Data Management in R: Day 1

Course by : Laura R Stein

Instructor: Ed Higgins

TAs: Sam Eliades + Megan Malish

4th August 2021

clear R's brain

```
rm(list=ls())
```

Install the packages

```
install.packages( "tidyverse" )
```

Note that you only have to do this one time

Load the packages

```
library(tidyverse )
```

Do this for every R session where you use the package(s)

We are ready to begin!

We will be using the dataset 'iris'. Note this is already loaded.

```
head(iris)
names(iris)
str(iris)
dim(iris)
summary(iris)
```

Using the pipe operator

```
iris %>% head
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
...
```

For levity, I am only showing top 5 rows. Your results will be longer.

Base R finding mean of sepal lengths

```
mean.sepal <- mean(iris$Sepal.Length)
mean.sepal
```

```
## [1] 5.843333
```

Using %>%

```
mean.sepal2 <- iris$Sepal.Length %>% mean()
mean.sepal2
```

```
## [1] 5.843333
```

```
iris$Sepal.Length %>% mean() -> mean.sepal3
mean.sepal3
```

```
## [1] 5.843333
```

Note that there are many ways to find the same answer in R!

Using filter() to choose rows Base R

equivalent

```
iris[iris$Species ==  "virginica",]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 101          6.3         3.3          6.0         2.5 virginica
## 102          5.8         2.7          5.1         1.9 virginica
## 103          7.1         3.0          5.9         2.1 virginica
## 104          6.3         2.9          5.6         1.8 virginica
...
```

using dplyr::filter()

```
filter(iris, Species ==  "virginica" )
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1           6.3         3.3          6.0         2.5 virginica
## 2           5.8         2.7          5.1         1.9 virginica
## 3           7.1         3.0          5.9         2.1 virginica
## 4           6.3         2.9          5.6         1.8 virginica
...
```

Equivalent code with %>% pipe

```
iris %>%
  filter(Species ==  "virginica" )
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1           6.3         3.3          6.0         2.5 virginica
## 2           5.8         2.7          5.1         1.9 virginica
## 3           7.1         3.0          5.9         2.1 virginica
## 4           6.3         2.9          5.6         1.8 virginica
...
```

Separate “and” conditions with a comma

```
iris %>%    filter(Species == "virginica",
Sepal.Length > 7.5)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          7.6         3.0          6.6         2.1 virginica
## 2          7.7         3.8          6.7         2.2 virginica
## 3          7.7         2.6          6.9         2.3 virginica
## 4          7.7         2.8          6.7         2.0 virginica
## 5          7.9         3.8          6.4         2.0 virginica
## 6          7.7         3.0          6.1         2.3 virginica
```

Picking rows with select()

```
iris %>%
  select(Species, Petal.Length)
```

```
##        Species Petal.Length
## 1       setosa          1.4
## 2       setosa          1.4
## 3       setosa          1.3
## 4       setosa          1.5
...
```

Select everything except for certain rows using (-)

```
iris %>%
  select(-Species)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1            5.1         3.5          1.4         0.2
## 2            4.9         3.0          1.4         0.2
## 3            4.7         3.2          1.3         0.2
## 4            4.6         3.1          1.5         0.2
...
```

Combining filter() and select() with %>%

```
iris %>%
  filter(Species ==  "virginica", Sepal.Width >  3.5) %>%
  select(Petal.Width)
```

```
##   Petal.Width
## 1         2.5
## 2         2.2
## 3         2.0
```

Creating new columns with mutate()

```
iris %>%
  mutate(Sepal.Area = Sepal.Width * Sepal.Length)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 2            4.9         3.0          1.4         0.2     setosa
## 3            4.7         3.2          1.3         0.2     setosa
## 4            4.6         3.1          1.5         0.2     setosa
...
```

Combine verbs using %>% and multiple functions

Combining verbs: Which flowers have sepal areas less than 15, ordered by area?

```
iris %>%
  mutate(Sepal.Area = Sepal.Width * Sepal.Length) %>%
  filter(Sepal.Area <  15) %>%
  arrange(Sepal.Area)
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species Sepal.Area
## 1           5.0         2.0          3.5         1.0 versicolor      10.00
## 2           4.5         2.3          1.3         0.3     setosa      10.35
## 3           5.0         2.3          3.3         1.0 versicolor      11.50
## 4           4.9         2.4          3.3         1.0 versicolor      11.76
...
```

Combining verbs: Which flowers have sepal areas less than 15, ordered by descending area?

```
iris %>%
  mutate(Sepal.Area = Sepal.Width * Sepal.Length) %>%
  filter(Sepal.Area <  15) %>%
  arrange(desc(Sepal.Area))
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species Sepal.Area
## 1           4.8         3.1          1.6         0.2     setosa      14.88
## 2           5.7         2.6          3.5         1.0 versicolor      14.82
## 3           4.6         3.2          1.4         0.2     setosa      14.72
## 4           4.9         3.0          1.4         0.2     setosa      14.70
...
```

Combining verbs: How many flowers have sepal areas less than 15?

```
iris %>%
  mutate(Sepal.Area = Sepal.Width * Sepal.Length) %>%
  filter(Sepal.Area <  15) %>%
  tally()
```

```
##    n
## 1 29
```

Combining verbs: How many flowers of each species have sepal areas less than 15?

```
iris %>%
  mutate(Sepal.Area = Sepal.Width * Sepal.Length) %>%
  filter(Sepal.Area <  15) %>%
  group_by(Species) %>%
  tally()
```

```
## # A tibble: 3 x 2
##   Species        n
##   <fct>      <int>
## 1 setosa        11
## 2 versicolor    15
## 3 virginica      3
```

Summarizing data

```
iris %>%
  summarize(mean.sepal.width = mean(Sepal.Width))
```

```
##   mean.sepal.width
## 1         3.057333
```

Summarize by specific groups:

```
iris %>%
  group_by(Species) %>%
  summarize(mean.sepal.width = mean(Sepal.Width))
```

```
## # A tibble: 3 x 2
##   Species    mean.sepal.width
##   <fct>                 <dbl>
## 1 setosa                 3.43
## 2 versicolor             2.77
## 3 virginica              2.97
```

Data Management in R: Day 2

Course by: Laura Stein
Instructor: Ed Higgins
TAs: Sam Eliades + Megan Malish
5th August 2021

Clear R’s brain

```
rm(list=ls())
```

Call tidyverse

```
library(tidyverse)
```

Remember that we installed tidyverse last time, so we do not need to do that again. We do, however, need to call in the library at the beginning of our session.

We are now ready to get started!

We use the code *read_csv* to bring in a csv file

We will use an online, freely available dataset of the McDonald’s menu:

```
menu <- read_csv( "http://joeystanley.com/data/menu.csv" )
```

```
## Parsed with column specification:
## cols(
##   Category = col_character(),
##   Item = col_character(),
##   Oz = col_double(),
##   Calories = col_integer(),
##   Fat = col_double(),
##   Sugars = col_integer()
## )
```

Note that *read_csv*, unlike *read.csv*, automatically shows us how R reads in the data, as if we were using *str()*. It has also turned our data frame into a “tibble”, the tidyverse’s version of a table.

Let’s look at the menu as a whole dataframe

We will use base R for this:

```
menu.df <- as.data.frame(menu)
menu.df
```

I am not pasting it here because it is so big.

PLUS, it’s so difficult to parse! Let’s see how tidyverse converted it:

```
menu
```

```
## # A tibble: 260 x 6
##    Category  Item                                  Oz Calories   Fat Sugars
##    <chr>     <chr>                              <dbl>    <int> <dbl>  <int>
##  1 Breakfast Egg McMuffin                         4.8      300    13      3
##  2 Breakfast Egg White Delight                    4.8      250     8      3
##  3 Breakfast Sausage McMuffin                     3.9      370    23      2
##  4 Breakfast Sausage McMuffin with Egg            5.7      450    28      2
##  5 Breakfast Sausage McMuffin with Egg Whites     5.7      400    23      2
##  6 Breakfast Steak & Egg McMuffin                 6.5      430    23      3
##  7 Breakfast Bacon, Egg & Cheese Biscuit (Reg~    5.3      460    26      3
##  8 Breakfast Bacon, Egg & Cheese Biscuit (Lar~    5.8      520    30      4
##  9 Breakfast Bacon, Egg & Cheese Biscuit with~    5.4      410    20      3
## 10 Breakfast Bacon, Egg & Cheese Biscuit with~    5.9      470    25      4 ## #
... with 250 more rows
```

Note that this is similar to the *head()* and *str()* functions combined

Let’s refresh on some dplyr functions for selecting and reordering columns

Only show Calories and Sugars

```
menu %>%
  select(Calories, Sugars)
```

```
## # A tibble: 260 x 2
##    Calories Sugars
##       <int>  <int>
##  1      300      3
##  2      250      3
...
```

