Statistics in R Style Guide

Stats TAs

September 16, 2020

In order to create a helpful resource for learning R, we will update this document frequently over the course of the semester to include the most recent and relevant information.

Week 0

Preliminaries/Fundamentals

All text (ie your write-up) in an Rmd document goes in this space – the text space. By contrast, all of your code to determine the results goes in a code chunk, such as the code chunk directly below this line:

```
# put code here; e.g.
a <- mean(iris$Petal.Length)
```

HTML Title

When you start your homework as a .Rmd file, you should change the `title` and `author` fields of the document to reflect it being your own work in the same format given above.

```
## Good .Rmd header
---
title: "HW1_kfurlong"
author: "Kyle Furlong"
date: "August 28, 2018"
output: html_document
---

## Bad .Rmd header
---
title: "Homework 1"
author: "unknown author"
date: "August 28, 2018"
output: html_document
---
```

Loading libraries

Make sure to load any libraries in your Rmd document before entering any commands from that library. Since we frequently use `tidyverse` commands in this class, we recommend always loading this at the beginning.

```
library(tidyverse)
```

Writing code to the command line

There is almost never a good reason to do this. Keep all of your code within the Rmd or R script as a good practice; delete the code you do not need at the end.

Getting Help in R

For many questions involving specific functions or packages in R, you can get help using the Help tab in the bottom right pane of R Studio. You can also do this by typing `?functionName` in the Console of R Studio.

```
# Good Examples
?summary
?mean
```

Running a Single Line of Code

In some cases, you may not want to run the entire code chunk for a problem. To only run a single line of code, select the line of code you want to run and hit `CTRL + Enter` on Windows or `Command + Enter` on Macs

Naming conventions

When naming functions and variables, it is important that these objects are both concise and meaningful. Generally, one should avoid using hyphens and underscores in function and variable names. To separate individual words in a variable, use a dot (period).

```
# Good Examples
countDogs <- function(vector){
  # function content here
}
n.dogs <- sum(x)
numberDogs <- sum(x)

# Bad Examples
count_Number_of_Dogs_in_Dataframe <- function(vector){
  # function content here
}
N-Dogs <- sum(x)
```

Furthermore, it is crucial that your object names are unique and not the name of existing functions or variables. Above all, one should strive to be as consistent as possible in one's naming conventions and general coding.

```
# Bad Examples
FALSE <- TRUE
TRUE <- sum(x)
c <- is.na(x)
mean <- abs(-5)
```

Syntax

Line Length

Try to keep the length of each line of code under 80 characters. As a general rule of thumb, you should not neeed to scroll horizontally to read a single line of code.

Assignment Operator

When creating a new object, use the <- operator and *not* the = operator.

```r
# Good Example
newVariable <- 5 + 2

# Bad Example
bad.variable = 3 + 17
```

Spacing

Place spaces around all binary operators (=, +, -, <-, etc.). Do not place a space before a comma, but always place one after a comma.

```r
# Good Examples
small.cars.df <- cars[cars$speed > 5, "dist"]
simpleCalculation <- (3 * 2) + 17 - (4 / 3)

# Bad Examples
small.cars.df<-cars[cars$speed>5,"dist"]
simpleCalculation<-(3*2)+17-(4/3)
```

The only exception to the above rule involves the colon operator `:` or `::`. In these cases, do not put spaces around the operator.

```r
# Good
x <- 1:10
purrr::map

# Bad
x <- 1 : 10
purrr :: map
```

Miscellaneous

Printing an Object

It's possible to print an object many ways in R. We recommend you do so by simply writing the name of the object.

```r
# Good Example
sum.8 <- 4 + 4
sum.8
```

Defining Arguments in Functions

For some functions, particularly longer and more complex ones, it may be helpful to explicitly define the arguments. This will help you and the reader keep track of what each part of the syntax means. A general rule of thumb is if you need to look up the arguments in the Help files of a function, you should define the arguments when writing your code.

```r
# Good Examples
seq(from = 1950, to = 2010, by = 10)
round(x = pi, digits = 5)
```

Saving Objects and the Workspace

When you attempt to close R, you will be prompted to save your workspace in many cases. Doing so will save the data and variables listed in your `Environment` and enable you continue working with them when you open R again. However, this is largely not necessary in this course and generally you should not do it.

It's possible to save your images and data files in R. For more information on this, see section 1.3.6 of your textbook.

Calling a function from a package

Sometimes you may wish to use a function from a package without loading the package directly. Othertimes, R may be tempermental about having multiple functions loaded in the environment that have the same name, and will default to the function you *don't* want. In either of these cases you may wish to call the namespace of the function directly using the `::` operator in combination with the package name.

```r
mtcars %>%
  dplyr::mutate(var = cyl + mpg) %>%
  head()
```

```
##    mpg cyl disp  hp drat    wt  qsec vs am gear carb  var
## 1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4 27.0
## 2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4 27.0
## 3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1 26.8
## 4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1 27.4
## 5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2 26.7
## 6 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1 24.1
```

Accessing columns and rows from a dataframe

There are many ways to access or return a variable from a data frame. When you are working with a single column in the data frame, we recommend using the '$' operator. When you are working with mutliple columns we recommend using square brackets.

