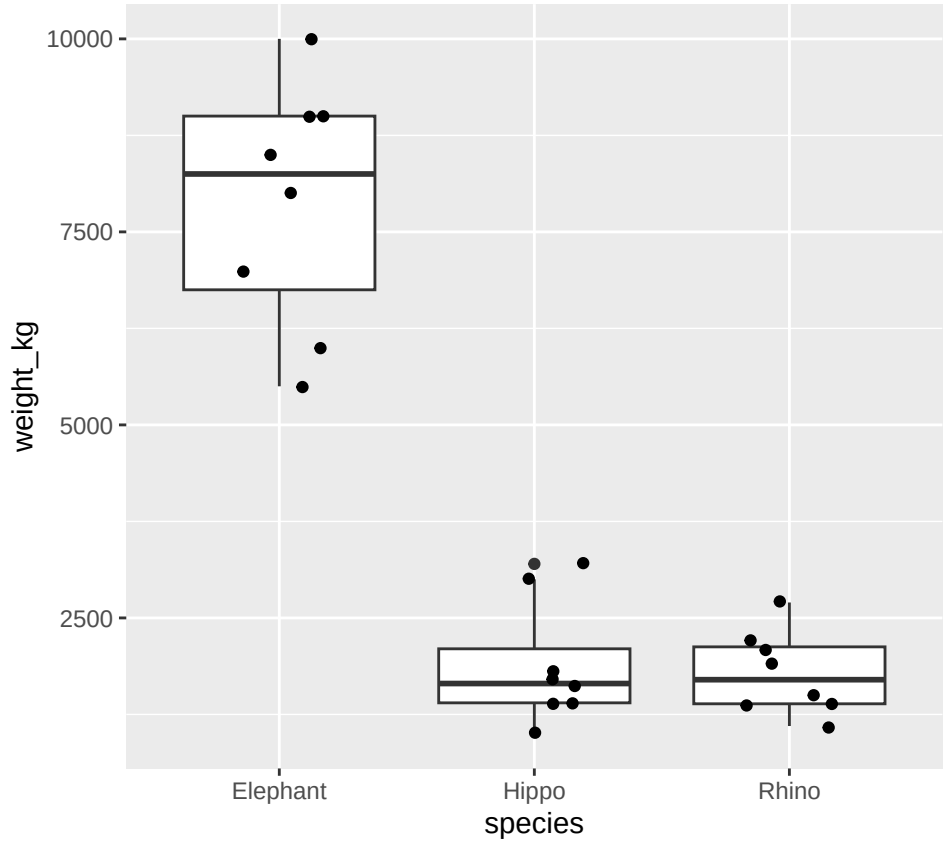
8 ANSWER

```
##     species    sex age_class individual weight_kg height_m       BMI
## 1  Elephant            Child          A      8000      4.0  500.0000
## 2  Elephant            Child          B      6000      3.0  666.6667
## 3  Elephant            Adult          C     10000      7.0  204.0816
## 4  Elephant            Adult          D      9000      6.5  213.0178
## 5  Elephant            Child          E      7000      4.0  437.5000
## 6  Elephant            Child          F      5500      3.0  611.1111
## 7  Elephant            Adult          G      9000      6.4  219.7266
## 8  Elephant            Adult          H      8500      6.5  201.1834
## 9     Hippo Female     Adult          A      1400      1.4  714.2857
## 10    Hippo Female     Adult          B      1600      1.4  816.3265
## 11    Hippo   Male     Adult          C      3000      1.5 1333.3333
## 12    Hippo   Male     Adult          D      3200      1.6 1250.0000
## 13    Hippo Female     Child          E      1000      0.7 2040.8163
## 14    Hippo Female     Child          F      1400      0.9 1728.3951
## 15    Hippo   Male     Child          G      1700      1.0 1700.0000
## 16    Hippo   Male     Child          H      1800      1.0 1800.0000
## 17    Rhino            Adult          A      1900      1.5  844.4444
## 18    Rhino            Adult          B      2200      1.6  859.3750
## 19    Rhino            Adult          C      2100      1.6  820.3125
## 20    Rhino            Adult          D      2700      1.8  833.3333
## 21    Rhino            Child          E      1500      1.3  887.5740
```