What if we want to look at everything except Item?

```
menu %>%
  select(-Item)
```

```
## # A tibble: 260 x 5
##    Category     Oz Calories   Fat Sugars
##    <chr>     <dbl>    <int> <dbl>  <int>
##  1 Breakfast   4.8      300    13      3
##  2 Breakfast   4.8      250     8      3
...
```

Note that we can ask tidyverse to eliminate more than one column

```
menu %>%
  select(-Item, -Category)
```

```
## # A tibble: 260 x 4
##       Oz Calories   Fat Sugars
##    <dbl>    <int> <dbl>  <int>
##  1   4.8      300    13      3
##  2   4.8      250     8      3
...
```

Alternatively, if the columns you want to keep are in order, use a colon:

```
menu %>%
  select(Oz:Sugars)
```

```
## # A tibble: 260 x 4
##       Oz Calories   Fat Sugars
##    <dbl>    <int> <dbl>  <int>
##  1   4.8      300    13      3
##  2   4.8      250     8      3
...
```

You can also select everything *except* for these columns easily:

```
menu %>%
  select(-(Oz:Sugars))
```

```
## # A tibble: 260 x 2
##    Category  Item
##    <chr>     <chr>
##  1 Breakfast Egg McMuffin
##  2 Breakfast Egg White Delight
...
```

You can also reorder your columns easily, just select them in the order you want

```
menu %>%
  select(Oz, Sugars, Fat, Calories)
```

```
## # A tibble: 260 x 4
##       Oz Sugars   Fat Calories
##    <dbl>  <int> <dbl>    <int>
##  1   4.8      3    13      300
##  2   4.8      3     8      250
...
```

Especially useful if you want to see something hidden near the end of your dataset!

This is how you would have to do the same thing in base R:

```
subset(menu, select=c(Oz, Sugars, Fat, Calories))
```

```
## # A tibble: 260 x 4
##       Oz Sugars   Fat Calories
##    <dbl>  <int> <dbl>    <int>
##  1   4.8      3    13      300
##  2   4.8      3     8      250
...
```

These are about equivalent for a few columns, but what if you have a lot of them? The ‘everything’ function is handy for this. Let’s say we just want the Sugars column to be in the front but *everything()* else the same

```
menu %>%
  select(Sugars, everything())
```

```
## # A tibble: 260 x 6
##    Sugars Category  Item                                Oz Calories   Fat
##     <int> <chr>     <chr>                            <dbl>    <int> <dbl>
##  1      3 Breakfast Egg McMuffin                       4.8      300    13
##  2      3 Breakfast Egg White Delight                  4.8      250     8
...
```

This is where tidyverse is much more convenient than base R!

Let’s remember the *mutate()* function.

Mutate adds a new column. What if we want to make a global change to a column (for example, round Oz up to an integer?)

```
menu %>%
  mutate(Oz_rounded = round(Oz,  0)) %>%
  select(Category:Oz, Oz_rounded, everything())
```

```
## # A tibble: 260 x 7
##    Category  Item                        Oz Oz_rounded Calories   Fat Sugars
##    <chr>     <chr>                    <dbl>      <dbl>    <int> <dbl>  <int>
##  1 Breakfast Egg McMuffin               4.8          5      300    13      3 ##
2 Breakfast Egg White Delight        4.8          5      250     8      3
...
```

Note that we have created a new column name (Oz_rounded) first. We can call this whatever we want. We are telling R that we want this new column to contain a function that rounds Oz to the nearest integer. We use the *round()* function, tell R where the variables are (Oz), and that we want zero decimal places (0). Finally, we reorder the columns so that we can easily see our new Oz_rounded column. *Mutate()* will automatically place new columns at the end of the dataset.

What if we need to change how R reads a variable?

This is very important for visualization and analysis. For example, we often write our fish families as a three-digit number, or call our trials 1, 2, 3. However, we need R to read them not as continuous variables but as factors.

For this dataset, R is reading Category as a character, but we need it as a factor. We can use *mutate()* for this.

```
menu %>%
  mutate(Category = as.factor(Category)) %>%
  print()
```

```
## # A tibble: 260 x 6
##    Category  Item                                Oz Calories   Fat Sugars
##    <fct>     <chr>                            <dbl>    <int> <dbl>  <int>
##  1 Breakfast Egg McMuffin                       4.8      300    13      3
##  2 Breakfast Egg White Delight                  4.8      250     8      3
...
```

Note that by using the same column name, we are changing the column itself and not creating a new one. If you are unsure about what you are doing, it never hurts to create a new column.

Let’s refresh renaming columns

```
menu %>%
  rename(name_of_food =  "Item",
         group =  "Category",
         serving_size =  "Oz",
         Sugar =  "Sugars") %>%
  print()
```

```
## # A tibble: 260 x 6
##    group     name_of_food                serving_size Calories   Fat Sugar
##    <chr>     <chr>                              <dbl>    <int> <dbl> <int>
##  1 Breakfast Egg McMuffin                         4.8      300    13     3
##  2 Breakfast Egg White Delight                    4.8      250     8     3
...
```

The new name of the column is to the left and is not in quotation marks.

This is how we would have to do the same thing in base R:

```
menu$name_of_food <- menu$Item
menu$group <- menu$Category
menu$serving_size <- menu$Oz
menu$Sugar <- menu$Sugars
menu <- menu[c( 8,7,9,4,5,10)]
menu
```

```
## # A tibble: 260 x 6
##    group     name_of_food                serving_size Calories   Fat Sugar
##    <chr>     <chr>                              <dbl>    <int> <dbl> <int>
##  1 Breakfast Egg McMuffin                         4.8      300    13     3
##  2 Breakfast Egg White Delight                    4.8      250     8     3
...
```

That is so much more annoying!

We may want to relevel our data, especially if we measured multiple things in different contexts

We use a package called **forcats** (for categories) to relevel

```
library(forcats)
```

Let’s see what our Categories look like

```
levels(menu$Category)
```

```
## NULL
```

NOTE: I actually can’t figure out why this won’t show the levels. Don’t worry, it will work from here on out!

Combining groups

Let’s say that we want Beef & Pork to be regrouped into “Red Meat”. We use the function *fct_recode()*

```
menu_temp <- menu %>%
  mutate(Category = fct_recode(Category,  "Red Meat" = "Beef & Pork" ))
levels(menu_temp$Category)
```

```
## [1] "Red Meat"           "Beverages"          "Breakfast"
## [4] "Chicken & Fish"     "Coffee & Tea"       "Desserts"
## [7] "Salads"             "Smoothies & Shakes" "Snacks & Sides"
```

First, we told R that we want to save these changes to the Category column instead of creating a new one. We then tell *fct_recode()* that we want to change levels in the column Category, that we want the new category to be called “Red Meat”, and that this will replace “Beef & Pork”.

For this example, let’s recode meats into multiple categories. We can do this all at once:

```
menu_temp <- menu %>%
  mutate(Category = fct_recode(Category,
                               "Red Meat"  = "Beef & Pork" ,
                               "White Meat"  = "Chicken & Fish" ))
levels(menu_temp$Category)
```

```
## [1] "Red Meat"           "Beverages"          "Breakfast"
## [4] "White Meat"         "Coffee & Tea"       "Desserts"
## [7] "Salads"             "Smoothies & Shakes" "Snacks & Sides"
```

Let’s say we want to put multiple things into one category (for example, if we want to combine “Predator Moving” and “Predator Not Moving” into a category called “Predator Present”).

We can make everything more simple by creating umbrella categories:

```
menu_temp <- menu %>%
  mutate(Category = fct_recode(Category,
                               "Meats" = "Beef & Pork" ,
                               "Meats" = "Chicken & Fish" ,
                               "Beverages"  = "Coffee & Tea" ,
                               "Sweets" = "Desserts" ,
                               "Sweets" = "Smoothies & Shakes" ))
levels(menu_temp$Category)
```

```
## [1] "Meats"          "Beverages"      "Breakfast"      "Sweets"
## [5] "Salads"         "Snacks & Sides"
```

Briefly, this is what the same thing would look like in base R:

First, we turn Category into a factor and allow for the new categories

```
menu$Category <- factor(menu$Category,
                        levels=c(unique(menu$Category),   "Meats", "Sweets" ))
```

Then Change the values, saving them to objects

```
menu[menu$Category %in% c("Beef & Pork", "Chicken & Fish"),]$Category <- "Meats" menu[menu$Category
== "Coffee & Tea",]$Category <- "Beverages"
menu[menu$Category %in% c("Desserts", "Smoothies & Shakes"),]$Category <- "Sweets"
```

And finally, we remove the old levels

```
menu_baseR <- droplevels(menu)
levels(menu_baseR$Category)
```

```
## [1] "Breakfast"      "Salads"         "Snacks & Sides" "Beverages"
## [5] "Meats"          "Sweets"
```

Boo! Why do that, when we can do the following?