Columns

```r
# Good Single Variable Examples: Tidyverse
cars.distance.df <- cars %>%
  select(dist)

# Good Single Variable Examples: Base R
cars.distance <- cars$dist

# Good Multi-Variable Examples: Tidyverse
cars.distance.and.speed.df <- cars %>%
  select(dist, speed)

# Good Multi-variable Examples: Base R
cars.distance.and.speed <- cars[, c("dist", "speed")]
```

Rows

Somtimes we may only want particular row indeces of a dataframe. Note that this happens farily infrequently, and more often we wish to have well-defined subsets of a variable instead (see below).

```r
# Good example of taking rows 1-5: Tidyverse
cars.distance.rows <- cars %>%
  slice(1:5)

# Good Single Variable Examples: Base R
cars.distance.rows <- cars[c(1:5), ]
```

We can also combine these with the column operations:

```
# Good example of taking rows 1-5: Tidyverse
cars.distance.df <- mtcars %>%
  select(mpg, cyl)
  slice(1:5)

# Good Single Variable Examples: Base R
cars.distance <- mtcars[c(1:5), c('mpg', 'cyl')]
```

Week One

Summarizing the distribution of a variable

The `summary` command returns the basic quantiles of the distribution of a continuous variable (min, 25th percentile, median, 75th percentile, and maximum) as well as the mean.

```
summary(iris$Sepal.Length)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.300   5.100   5.800   5.843   6.400   7.900
```

Consider the case now where we want to summarize a variable by the values of some categorical variable; for example, we might want to know what the distribution of `Sepal.Length` is among specific types of flowers (eg virginica and setosa). We can use base r subsetting operations to do this:

```
summary(iris$Sepal.Length[iris$Species == 'virginica'])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.900   6.225   6.500   6.588   6.900   7.900
```

```
summary(iris$Sepal.Length[iris$Species == 'setosa'])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.300   4.800   5.000   5.006   5.200   5.800
```

Alternatively, we can use the command `tapply`; this is especially useful when the categorical variable has several values:

```
tapply(iris$Sepal.Length, iris$Species, summary)
```

```
## $setosa
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.300   4.800   5.000   5.006   5.200   5.800
##
## $versicolor
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.900   5.600   5.900   5.936   6.300   7.000
##
## $virginica
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.900   6.225   6.500   6.588   6.900   7.900
```

Other useful summary statistics include `range`, `mean`, `median`, and `quantile`, which all return the statistics you would expect.

```
range(iris$Sepal.Length)
```

```
## [1] 4.3 7.9
```

Returning distinct values of a variable

Sometimes we may have a categorical variable and we want to directly examine the distinct of unique categories. One way to do this is using the `unique` command:

```
unique(iris$Species)
```

```
## [1] setosa     versicolor virginica
## Levels: setosa versicolor virginica
```

Note that `unique` returns a vector. We can get the same information as a dataframe by using the `distinct` command in combination with `select`:

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

```
##      Species
## 1     setosa
## 2 versicolor
## 3  virginica
```

Creating a New Variable in a dataframe

You may wish to create a new variable in a dataframe, either with entirely new values or with values that are functions of other columns in the existing dataframe. We recommend using the `mutate` command to do this.

```
#tidyverse
mtcars <- mtcars %>%
  mutate(mpg_by_cyl = mpg/cyl)

#base r (alternative)
mtcars$mpg_by_cyl <- mtcars$mpg/mtcars$cyl
```

Recoding categorical variables

Say we want to recode a variable based on existing values. One option is `if_else` statements. Another option is to use `case_when`.

For example, say we want to recode the flower types `virginica`, `versicolor`, and `setosa` to only two categories: `setosa` and `other`. We can use `case_when` or `if_else` as below. Note that `case_when` will is more helpful than `if_else` the number of categories we wish to recode to is greater than two.

```
# if_else example
iris <- iris %>%
  mutate(new_species = if_else(Species == 'versicolor' | Species == 'virginica', 'other', 'setosa'))
```

Notice that in the example above we use the `|` command to indicate or; so the `if_else` statement reads that if species is versicolor or species is virginica to return the value ‘other’; otherwise, the species is setosa. Alternatively, we could have written `if_else(Species == 'setosa', 'setosa', 'other')`.

Summary statistics of data by groups

Suppose we want to know specific functions of a distribution again by the values of some categorical variable. In the example below we calculate the mean and standard deviation of petal length the three types of species in the iris dataset.

```
iris %>%
  group_by(Species) %>%
  summarize(mean.pl = mean(Petal.Length),
            sd.pl   = sd(Petal.Length))
```

```
## # A tibble: 3 x 3
##   Species    mean.pl sd.pl
##   <fct>        <dbl> <dbl>
## 1 setosa        1.46 0.174
## 2 versicolor    4.26 0.470
## 3 virginica     5.55 0.552
```

Note that we can't use the `summary` command in this framework because it returns a vector; however, you can use specific commands, eg `mean`, `sd`, `min`, `max`, that return individual values.

Filtering a Dataset

Many times, you may want or need to select certain rows of your data to focus on a specific segment. There are many ways to do this in R. We recommend using the `filter` function; however, both methods below will subset the dataset and return a new dataset with all of the columns present in the original dataset.

```
##  Example: Select the rows from the iris dataset where the Species is Virginica and
## the Petal.Length is longer than 5.2 units
iris <- iris %>%
  filter(Species == "virginica" & Petal.Length > 5.2)

## Alternative using base r
iris <- iris[iris$Species == "virginica" & iris$Petal.Length > 5.2, ]
```

Summarize Subsetted Data

Other times, it is helpful to filter your data from a data frame with multiple variables to then summarize the value of some other variable. This is most often used when preparing a dataset for a specific calculation.

```
## Example: What is the mean Petal.Width of Virginica flowers with Petal.Length longer than 5.2 units?
iris %>%
  filter(Species == "virginica" & Petal.Length > 5.2) %>%
  summarize(Petal.Width = mean(Petal.Width))

## Example using the $ and [ ] method
mean(iris$Petal.Width[iris$Species == "virginica" & iris$Petal.Length > 5.2])
```

Week Two

Summarizing data by groups

We continue discussing summarizing data by groups here; see week one for taking functions of a subgroup of a dataset; here we consider some more complicated operations.