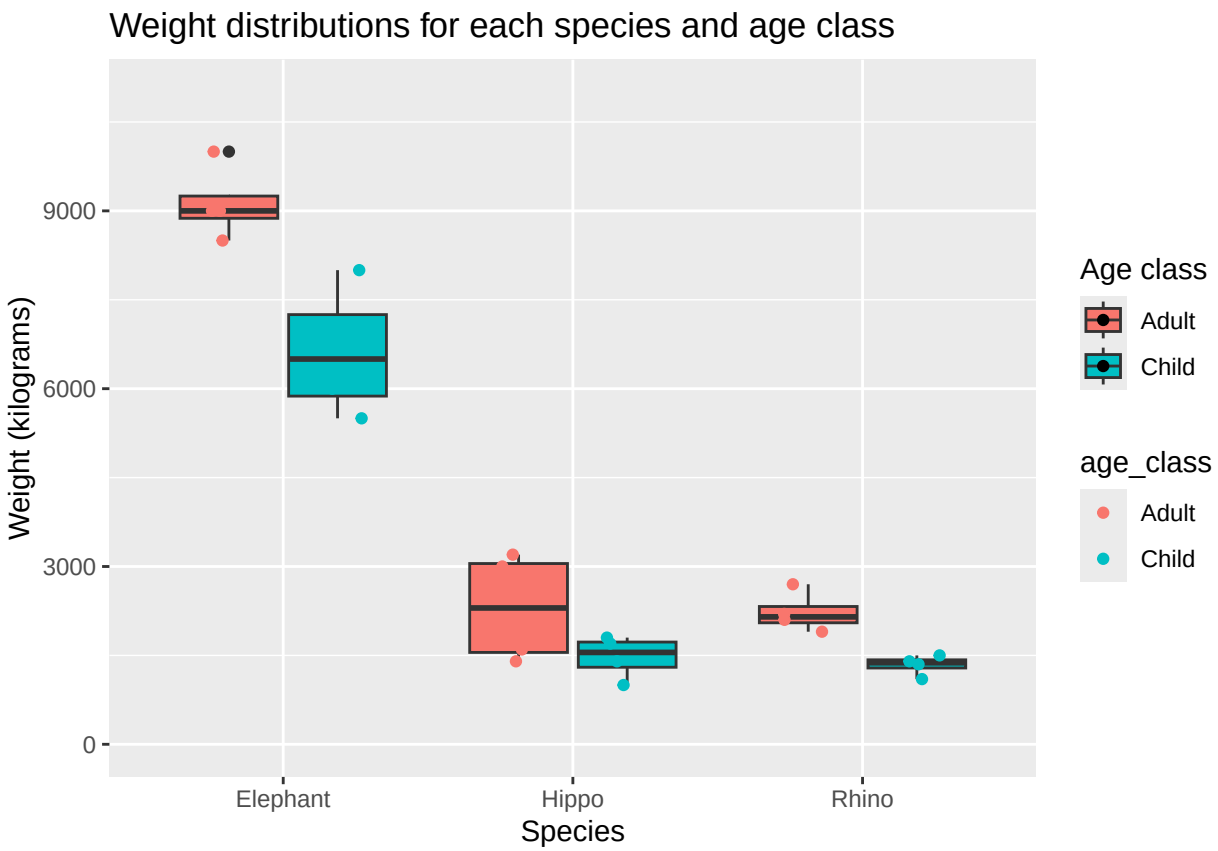
```
## 22    Rhino            Child          F      1400      1.2  972.2222
## 23    Rhino            Child          G      1350      1.1 1115.7025
## 24    Rhino            Child          H      1100      0.9 1358.0247
```

```
animal_weights %>%
  ggplot(aes(x = species, y = weight_kg)) +
  geom_boxplot()+geom_jitter(width=0.2)
```