We can make this even easier with concatenation and *fct_collapse()*

```
menu_temp <- menu %>%
  mutate(Category = fct_collapse(Category,
                                 "Meats" = c("Beef & Pork" , "Chicken & Fish" ),
                                 "Beverages"  = "Coffee & Tea" ,
                                 "Sweets" = c("Desserts" , "Smoothies & Shakes" )))
levels(menu_temp$Category)
```

```
## [1] "Breakfast"      "Meats"          "Salads"         "Snacks & Sides"
## [5] "Sweets"         "Beverages"
```

Now we can filter based on these new levels

```
menu_temp %>%
  filter(Category == "Meats")
```

```
## # A tibble: 42 x 6
##    Category Item                                Oz Calories   Fat Sugars
##    <fct>    <chr>                            <dbl>    <int> <dbl>  <int>
##  1 Meats    Big Mac                            7.4      530    27      9
##  2 Meats    Quarter Pounder with Cheese        7.1      520    26     10
...
```

We want to know how many meat and breakfast items are equal or over 11 Oz

```
menu_temp %>%
  filter(Category %in% c("Meats", "Breakfast"),
         Oz >= 11)
```

```
## # A tibble: 9 x 6
##   Category  Item                                Oz Calories   Fat Sugars
##   <fct>     <chr>                            <dbl>    <int> <dbl>  <int>
## 1 Breakfast Big Breakfast with Hotcakes (Regu~  14.8     1090    56     17
## 2 Breakfast Big Breakfast with Hotcakes (Larg~  15.3     1150    60     17
...
```

We are ready to reshape our data

Let’s use a smaller dataset

Here is a dataset in wide format:

```
vowels_wide <- read_csv("http://joeystanley.com/data/vowels_wide.csv") %>%
  print()
```

```
## Parsed with column specification:
## cols(
##   word = col_character(),
##   t_0.2 = col_double(),
##   t_0.35 = col_double(),
##   t_0.5 = col_double(),
##   t_0.65 = col_double(),
##   t_0.8 = col_double()
## )
```

```
## # A tibble: 3 x 6
##   word  t_0.2 t_0.35 t_0.5 t_0.65 t_0.8
##   <chr> <dbl>  <dbl> <dbl>  <dbl> <dbl>
## 1 fall   516.   480.  492.   459.  480.
## 2 feel   320.   318.  355.   483.  514.
## 3 fool   297.   304.  299.   299.  296.
```

Here is the same dataset in tall format:

```
vowels_tall <- read_csv( "http://joeystanley.com/data/vowels_tall.csv" ) %>%
  print()
```

```
## Parsed with column specification:
## cols(
##   word = col_character(),
##   time = col_double(),
##   Hz = col_double()
## )
```

```
## # A tibble: 15 x 3
##    word   time    Hz
##    <chr> <dbl> <dbl>
##  1 fall   0.2   516.
##  2 fall   0.35  480.
##  3 fall   0.5   492.
##  4 fall   0.65  459.
##  5 fall   0.8   480.
##  6 feel   0.2   320.
##  7 feel   0.35  318.
##  8 feel   0.5   355.
##  9 feel   0.65  483.
## 10 feel   0.8   514.
## 11 fool   0.2   297.
## 12 fool   0.35  304.
## 13 fool   0.5   299.
## 14 fool   0.65  299.
## 15 fool   0.8   296.
```

Let’s go from wide to tall using *gather()*

The way *gather()* works is you need three arguments. First, is the key, which is some arbitrary name that will be given to the new column we’re going to create. Next is the value argument, which is the name of the column you’re creating that contains the information. Finally, you just name the columns that you want to be combined into one.

Here, we want to combine all columns except for ‘word’.

```
vowels_wide %>%
  gather(key = "time", value = "Hz", -word)
```

```
## # A tibble: 15 x 3
##    word  time      Hz
##    <chr> <chr>  <dbl>
##  1 fall  t_0.2   516.
##  2 feel  t_0.2   320.
##  3 fool  t_0.2   297.
##  4 fall  t_0.35  480.
##  5 feel  t_0.35  318.
##  6 fool  t_0.35  304.
##  7 fall  t_0.5   492.
##  8 feel  t_0.5   355.
##  9 fool  t_0.5   299.
## 10 fall  t_0.65  459.
## 11 feel  t_0.65  483.
## 12 fool  t_0.65  299.
## 13 fall  t_0.8   480.
## 14 feel  t_0.8   514.
## 15 fool  t_0.8   296.
```

Hmm, we have a weird t_ in our time category. How do we get rid of it?

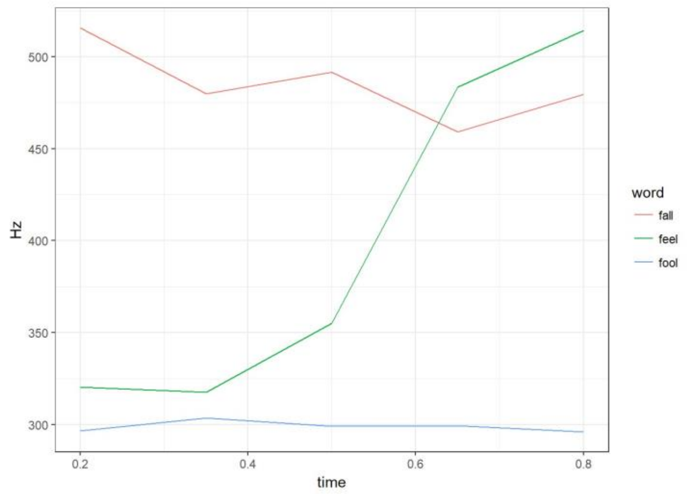
```
vowels_wide %>%
  gather(key = "time", value = "Hz", -word) %>%
  separate(time, into = c( "throw_away", "time"), sep="_") %>%
  select(-throw_away)
```

```
## # A tibble: 15 x 3
##    word  time     Hz
##    <chr> <chr> <dbl>
##  1 fall  0.2    516.
##  2 feel  0.2    320.
##  3 fool  0.2    297.
##  4 fall  0.35   480.
##  5 feel  0.35   318.
##  6 fool  0.35   304.
##  7 fall  0.5    492.
##  8 feel  0.5    355.
##  9 fool  0.5    299.
## 10 fall  0.65   459.
## 11 feel  0.65   483.
## 12 fool  0.65   299.
## 13 fall  0.8    480.
## 14 feel  0.8    514.
```

Beautiful! We have used the function *separate()*. We tell *separate()* which column we want to, well, separate apart, which here is ‘time’. We then use the ‘into’ function to tell *separate()* which two new columns we want them separated into. We have named them “throw_away” and “time”. Finally, we tell *separate()* that we want the values in the column to be split at the underscore (_). Finally, we use *select()* to look at all the columns except the “throw_away” column, which will only have “t_” in it.

Tall format allows you to make lines in ggplot. Here is an example (we will do ggplot in more detail later)

```
vowels_tall %>%
  ggplot(aes(x=time, y=Hz, group=word, color=word)) +
  geom_line() +
  theme_bw()
```



What about going from tall to wide? We use *spread()*

With *spread()*, the syntax is similar to *gather()*. The key argument is where you specify which column contains the values that will become the colum names. In this case it's the time column since we want to spread each point in time into its own column. The value argument contains the column in the wide format that contains what you want to fill those cells. In this case, it's the Hz column. We tell spread to distinguish the columns using an underscore with sep.

```
vowels_tall %>%
  spread(key = time, value = Hz, sep =  "_") %>%
  print()
```

```
## # A tibble: 3 x 6
##   word  time_0.2 time_0.35 time_0.5 time_0.65 time_0.8
##   <chr>    <dbl>     <dbl>    <dbl>     <dbl>    <dbl>
## 1 fall      516.      480.     492.      459.     480.
## 2 feel      320.      318.     355.      483.     514.
## 3 fool      297.      304.     299.      299.     296.
```