Taking differences within groups

Here we create a column that takes the difference in the mean petal length between versicolor and setosa species of iris. Note that this combines tidyverse syntax with base r vector subsetting notation.

```
my_summary <- iris %>%
  filter(Species != 'virginica') %>%
  group_by(Species) %>%
  summarize(mean.pl = mean(Petal.Length))
```

```
fx1 <- my_summary$mean.pl[my_summary$Species == 'versicolor'] - my_summary$mean.pl[my_summary$Species ==
fx1
```

```
## [1] 2.798
```

Quantiles

Say we want to know the 0th percentile, 25th percentile, median, 75th percentile, and 100th percentile of petal length. In this case we can use the `quantile` function. This function takes a vector input and an argument, `probs`, that reflects the specific quantiles of interest.

```
quantile(iris$Petal.Length, probs = c(0, 0.25, 0.5, 0.75, 1))
```

```
##   0%  25%  50%  75% 100%
## 1.00 1.60 4.35 5.10 6.90
```

Now assume we want to know the difference in the petal length quantiles between versicolor and virginia species. We can simply take the difference between the two quantiles!

```
quantile(iris$Petal.Length[iris$Species == 'versicolor'], probs = c(0, 0.25, 0.5, 0.75, 1)) -
  quantile(iris$Petal.Length[iris$Species == 'virginica'], probs = c(0, 0.25, 0.5, 0.75, 1))
```

```
##     0%    25%    50%    75%   100%
## -1.500 -1.100 -1.200 -1.275 -1.800
```

Tables

First we count the number of observations of `cyl` ersus `carb` in the `mtcars` dataset.

Second we calculate the total proportion of each of these cells.

Third we generate row proportions. Fourth we generate column proportions.

```
# counts
table(mtcars$cyl, mtcars$carb)
```

```
##
##     1 2 3 4 6 8
##   4 5 6 0 0 0 0
##   6 2 0 0 4 1 0
##   8 0 4 3 6 0 1
```

```
# joint distribution props
prop.table(table(mtcars$cyl, mtcars$carb))
```

```
##
##           1       2       3       4       6       8
##   4 0.15625 0.18750 0.00000 0.00000 0.00000 0.00000
##   6 0.06250 0.00000 0.00000 0.12500 0.03125 0.00000
##   8 0.00000 0.12500 0.09375 0.18750 0.00000 0.03125
```

```
# marginal distribution props
prop.table(table(mtcars$cyl, mtcars$carb), margin = 1) #row proportions
```

```
##
##             1          2          3          4          6          8
##   4 0.45454545 0.54545455 0.00000000 0.00000000 0.00000000 0.00000000
##   6 0.28571429 0.00000000 0.00000000 0.57142857 0.14285714 0.00000000
##   8 0.00000000 0.28571429 0.21428571 0.42857143 0.00000000 0.07142857
```

```r
prop.table(table(mtcars$cyl, mtcars$carb), margin = 2) #col proportions
```

```
##
##             1         2         3         4         6         8
##   4 0.7142857 0.6000000 0.0000000 0.0000000 0.0000000 0.0000000
##   6 0.2857143 0.0000000 0.0000000 0.4000000 1.0000000 0.0000000
##   8 0.0000000 0.4000000 1.0000000 0.6000000 0.0000000 1.0000000
```

NAs

na.rm

Many functions - for example `mean` and `sd` - will return `NA` or `NaN` if there are missing data elements within the input vector. Often we wish to disregard the missing data and simply to calculate the values from the observed data. To do this, we include the argument `na.rm = TRUE`.

```r
my_data <- c(NA, 1, 2, 3, NA)
mean(my_data) # returns NA
```

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

```r
mean(my_data, na.rm = TRUE) # returns 2
```

```
## [1] 2
```

Similarly, you may also sometimes need to remove NAs after calling the summarize command when calling a function such as `mean` or `median`. For example:

```r
tibble(x = c(NA, 1, 2, 3, 4),
       y = c(1:5)) %>%
  summarize(mean_x = mean(x, na.rm = TRUE),
            median_x = median(x, na.rm = T))
```

```
## # A tibble: 1 x 2
##   mean_x median_x
##    <dbl>    <dbl>
## 1    2.5      2.5
```

Notice that if you do not add this argument you will return NA similar to when you apply these functions on a vector input:

```r
tibble(x = c(NA, 1, 2, 3, 4),
       y = c(1:5)) %>%
  summarize(mean_x = mean(x),
            median_x = median(x))
```

```
## # A tibble: 1 x 2
##   mean_x median_x
##    <dbl>    <dbl>
## 1     NA       NA
```

Filtering rows by NA

You may wish to remove rows directly from your dataframe that contain `NA`. This is the proper way to do this:

```r
tibble(x = c(NA, 1, 2, 3, 4),
       y = c(1:5)) %>%
  filter(!is.na(x))
```

```
## # A tibble: 4 x 2
##       x     y
##   <dbl> <int>
## 1     1     2
## 2     2     3
## 3     3     4
## 4     4     5
```

We note this in particular because you may be tempted to run the following:

```
# BAD EXAMPLE: NEVER FOLLOW THIS
tibble(x = c(NA, 1, 2, 3, 4),
       y = c(1:5)) %>%
  filter(x != NA)
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: x <dbl>, y <int>
```

However, this does not work.