9 ANSWER




```
animal_weights %>%
  ggplot(aes(x = species, y = weight_kg, fill = age_class)) +
  geom_boxplot() +
  # a)
  labs(x = "Species", y = "Weight (kilograms)", fill = "Age class") +
  # b)
  ggtitle("Weight distributions for each species and age class") +
  # c)
  ylim(0,11000)+
  # d)
   geom_point(aes(col=age_class),position=position_jitterdodge(jitter.width = .2))
```



10 ANSWER

Note: If you assigned a column to the "fill" aesthetic (same thing goes for other aesthetics, like "color" or "shape"), you can also specify a new label for it as I did in the labs() arguement. That label will show up over the legend.

4) Merging

Another useful data function is `merge()`. This is a very useful data function that allows combinations of files using columns with the same name. Imagine you want to make a map of mussel cover for the entire west coast of California and have a file with mussel cover as a function of site and year (`mussels by site and year.csv`) and a separate file that contains latitude and longitude locations for each site (`lat and long by site.csv`). You want to merge these two files so you have site, year, latitude, longitude, and mussel cover in a single dataframe.

Let's clear our working directory and bring in those two datasets.

```
rm(list=ls()) #fresh start
loc=read.csv("lat and long by site.csv")
muss=read.csv("mussels by site and year.csv")
```

The obvious problem is that the mussel file has 1097 rows (one for each site and year the site was sampled) whereas the location file has 89 rows (one for each site).

```
nrow(loc)
```

```
## [1] 89
```

```
nrow(muss)
```

```
## [1] 1097
```

One very slow and tedious and error-risky way of accomplishing this is to manually copy and paste the lat and long for each site into the mussel cover spreadsheet. (If you were using Excel you might use the `vlookup()` function). Instead, we can use the `merge()` function, in which you specify the two dataframes to merge and the common column name!

```
colnames(loc)
```

```
## [1] "site"      "latitude"  "longitude"
```

```
colnames(muss)
```

```
## [1] "site"             "Year"             "Mussel.cover...."
```

It looks like `site` is the column name that is shared by the two datasets, so we'll want to merge by that column. As an aside, see how weird the `Mussel.cover....` column name is? This is what happens when you have spaces in column names. R hates spaces in column names. I always carefully check each column name in Excel before saving as a .csv file and importing into R to avoid things like this. Before we merge, let's rename the `Mussel.cover....` column.

The easy, lazy, risky way to do this in base R would be something like `colnames(muss)[3]="MusselCover"` (in plain English, "change the third column name of the muss dataset to"MusselCover".

But I want to teach you the tidyverse (dplyr) version - you can name lots of columns at once if you want to, and you specify the old column name which makes this method less prone to errors. This is the general format:

dataframe = dataframe %>% rename("newname1" = "oldname1", "newname2" = "oldname2")

Try it out!

```
muss = muss %>%
  rename("MusselCover" = "Mussel.cover....")

colnames(muss) #check that it worked
```

```
## [1] "site"        "Year"        "MusselCover"
```

Now let's merge the two datasets by `site`. Check out the helpfile for `merge()` using `help(merge)`. You'll see that the function requires an "x" data frame, a "y" data frame, and a "by" specification for the shared column name. There are lots of other options, too.

```
mussloc=merge(x=loc,y=muss,by="site")
head(mussloc)
```

```
##       site latitude longitude Year MusselCover
## 1 Alcatraz   37.825 -122.4219 2006  0.08333333
## 2 Alcatraz   37.825 -122.4219 2007  0.00000000
## 3 Alcatraz   37.825 -122.4219 2008  0.00000000
## 4 Alcatraz   37.825 -122.4219 2009  0.00000000
## 5 Alcatraz   37.825 -122.4219 2010  0.00000000
## 6 Alcatraz   37.825 -122.4219 2011  0.23076923
```

Make sure to check that the resulting dataframe, `mussloc`, looks OK (i.e., has the appropriate number of rows and columns and data). You have all the tools to do this now!