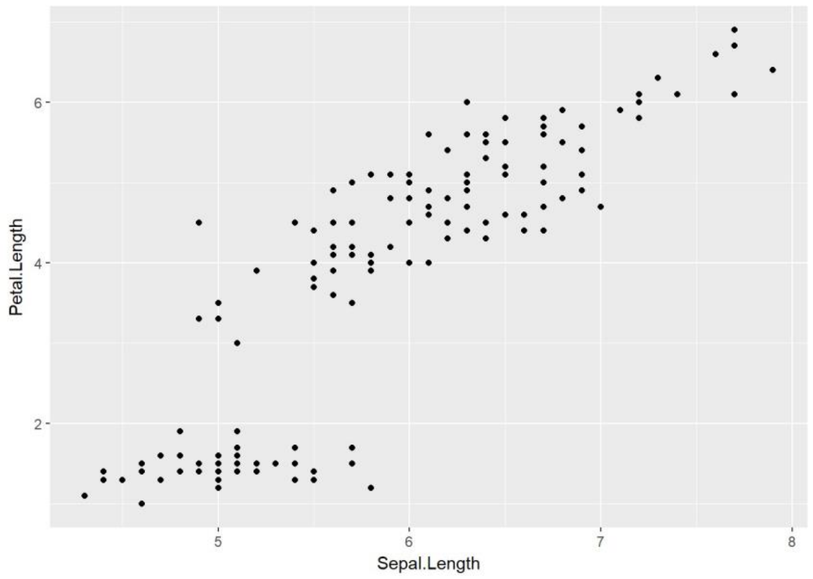
Now we are ready to play with ggplot2

Note that ggplot2 works with data frames, not vectors like base R plots

Let's just make a quick scatterplot using the iris dataset

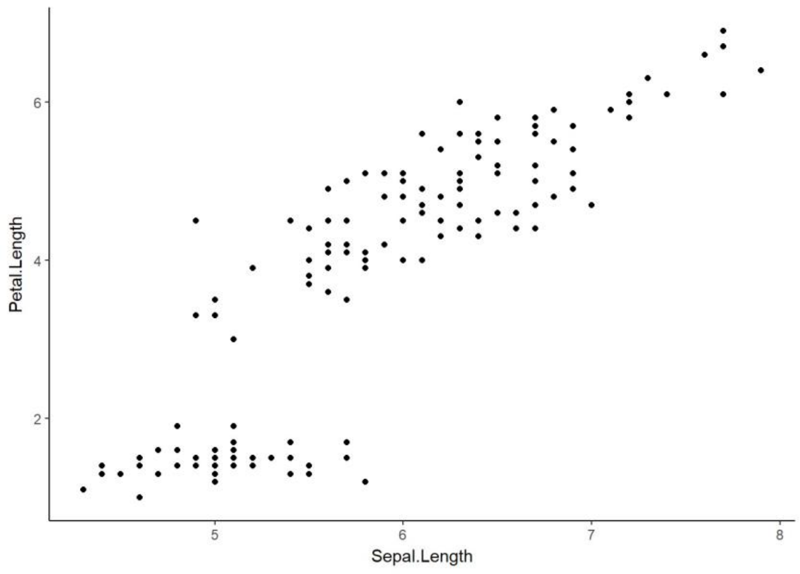
ggplot() functions much like the tidyverse pipeline, only instead of %>%, we use + There are three major functions in *ggplot()*: **ggplot**, or "*g*rammar of *g*raphics". This is where you create your base plot. **geom**. Geom tells ggplot what type of graph (or the *geom*etry) of your plot will be. **aes**. Aes describes the *aes*thetics of your plot.

```
ggplot(iris, aes(x = Sepal.Length, y =
```



Gross, I hate that grey background. Let's get rid of it:

```
ggplot(iris, aes(x = Sepal.Length, y = Petal.Length)) +
  geom_point() +
  theme_classic()
```

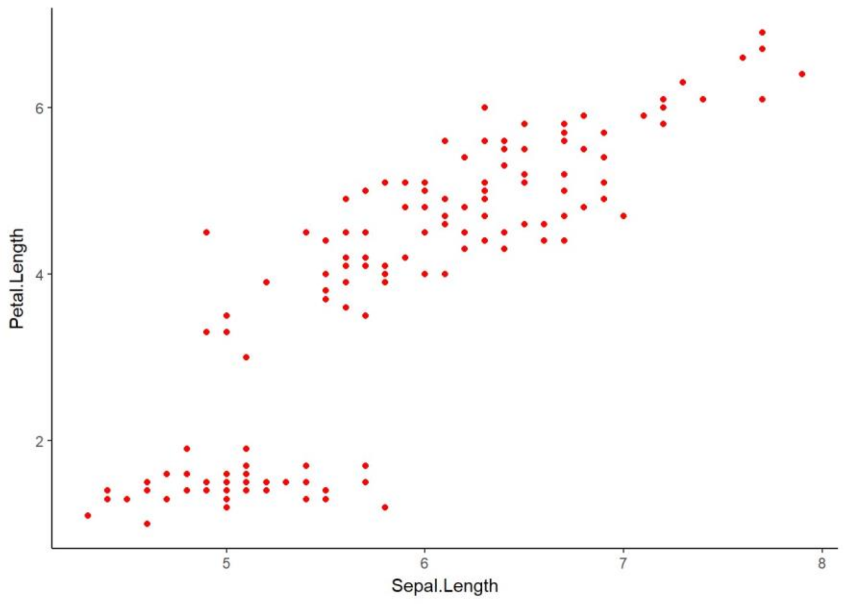


Note there are lots of “themes” and they can be customized.

What we have done here is called *ggplot()* to make a plot using dataset iris. We then tell it that for the aesthetics of the base plot, we want Sepal.Length on the x-axis and Petal.Length on the y-axis. Finally, we tell it that we want it to be a scatterplot (or create *points*) using geom_*point*.

Changing the colors

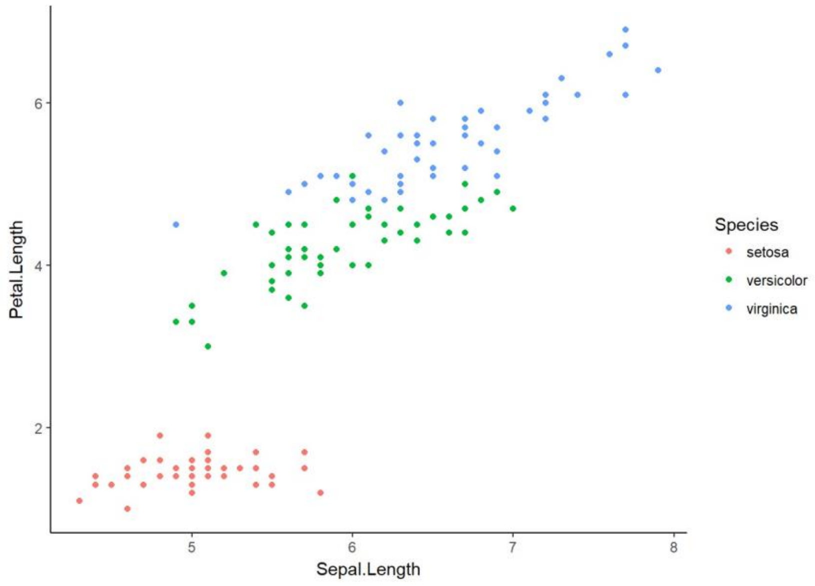
```
ggplot(iris, aes(x = Sepal.Length, y = Petal.Length)) +
  geom_point(color =  "red" ) +
  theme_classic()
```



This is easy, we tell R that we want the geometric points to be red.

We want color based on species

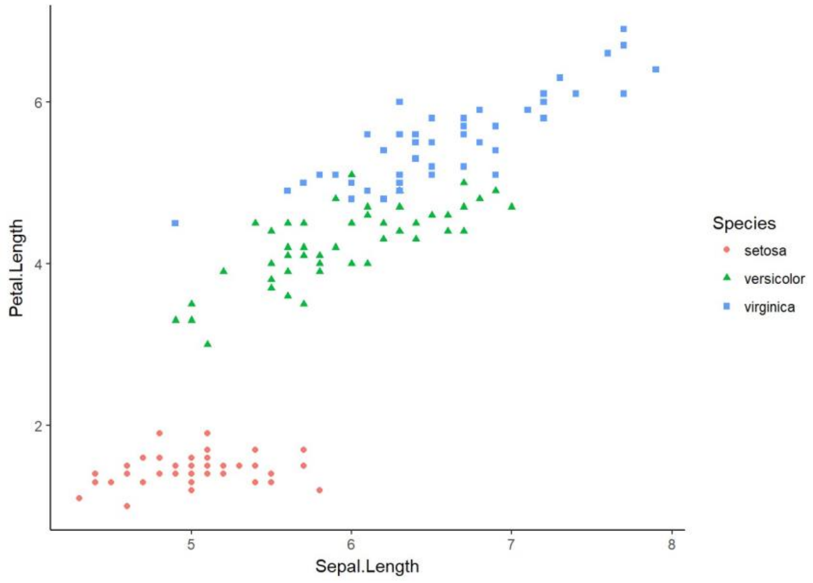
```
ggplot(iris, aes(x = Sepal.Length, y = Petal.Length, color = Species)) +
  geom_point() +
  theme_classic()
```



We tell R under the aesthetics that we want color to depend on Species. Note that now we don't have color specified in *geom_point()*.