Within text values

One nice feature of Rmd values is that you can call numbers within the text of the documet that are stored in a previous r code chunk. For example, say I want to store the value of the mean of the integers ranging from 1 to 100:

```
my_value <- mean(c(1:200))
```

I can then call the value, 100.5, in-line by using the syntax demonstrated in the Rmd file (refer to this if you are looking at the PDF).

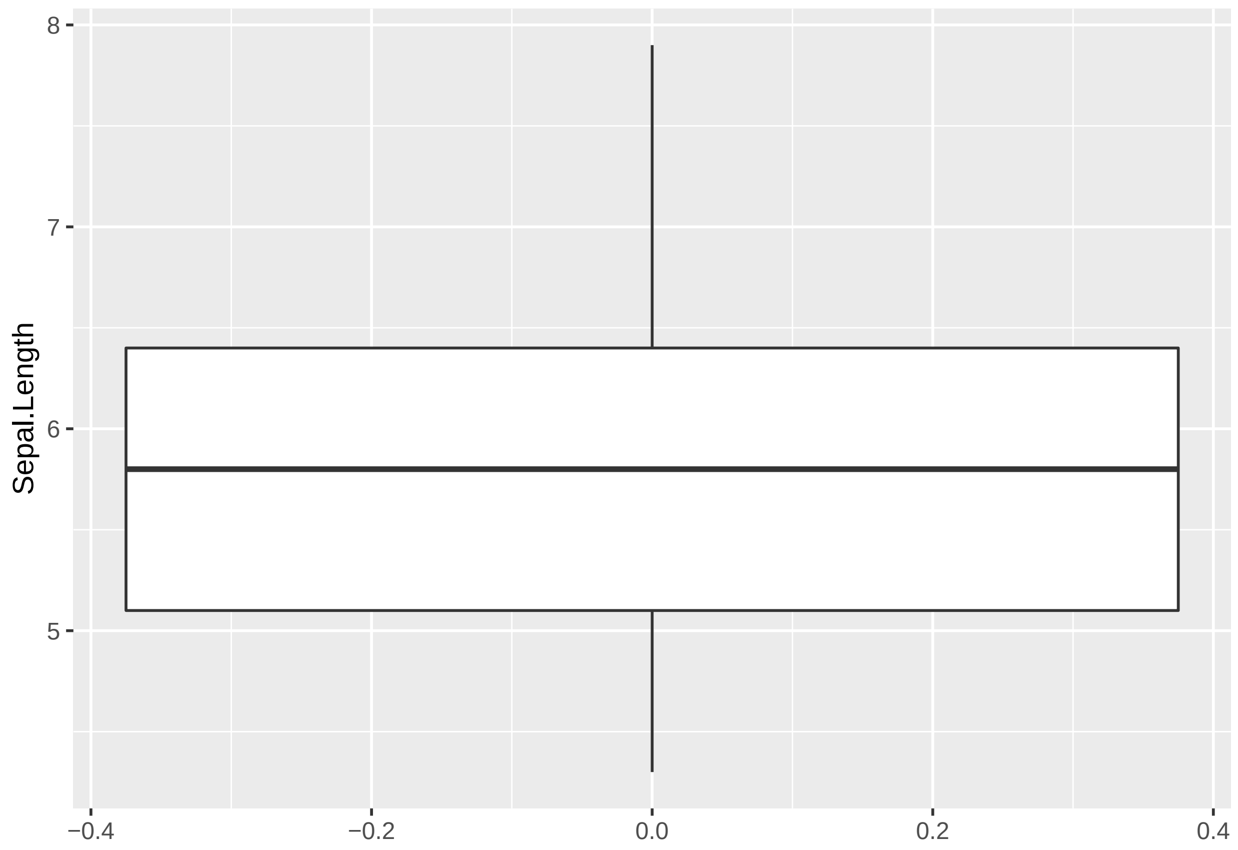
Week Three

For this week we introduce `ggplot` and demonstrate how to take differences in quantiles by groups, which you will need for your homework. We also include a couple of additional helpful tricks on using the `summarize` function and reshaping your dataframe that may be useful in the future.