One additional way to check that the merge worked properly is using the `aggregate()` function (I am adding `head()` to this line of code so the RMarkdown PDF I give you doesn't spit out two pages of data):

```
head(aggregate(latitude~site,data=mussloc,FUN="sd"))
```

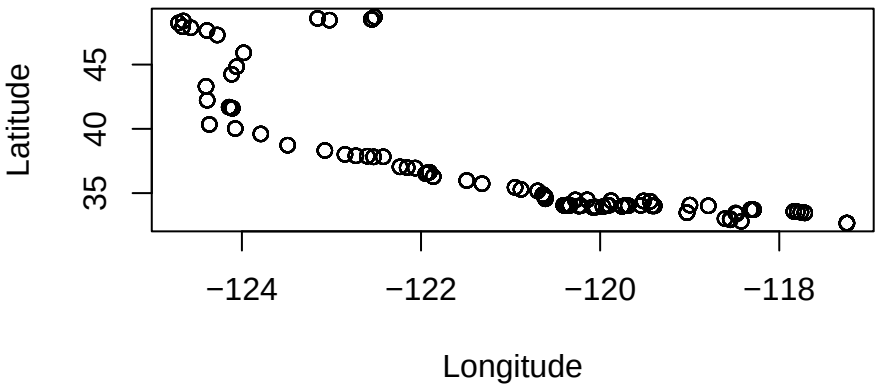
```
##            site latitude
## 1      Alcatraz        0
## 2       Alegria        0
## 3 American Camp        0
## 4 Andrew Molera        0
## 5  Arroyo Hondo        0
## 6     Bird Rock        0
```

Looks like the standard deviation of latitude is 0 for all sites, which indicates that the merge worked properly!

Remember as you get more and more comfortable in R that it is SUPER easy to make mistakes - and there's no harm in taking a few seconds to check your work. (trust me, finding your mistake early will save a lot of time and effort in the future).

Now let's make a map! A super basic one, that assumes latitude and longitude can be plotted on the x and y axes without any sort of correction for the fact that the earth is not flat....

```
plot(latitude~longitude,data=mussloc,
     ylab="Latitude",xlab="Longitude")
```



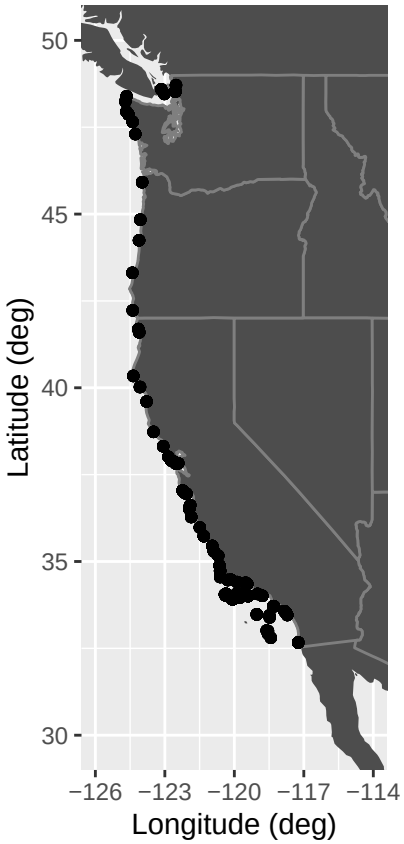
...or we can do it for real...

```
library(ggmap) #package for making pretty maps with ggplot
library(mapdata) #package for pulling out map data
yrange=c(30,50)
xrange=c(-126,-114)

#world map data
w=map_data("world",ylim=yrange,xlim=xrange)

#states data
states=map_data("state")
```

```
#create plot
 z=ggplot()+labs(y= "Latitude (deg)", x = "Longitude (deg)")+
  geom_polygon(data=w,aes(x=long,y=lat,group=group),fill="grey30")+
   geom_polygon(data=states,aes(x=long,y=lat,fill=region,group=group),
                fill="grey30",color="grey50")+
  coord_fixed(1.5,xlim=xrange,ylim=yrange)+
  geom_point(data=mussloc,aes(x=longitude,y=latitude))
 z
```