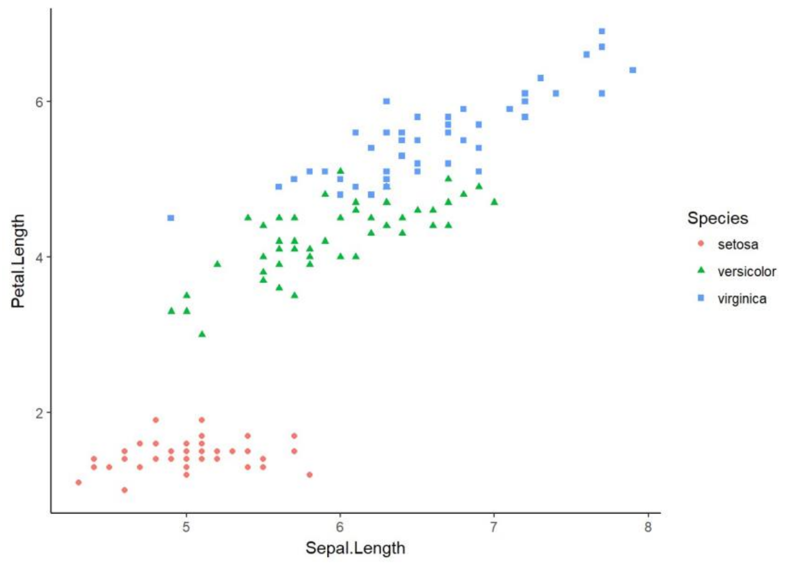
We also want the points to be different shapes based on species

```
ggplot(iris, aes(x = Sepal.Length, y = Petal.Length, color = Species, shape = Species)) +
  geom_point() +
  theme_classic()
```



Again, we just add "shape=" to our general aesthetics. But there are multiple ways to do this. We can also place color and shape into *geom_point()*

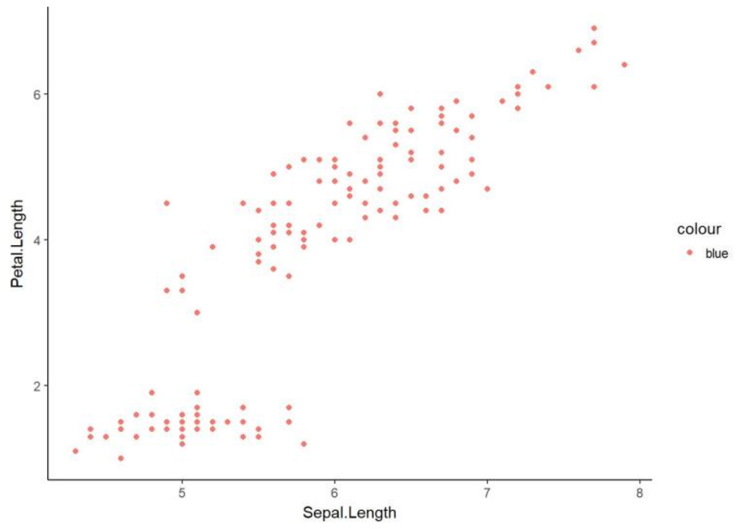
```
ggplot(iris, aes(x = Sepal.Length, y = Petal.Length)) +
  geom_point(aes(color = Species, shape = Species)) +
  theme_classic()
```



Note aesthetics are for **mapping only**

What if we want to color all points blue?

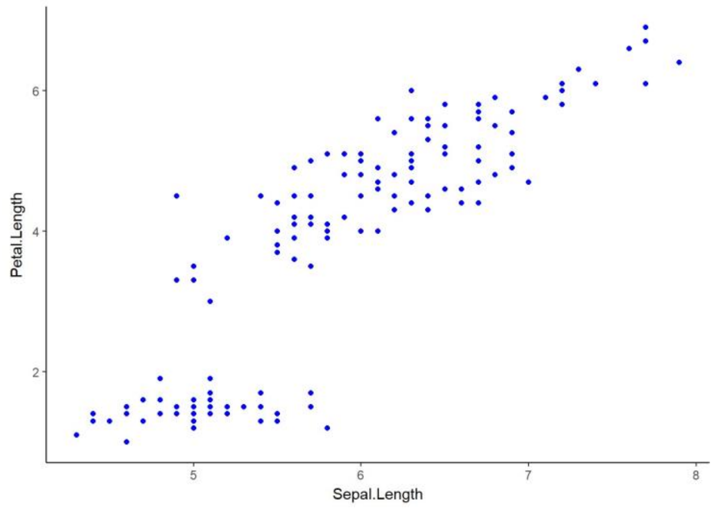
```
ggplot(iris, aes(x = Sepal.Length, y = Petal.Length, color =  "blue")) +
  geom_point() +
  theme_classic()
```



Oops! Because this is under an aesthetics tag, it is trying to group all points by a category called “blue”.

Correctly color all points blue:

```
ggplot(iris, aes(x = Sepal.Length, y = Petal.Length)) +
  geom_point(color =  "blue") +
  theme_classic()
```



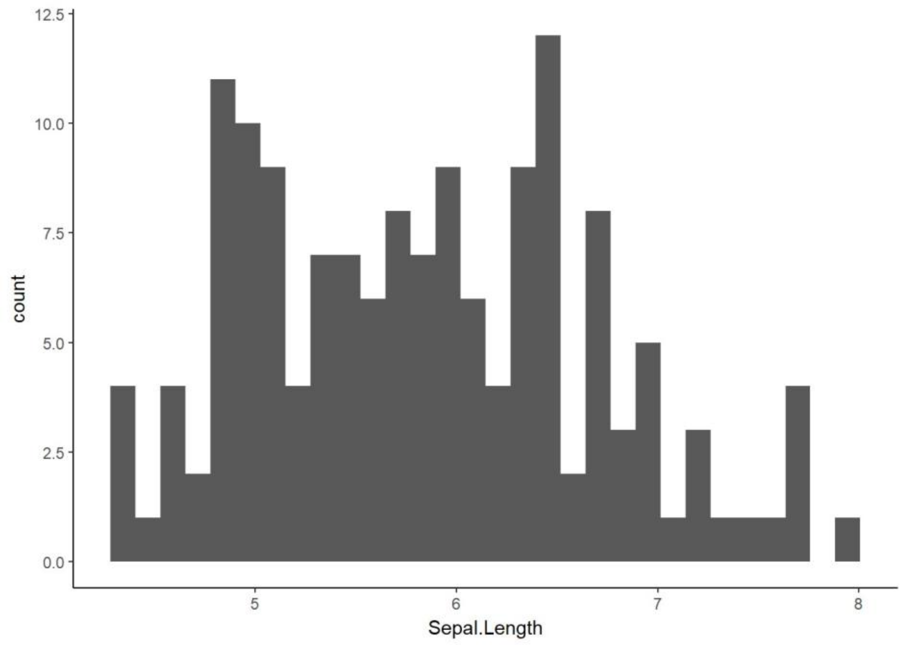
Remember histograms? Let’s make some!

We will go back to the ‘iris’ dataset

Making a histogram is simple. We simply change our geom:

```
ggplot(iris, aes(x = Sepal.Length)) +
  geom_histogram() +
  theme_classic()
```

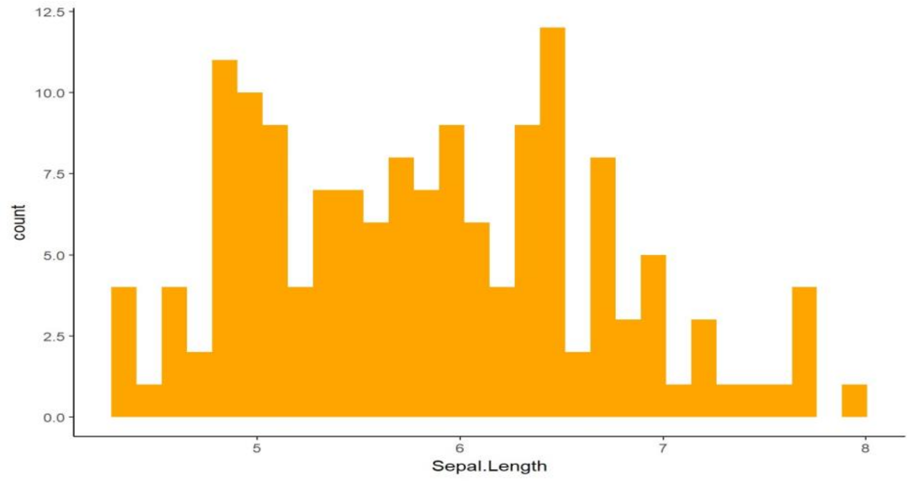
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Let’s color it in

```
ggplot(iris, aes(x = Sepal.Length)) +
  geom_histogram( fill =  "orange" ) +
  theme_classic()
```