ggplot

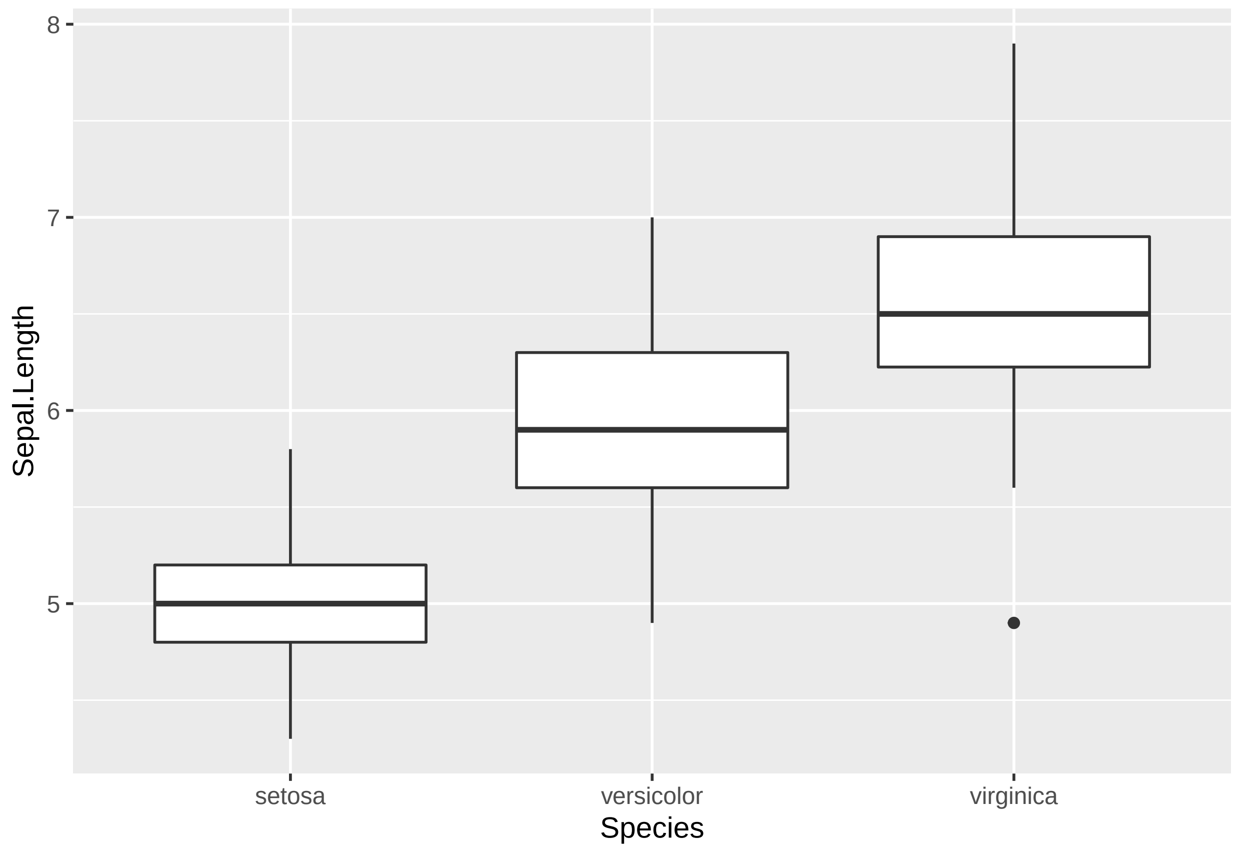
`ggplot2` is a part of the `tidyverse` that allows you to visualize your data. Here we introduce `ggplot2` by covering how to make boxplots.

```
iris %>%
  ggplot(aes(y = Sepal.Length)) + #notice the + rather than %>%
  geom_boxplot()
```



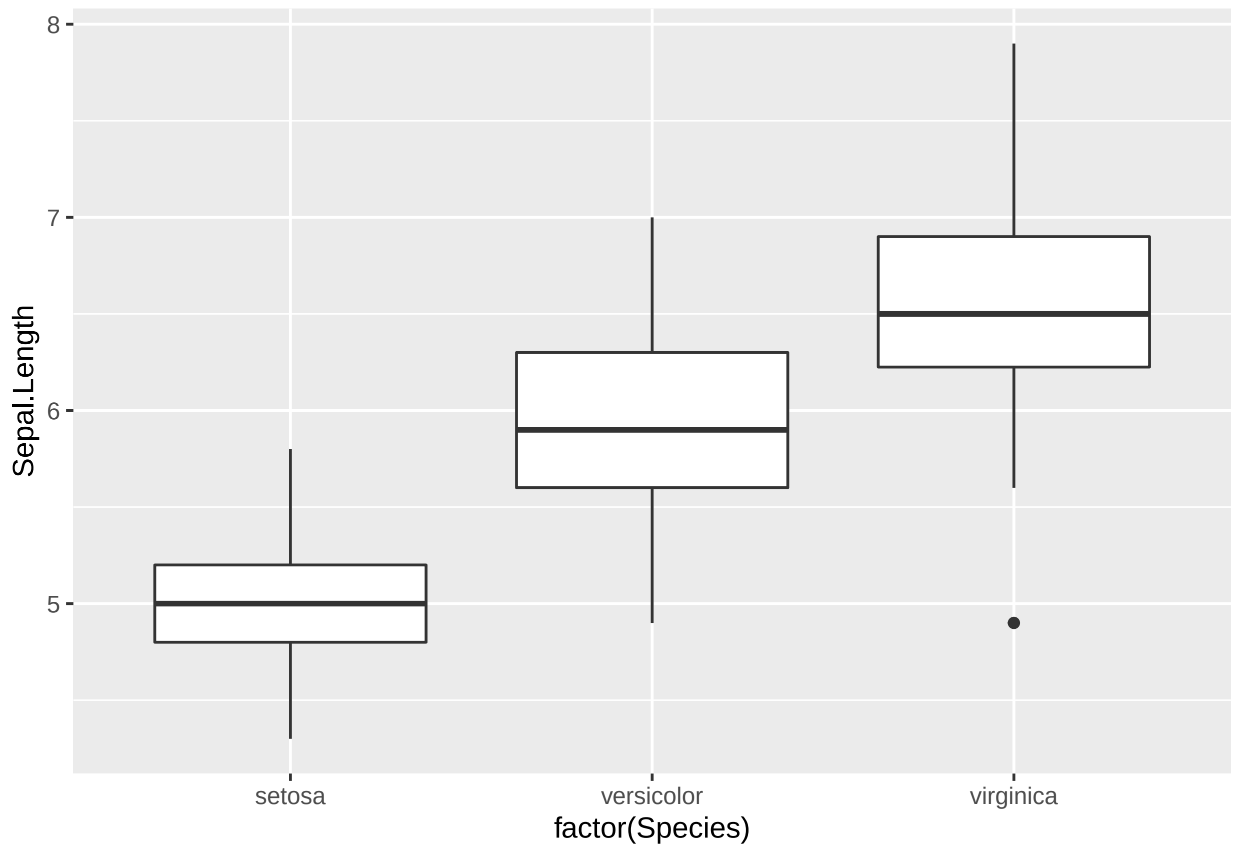
Say I want to look at boxplots of the distribution of Sepal.Length for each species of flower. I can include the additional boxes as an x aesthetic:

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



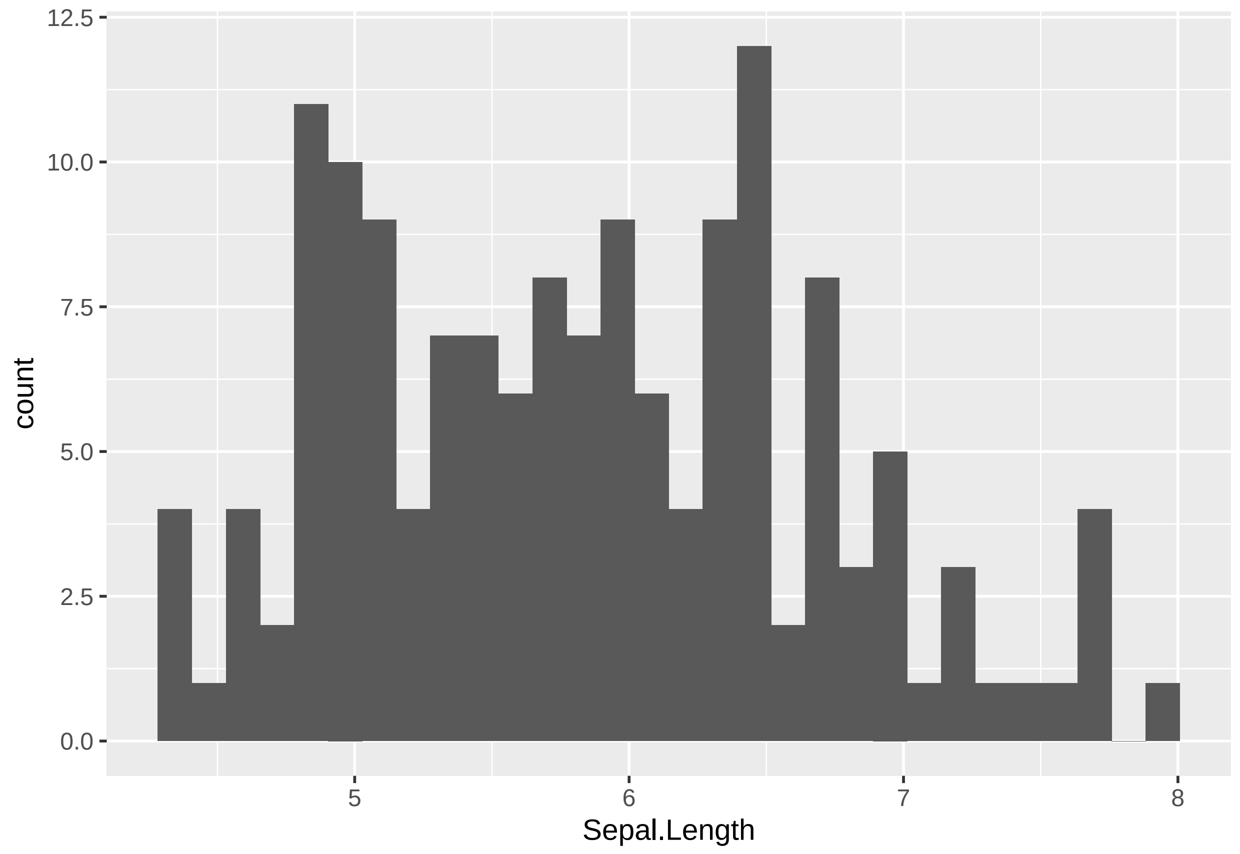
Note however, that x must be specified as a factor variable or this won't work. If your data has categorical variables, you can turn the variable into a factor variable using the `factor` command within the aesthetic call.

```
iris %>%
  ggplot(aes(y = Sepal.Length, x = factor(Species))) +
  geom_boxplot()
```