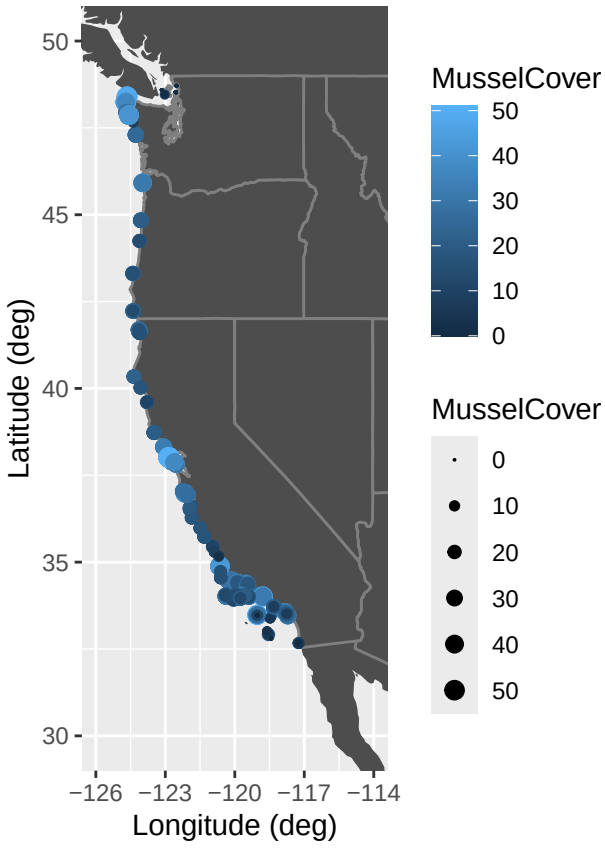


See if you can figure out what each line of that code does... if you can't, ask us :).

The plot above is a bit different from other plots we've been making because we use multiple data frames to layer polygons and points onto the plot (the `w` dataset that has the world map, the `states` dataset that has the United States, and the `mussloc` dataset that has the mussel data). Because we're using multiple datasets for the plot, we didn't actually specify a dataset within the top of the plot (in `ggplot()`). This is fine!

What if we want to show the proportional cover of mussels using both the size and color of points?

```
#create plot
 z=ggplot()+labs(y= "Latitude (deg)", x = "Longitude (deg)")+
  geom_polygon(data=w,aes(x=long,y=lat,group=group),fill="grey30")+
  geom_polygon(data=states,aes(x=long,y=lat,fill=region,group=group),
               fill="grey30",color="grey50")+
  coord_fixed(1.5,xlim=xrange,ylim=yrange)+
  geom_point(data=mussloc,aes(x=longitude,y=latitude,size=MusselCover,color=MusselCover))+
  scale_size(range = c(0, 3))
 z
```