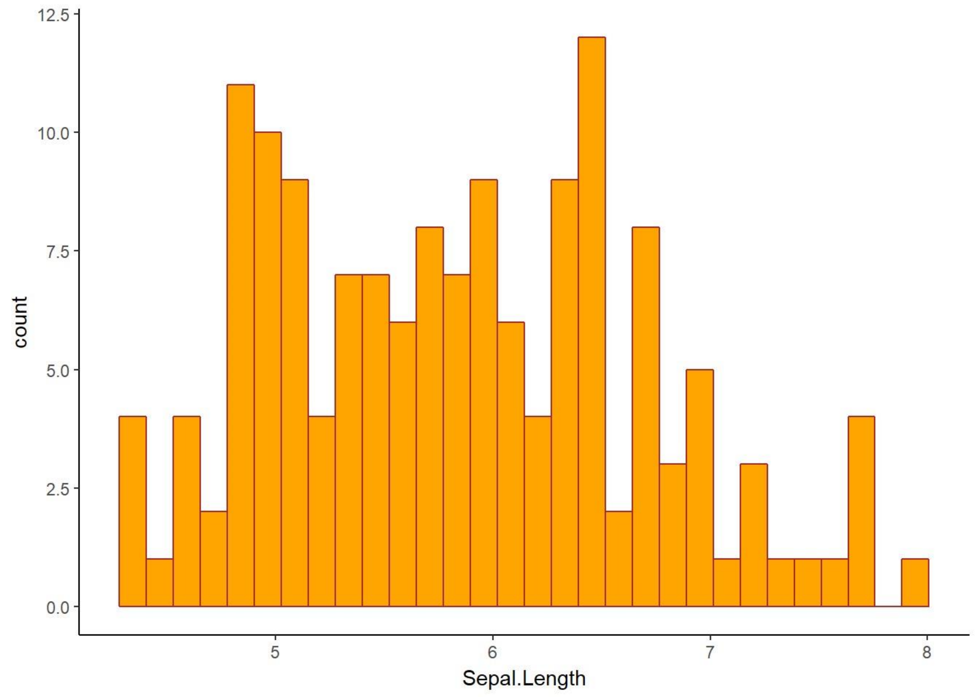
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



It's hard to distinguish our bars. Let's give them an outline.

```
ggplot(iris, aes(x = Sepal.Length)) +
  geom_histogram( fill =  "orange", color =  "brown"  ) +
  theme_classic()
```

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

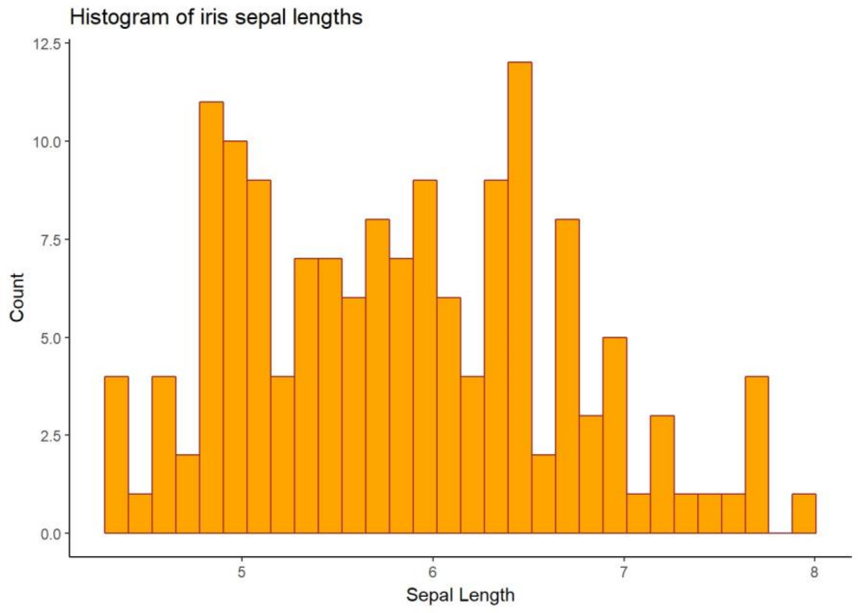


Note that for most ggplot geoms, *fill* will determine the inside color, while *col* determines the outline color.

What are we looking at? Let's add labels.

```
ggplot(iris, aes(x = Sepal.Length)) +
  geom_histogram( fill =  "orange", color = "brown"  ) +
  xlab("Sepal Length" ) +
  ylab("Count") +
  ggtitle("Histogram of iris sepal lengths" ) +
  theme_classic()
```

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

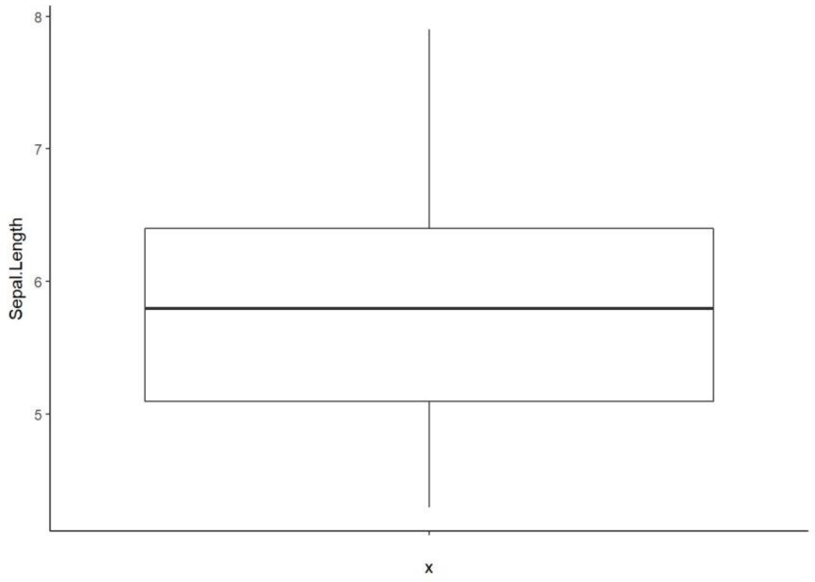


Again, this is easy. We just add *xlab*, *ylab*, and *ggtitle*. These are further customizable with size, color, placement, etc.

Remember boxplots? Let’s make boxplots!

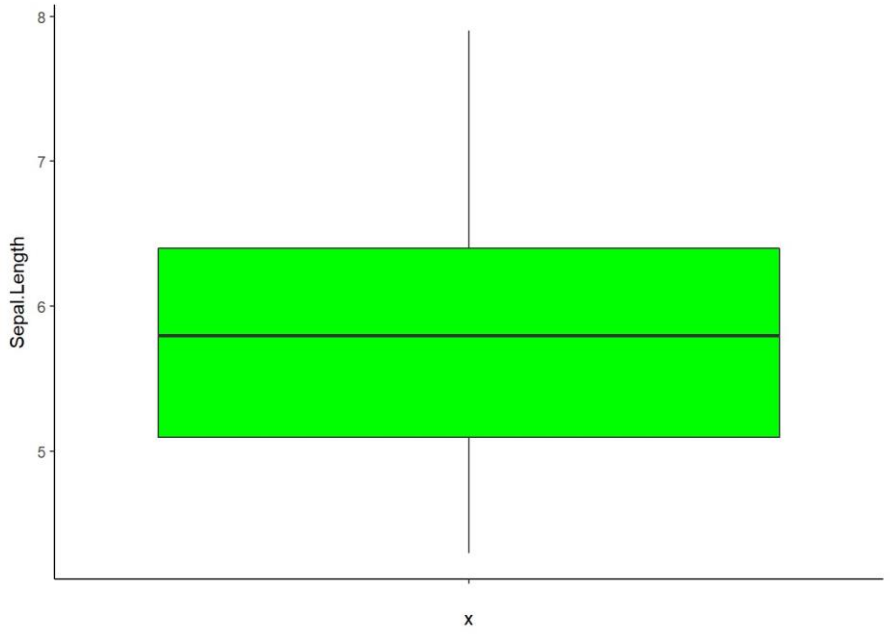
Just one box showing Sepal.Length (leaving *x* blank). As you might expect, we just change our geom:

```
ggplot(iris, aes(x =  "", y = Sepal.Length)) +
  geom_boxplot() +
  theme_classic()
```