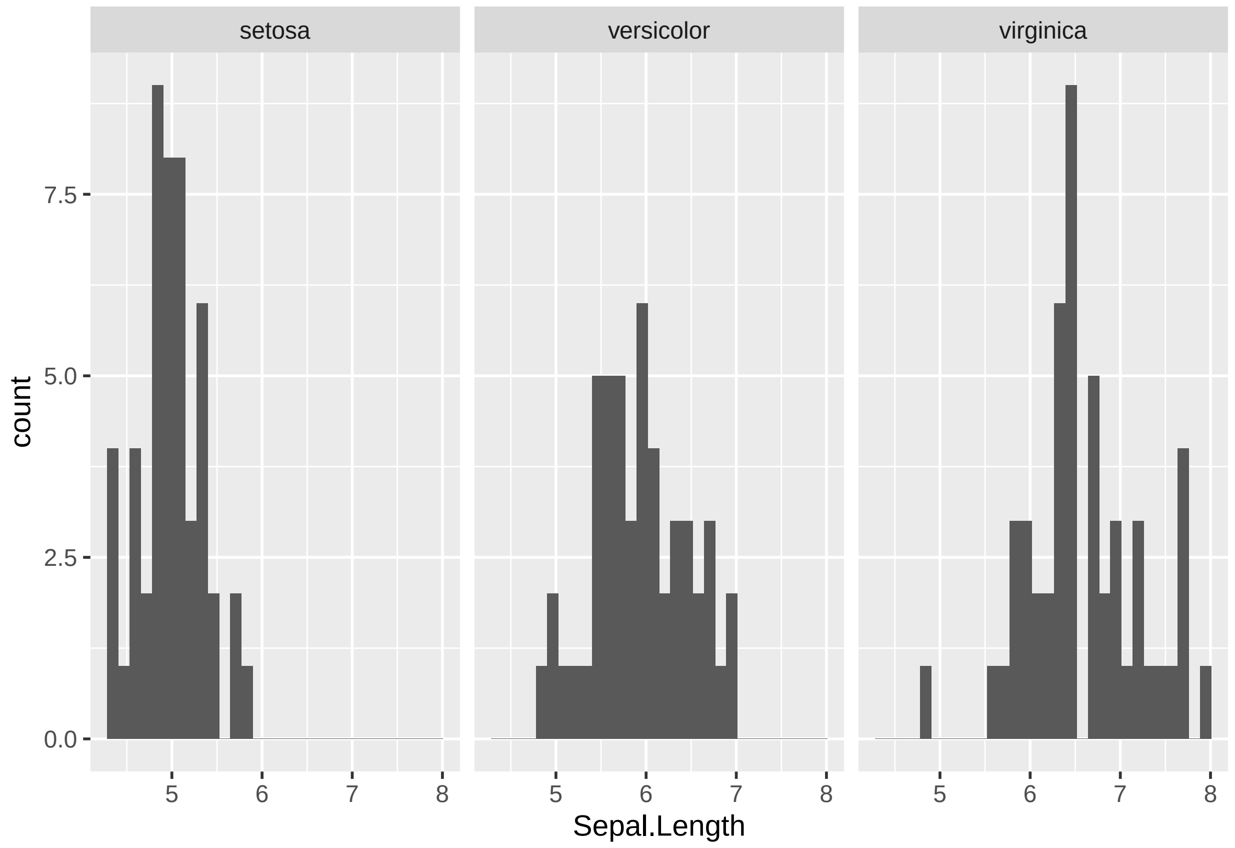
Lastly, another useful type of plot to summarize a univariate distribution is the histogram. We can also make a histogram as follows:

```
iris %>%
  ggplot(aes(x = Sepal.Length)) +
  geom_histogram()
```



Again we may wish to see multiple histograms on the same plot. In this case the coding is a bit more complicated: intuitively, this is because to create multiple histograms we essentially need multiple plots in a process called `faceting`. Contrast this to a boxplot, where we can simply put different factor levels on another axis. Regardless, faceting is extremely useful:

```
iris %>%
  ggplot(aes(x = Sepal.Length)) +
  geom_histogram() +
  facet_wrap(~Species)
```



Other useful `geoms` (ie chartypes) you may wish to experiment with include `geom_line` for line graphs, `geom_point` for points. When you have multiple dimensional data, you can specify x, y, in the `aes` command. You can read more about `ggplot2` generally online, though we will include more useful commands here from the `ggplot2` library in the following weeks.

Using summarize

###summarize with quantiles

The `quantile` function takes a vector argument and returns a vector, so it can be more difficult to use within the `tidyverse`. We therefore show the syntax necessary to make this work first in `tidyverse` and then in base r. We then demonstrate how to take the differences in quantiles by group.

Below we show how to calculate the quartiles of `Sepal.Length` for each `Species` within the `iris` dataset. We then show how to take the difference in these quantiles.

Note: this was also briefly touched on above, but I cover it again here because this does come up on Homework 3.

The easiest way to take differences in quantiles by groups is by using the base r function `tapply`:

```r
sl.quantiles <- tapply(iris$Sepal.Length, iris$Species, quantile, probs = c(0, 0.25, 0.5, 0.75, 1))

sl.quantiles
```

```
## $setosa
##   0%  25%  50%  75% 100% 
##  4.3  4.8  5.0  5.2  5.8 
## 
## $versicolor
##   0%  25%  50%  75% 100% 
##  4.9  5.6  5.9  6.3  7.0 
## 
## $virginica
##    0%   25%   50%   75%  100% 
## 4.900 6.225 6.500 6.900 7.900
```

```
sl.quantiles$versicolor - sl.quantiles$setosa
```

```
##   0%  25%  50%  75% 100% 
##  0.6  0.8  0.9  1.1  1.2
```

Notice that the output of `tapply` is a list of quantiles (`class(sl.quantiles)`). Each element of the list corresponds to the particular value of `Species`. we can access each element of this (named) list using the `$` operator. In the final line we took the difference in the quantiles of versicolor versus setosa species.

While base r is actually somewhat simpler than `tidyverse` for this calculation; however, as with most base r functions, it is less immediately obvious what you actually did when reviewing the code later. Moreover, the output is a list of vectors, rather than a dataframe, which may not be the most desirable format for your output. You can also calculate this using the `tidyverse` syntax we've already seen is below.