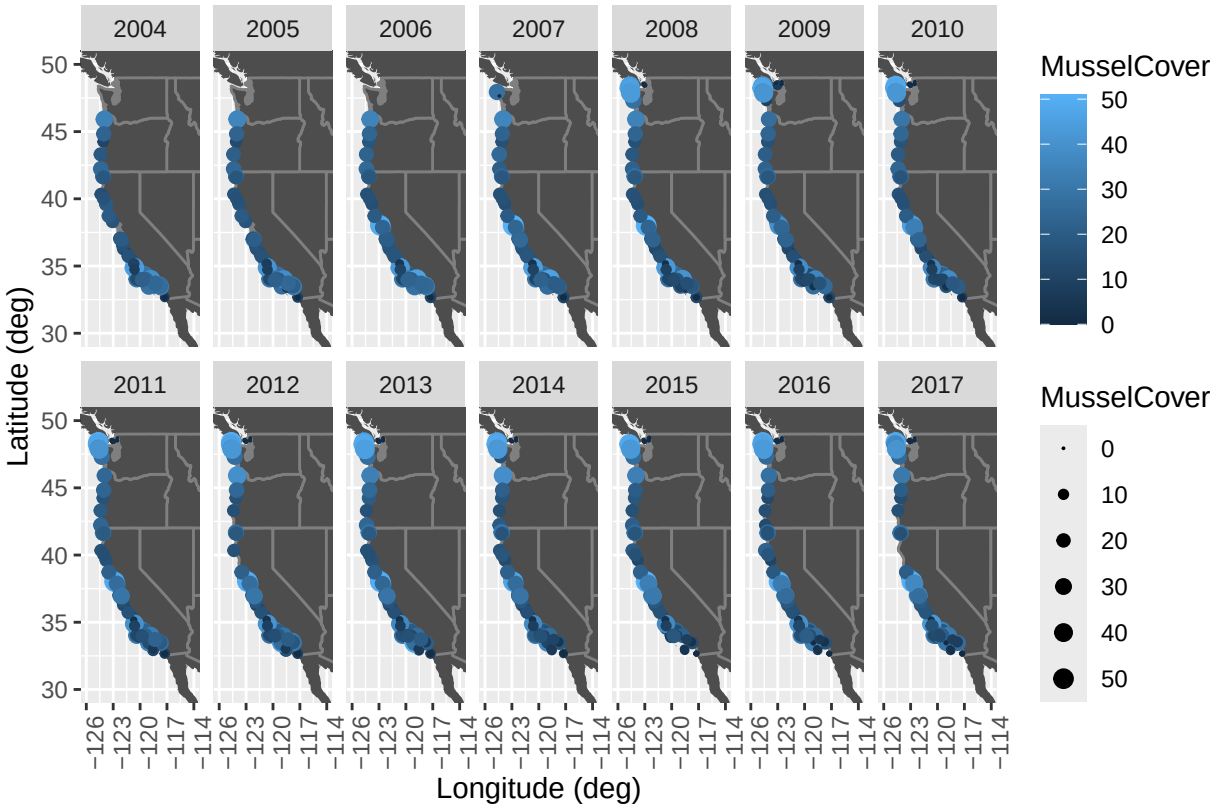
So pretty! How do we save that high resolution figure?

```
ggsave('MusselCover.png')
```

Using `help(ggsave)` you can see other possible inputs to the function, like height and width and units, that you might want to use to save the figure in an easier-to-read way.

Wait though... we have data over a number of years, right? Say we want to show these temporal patterns.... we can use `facet_wrap()` by year with 7 columns (see the end of the next code chunk)

```
z=ggplot()+labs(y= "Latitude (deg)", x = "Longitude (deg)")+
 geom_polygon(data=w,aes(x=long,y=lat,group=group),fill="grey30")+
 geom_polygon(data=states,aes(x=long,y=lat,fill=region,group=group),
              fill="grey30",color="grey50")+
 coord_fixed(1.5,xlim=xrange,ylim=yrange)+
 geom_point(mussloc,mapping=aes(x=longitude,y=latitude,size=MusselCover,color=MusselCover))+
 scale_size(range = c(0, 3))+
 theme(axis.text.x=element_text(angle=90))
z+facet_wrap(~ Year, ncol=7)
```



PS: I added `+theme(axis.text.x=element_text(angle=90))` at the end of the code chunk to rotate the x-axis labels by 90 degrees so that you can actually read them when the figures are smashed together.

et voila! nice job :)