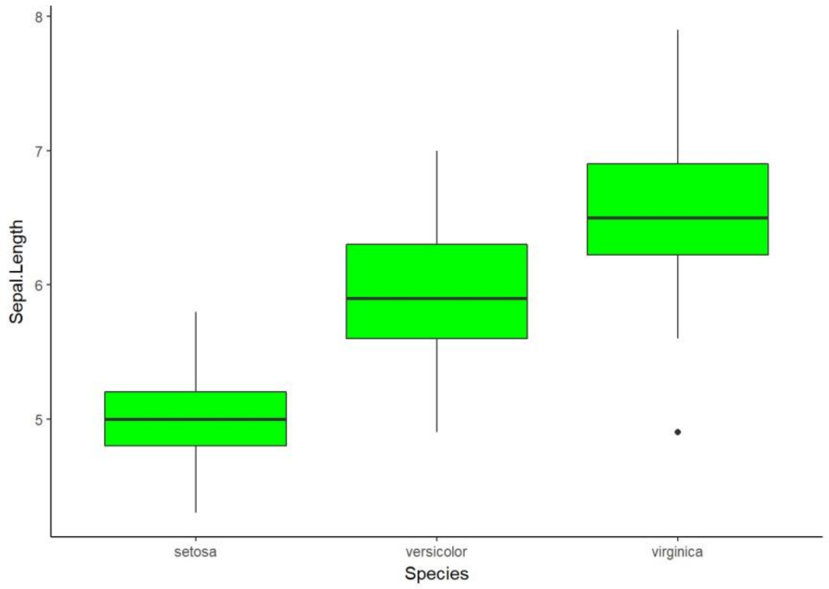
Let's color it in:

```
ggplot(iris, aes(x =  "", y = Sepal.Length)) +
  geom_boxplot(fill =  "green") +
  theme_classic()
```



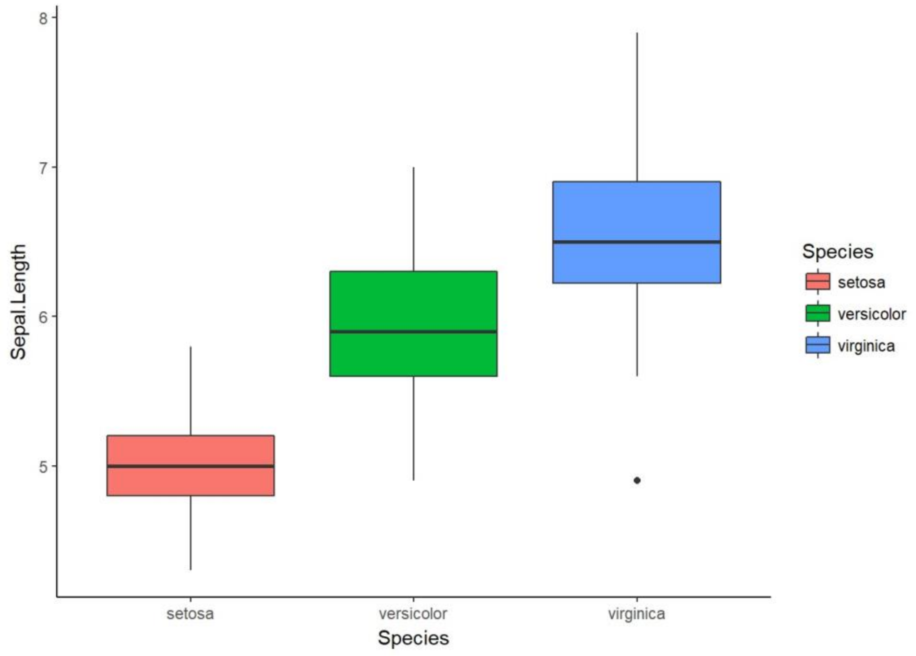
What if we want boxes by species? We add it into our *x*

```
ggplot(iris, aes(x = Species, y = Sepal.Length)) +
  geom_boxplot(fill =  "green") +
  theme_classic()
```



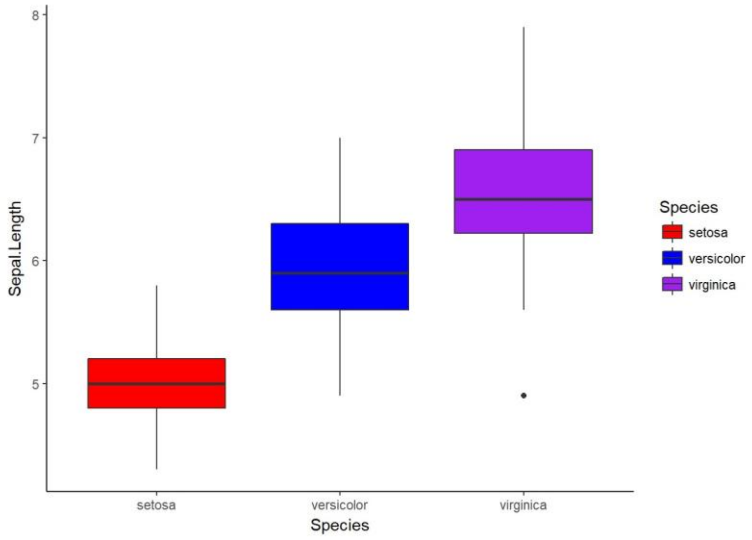
How do we get different colors for species? The same way we did for the scatterplot. Remember, however, that we want the boxes *fill*ed in:

```
ggplot(iris, aes(x = Species, y = Sepal.Length, fill = Species)) +
  geom_boxplot() +
  theme_classic()
```



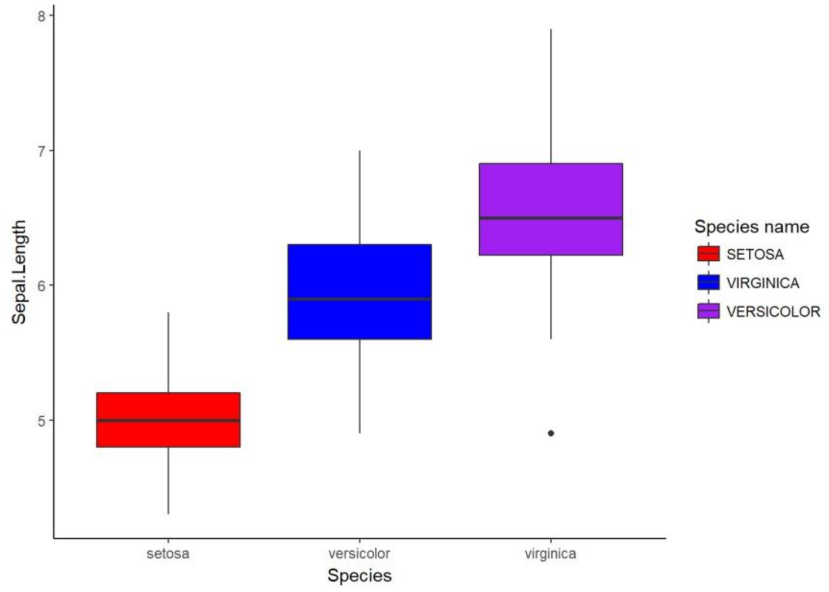
I want custom colors! I can do this using a function called *scale_fill_manual*:

```
ggplot(iris, aes(x = Species, y = Sepal.Length, fill = Species)) +
  geom_boxplot() +
  scale_fill_manual(values=c( "red", "blue", "purple" )) +
  theme_classic()
```



scale_fill_manual can also adjust the legend. *name* defines the legend title, while the *labels* function changes how they are presented.

```
ggplot(iris, aes(x = Species, y = Sepal.Length, fill = Species)) +
  geom_boxplot() +
  scale_fill_manual(values=c( "red", "blue", "purple" ), name = "Species name" , labels=c( "SETOSA",
"VIRGINICA" , "VERSICOLOR" )) +
  theme_classic()
```



Changing the order of factors

The order of factors is automatically done by alphabetical order. Changing the order can be key, especially if you have things like “before” and “after”.