First, we simply calculate quantiles using functions we've seen previously:

```
iris %>%
  group_by(Species) %>%
  mutate(q25 = quantile(Sepal.Length, probs = 0.25),
         q50 = quantile(Sepal.Length, probs = 0.5),
         q75 = quantile(Sepal.Length, probs = 0.75)) %>%
  distinct(q25, q50, q75)
```

```
## # A tibble: 3 x 4
## # Groups:   Species [3]
##   Species      q25   q50   q75
##   <fct>      <dbl> <dbl> <dbl>
## 1 setosa      4.8    5     5.2
## 2 versicolor  5.6    5.9   6.3
## 3 virginica   6.22   6.5   6.9
```

You may, however, not like typing the `quantile` function repeatedly. I'm not going to explain the syntax below, but here is somewhat nicer way to do this using `tidyverse` syntax for your reference (see below for more details on `pivot_wider`.

```
p = c(0, 0.25, 0.5, 0.75, 1) #specify quantiles in a separate vector

iris_quantiles <- iris %>%
  group_by(Species) %>%
  summarise(quantiles = list(sprintf("q%s", p*100)),
            sepal.length = list(quantile(Sepal.Length, p))) %>%
  unnest() %>%
  pivot_wider(id_cols = 'Species', names_from = quantiles, values_from = sepal.length)
```

To take the differences in quantiles between one category and others (for example, between each species and `setosa`, notice that our data is not in the most helpful format; therfore, we need to reshape the data (see below for more details on `pivot_longer`). Briefly, however, we can do that as follows (only the first six rows are output below):

```
iris_quantiles %>%
  pivot_longer(cols = starts_with('q'), names_to = 'key', values_to = 'value') %>%
  group_by(key) %>%
  mutate(diff = value - value[Species == 'setosa']) %>%
  head(6)
```

```
## # A tibble: 6 x 4
## # Groups:   key [5]
##   Species    key   value  diff
##   <fct>      <chr> <dbl> <dbl>
## 1 setosa     q0      4.3   0
## 2 setosa     q25     4.8   0
## 3 setosa     q50     5     0
## 4 setosa     q75     5.2   0
## 5 setosa     q100    5.8   0
## 6 versicolor q0      4.9   0.6
```

summarize_at

Sometimes you may want to summarize multiple variables with the same functions. For example, we may want the mean and standard deviation of both `Sepal.Length` and `Petal.Length`.

```
iris %>%
  summarize_at(vars(contains('Length')), funs(mean, sd))
```

```
##   Sepal.Length_mean Petal.Length_mean Sepal.Length_sd Petal.Length_sd
## 1          5.843333             3.758       0.8280661        1.765298
```

Notice the `contains` argument nested within the `vars` argument; this will select all columns that contain the string `Length`. Other useful functions here include `starts_with` and `ends_with` (what do you think these mean?) You can also simply include a character vector with the column names, eg,

```
iris %>%
  summarize_at(c("Sepal.Length", "Petal.Length"), funs(mean, sd))
```

```
##   Sepal.Length_mean Petal.Length_mean Sepal.Length_sd Petal.Length_sd
## 1          5.843333             3.758       0.8280661        1.765298
```

Finally, notice that the `list` argument takes the functions you want to apply to each of these columns, here including `mean` and `sd`, which are preceded by a `~`.

reshaping data

Sometimes you may wish to reshape your data. The two key functions here are `pivot_longer` and `pivot_wider`. `pivot_longer` reshapes your data from wide to long, `pivot_wider` from long to wide. For example, we may want one column in the `iris` dataset with the *values* of both sepal and petal length, which the variable name listed as a *key*.

```
iris_new <- iris %>%
  mutate(id = 1:nrow(.)) %>%
  pivot_longer(names_to = 'sepal_or_petal',
               values_to = 'value',
               cols = c('Sepal.Length', 'Petal.Length'))

head(iris_new)
```

```
## # A tibble: 6 x 7
##   Sepal.Width Petal.Width Species new_species    id sepal_or_petal value
##         <dbl>       <dbl> <fct>   <chr>       <int> <chr>          <dbl>
## 1         3.5         0.2 setosa  setosa          1 Sepal.Length     5.1
## 2         3.5         0.2 setosa  setosa          1 Petal.Length     1.4
## 3         3           0.2 setosa  setosa          2 Sepal.Length     4.9
## 4         3           0.2 setosa  setosa          2 Petal.Length     1.4
## 5         3.2         0.2 setosa  setosa          3 Sepal.Length     4.7
## 6         3.2         0.2 setosa  setosa          3 Petal.Length     1.3
```

By contrast, we can go from long to wide using `pivot_wider`. Spread simply takes the arguments of a column whose values become the column names (eg sepal_or_petal) and a column of values to populate those new columnes (eg value). *Warning:* `pivot_wider` will not work if you do not have unique ids for each observation (why I created the `id` variable above).

```
iris_new %>%
  pivot_wider(id_cols = 'id', names_from = 'sepal_or_petal', values_from = 'value') %>%
  head()
```

```
## # A tibble: 6 x 3
##      id Sepal.Length Petal.Length
##   <int>        <dbl>        <dbl>
## 1     1          5.1          1.4
## 2     2          4.9          1.4
## 3     3          4.7          1.3
## 4     4          4.6          1.5
## 5     5          5            1.4
## 6     6          5.4          1.7
```