Ordering depends on factor levels.

```
levels(iris$Species)
```

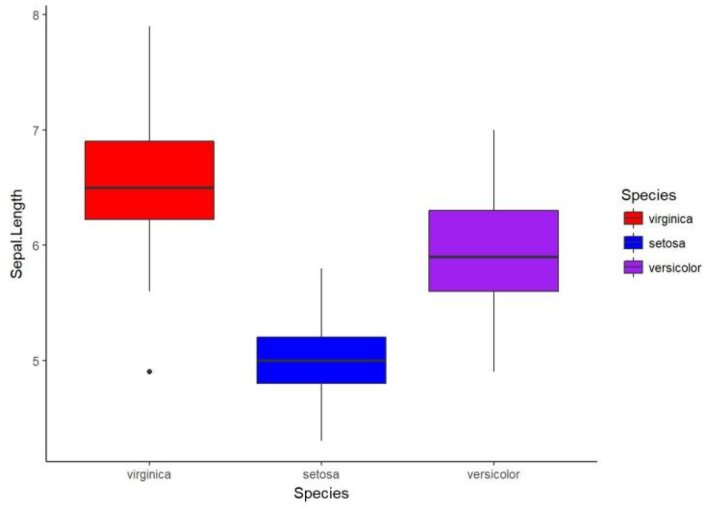
```
## [1] "setosa"     "versicolor" "virginica"
```

We can change order of levels like so:

```
iris$Species <- factor(iris$Species, levels=c( "virginica", "setosa", "versicolor"))
```

Replot:

```
ggplot(iris, aes(x = Species, y = Sepal.Length, fill = Species)) +
  geom_boxplot() +
  scale_fill_manual(values=c( "red", "blue", "purple")) +
  theme_classic()
```



Let’s make grouped boxplots!

We will start by creating another categorical variable for grouping purposes. First, we group the dataset by Species. We will use *mutate()* to create a new column called “size”. If Sepal.Width for any flower is greater than the median Sepal.Width of the species, it will be categorized as “big”. Otherwise, it will be categorized as “small”.

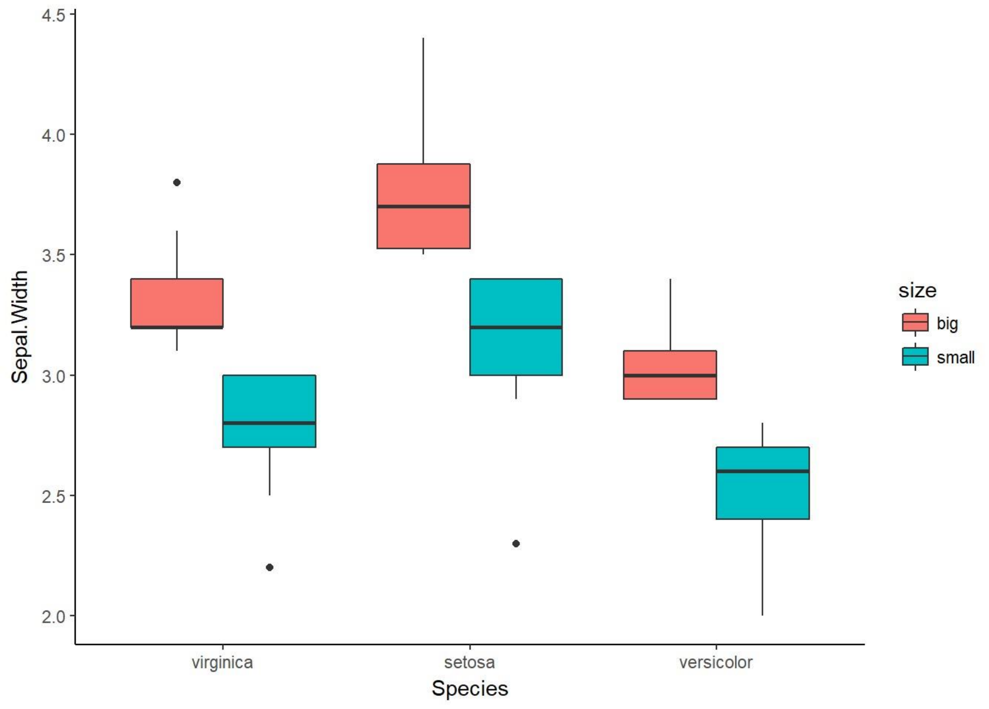
```
iris2 <- iris %>%
  group_by(Species) %>%
  mutate(size = ifelse( Sepal.Width > median(Sepal.Width) ,  "big" , "small" ))

head(iris2)
```

```
## # A tibble: 6 x 6
## # Groups:   Species [1]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species size
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>   <chr>
## 1          5.1         3.5          1.4         0.2 setosa  big
## 2          4.9         3            1.4         0.2 setosa  small
## 3          4.7         3.2          1.3         0.2 setosa  small
## 4          4.6         3.1          1.5         0.2 setosa  small
## 5          5           3.6          1.4         0.2 setosa  big
## 6          5.4         3.9          1.7         0.4 setosa  big
```

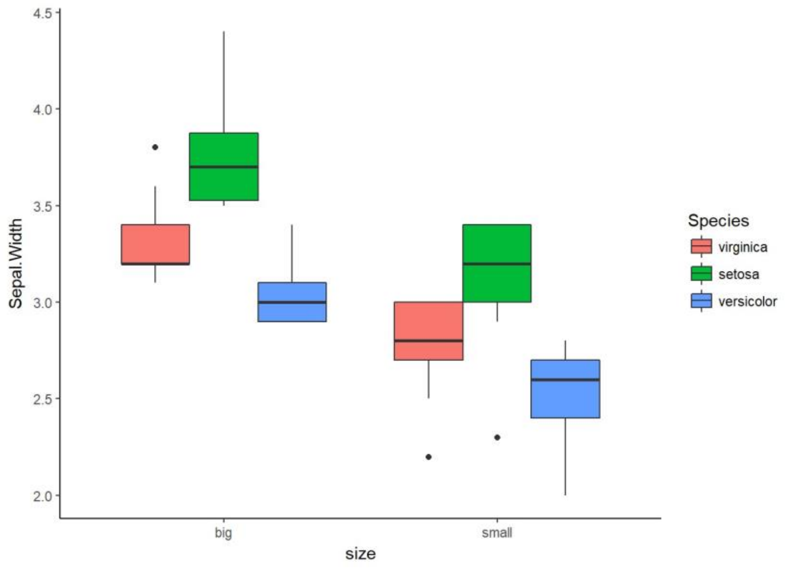
Now let’s graph based on size. We add size to the *fill* section in the main plot.

```
ggplot(iris2, aes( x = Species, fill=size, y=Sepal.Width)) +
  geom_boxplot() +
  theme_classic()
```



We can also graph it the other way. Depending on our hypotheses and analyses, we will choose the best way to represent our data.

```
ggplot(iris2, aes( x = size, fill = Species, y=Sepal.Width)) +
  geom_boxplot() +
  theme_classic()
```



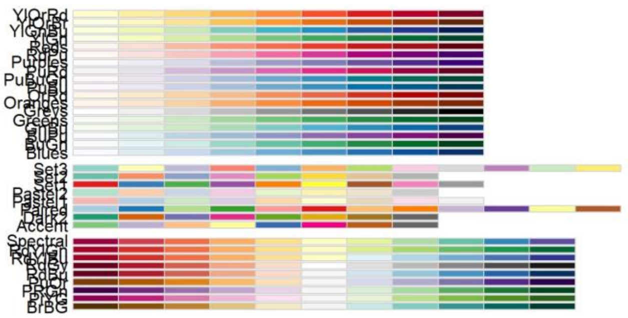
Colors! *scale__brewer()* uses pre-selected color gradients

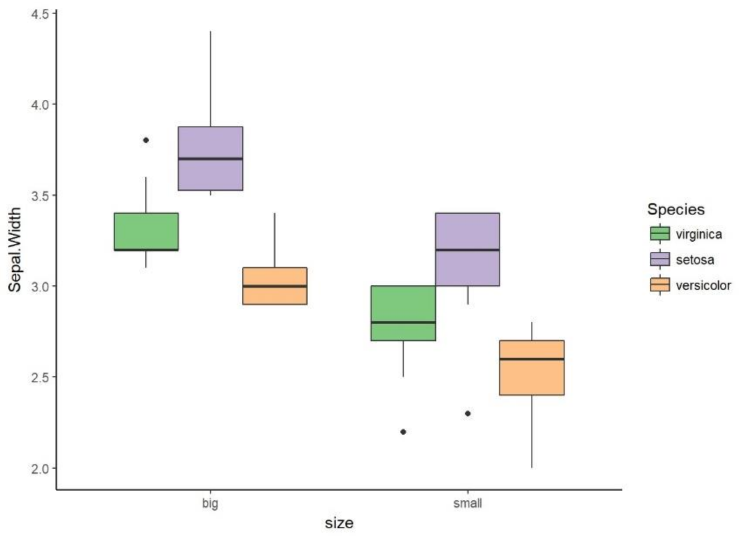
First, install and call **RColorBrewer**

```
install.packages( "RColorBrewer" )
library(RColorBrewer )
```

You can see the colors available to you using the following code:

```
display.brewer.all()
```





You can use any of the preset gradients to color your graphs by adding *palette=""* under *scale_fill_brewer()*. Here I am using "Accent".

```
ggplot(iris2, aes( x = size, fill = Species, y=Sepal.Width)) +
  geom_boxplot() +
  scale_fill_brewer(palette= "Accent") +
  theme_classic()
```