Statistical Computing

Isaac Quintanilla Salinas

Table of contents

Welcome

Preface

This is a book created to be used for a statistical computing course at the undergraduate level.

Part I

R Programming

1 Basic R Programming

1.1 Introduction

This chapter focuses on the basics of R programming. While most of your statistical analysis will be done with R functions, it is important to have an idea of what is going on. Additionally, we will cover other topics that you may or may not need to know. The topics we will cover are:

1. Basic calculations in R
2. Types of Data
3. R Objects
4. R Functions
5. R Packages

1.2 Basic Calculations

This section focuses on the basic calculation that can be done in R. This is done by using different operators in R. The table below provides some of the basic operators R can use:

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Divides
`^` or `**`	Exponentiate
`?`	Help Documentation

1.2.1 Calculator

1.2.1.1 Addition

To add numbers in R, all you need to use the `+` operator. For example $2 + 2 = 4$. When you type it in R you have:

```
2 + 2
```

```
[1] 4
```

When you ask R to perform a task, it prints out the result of the task. As we can see above, R prints out the number 4.

To add more than 2 numbers, you can simply just type it in.

```
2 + 2 + 2
```

```
[1] 6
```

This provides the number 6.

1.2.1.2 Subtraction

To subtract numbers, you need to use the `-` operator. Try `4 - 2`:

```
4 - 2
```

```
[1] 2
```

Try `4 - 6 - 4`

```
4 - 6 - 4
```

```
[1] -6
```

Notice that you get a negative number.

Now try `4 + 4 - 2 + 8`:

```r
4 + 4 - 2 + 8
```

```
[1] 14
```

1.2.1.3 Multiplication

To multiply numbers, you will need to use the `*` operator. Try `4 * 4`:

```r
4 * 4
```

```
[1] 16
```

1.2.1.4 Division

To divide numbers, you can use the `/` operator. Try `9 / 3`:

```r
9 / 3
```

```
[1] 3
```

1.2.1.5 Exponents

To exponentiate a number to the power of another number, you can use the `^` operator. Try `2^5`:

```r
2^5
```

```
[1] 32
```

If you want to find e^2, you will use the `exp()` function. Try `exp(2)`:

```r
exp(2)
```

```
[1] 7.389056
```

1.2.1.6 Roots

To take the n-th root of a value, use the `^` operator with the `/` operator to take the n-th root. For example, to take $\sqrt[5]{35}$, type `32^(1/5)`:

```r
32^(1/5)
```

```
[1] 2
```

1.2.1.7 Logarithms

To take the natural logarithm of a value, you will use the `log()` function. Try `log(5)`:

```r
log(5)
```

```
[1] 1.609438
```

If you want to take the logarithm of a different base, you will use the `log()` function with `base` argument. We will discuss this more in Section 1.4.

1.2.2 Comparing Numbers

Another important part of R is comparing numbers. When you compare two numbers, R will tell if the statement is `TRUE` or `FALSE`. Below are the different comparisons you can make:

Operator	Description
`>`	Greater Than
`<`	Less Than
`>=`	Greater than or equal
`<=`	Less than or equal
`==`	Equals
`!=`	Not Equals

1.2.2.1 Less than/Greater than

To check if one number is less than or greater than another number, you will use the `>` or `<` operators. Try `5 > 4`:

```r
5 > 4
```

```
[1] TRUE
```

Notice that R states it's true. It evaluates the expression and tells you if it's true or not. Try `5 < 4`:

```r
5 < 4
```

```
[1] FALSE
```

Notice that R tells you it is false.

1.2.2.2 Less than or equal to/Greater than or equal to

To check if one number is less than or equal to/greater than or equal to another number, you will use the `>=` or `<=` operators. Try `5 >= 5`:

```r
5 >= 5
```

```
[1] TRUE
```

Try `5 >= 4`:

```r
5 >= 4
```

```
[1] TRUE
```

Try `5 <= 4`

```r
5 <= 4
```

```
[1] FALSE
```

1.2.2.3 Equals and Not Equals

To check if 2 numbers are equal to each other, you can use the `==` operator. Try `3 == 3`:

```r
3 == 3
```

```
[1] TRUE
```

Try `4 == 3`

```r
3 == 4
```

```
[1] FALSE
```

Another way to see if 2 numbers are not equal to each other, you can use the `!=`. Try `3 != 4`:

```r
3 != 4
```

```
[1] TRUE
```

Try `3 != 3`:

```r
3 != 3
```

```
[1] FALSE
```

You may be asking why use `!=` instead of `==`. They both provides similar results. Well the reason is that you may need the `TRUE` output for analysis. One is only true when they are equal, while the other is true when they are not equal.

1.2.3 Help

The last operator we will discuss is the help operator `?`. If you want to know more about anything we talked about you can type `?` in front of a function and a help page will pop-up in your browser or in RStudio's 'Help' tab. For example you can type `?Arithmetic` or `?Comparison`, to review what we talked about. For other operators we didn't talk about use `?assignOps` and `?Logic`.

1.3 Types of Data

In R, the type of data, also known as class, we are using dictates how the programming works. For the most part, users will use *numeric*, *logical*, *POSIX* and *character* data types. Other types of data you may encounter are *complex* and *raw*. To obtain more information on them, use the `?` operator.

1.3.1 Numeric

The *numeric* class is the data that are numbers. Almost every analysis that you use will be based on the numeric class. To check if you have a numeric class, you just need to use the `is.numeric()` function. For example, try `is.numeric(5)`:

```r
is.numeric(5)
```

```
[1] TRUE
```

Numeric classes are essentially *double* and *integer* types of data. For example a *double* data is essentially a number with decimal value. An *integer* data are whole numbers. Try `is.numeric(5.63)`, `is.double(5.63)` and `is.integer(5.63)`:

```r
is.numeric(5.63)
```

```
[1] TRUE
```

```r
is.double(5.63)
```

```
[1] TRUE
```

```r
is.integer(5.63)
```

```
[1] FALSE
```

Notice how the value 5.63 is a *numeric* and *double* but not *integer*. Now let's try `is.numeric(7)`, `is.double(7)` and `is.integer(7)`:

```r
is.numeric(7)
```

```
[1] TRUE
```

```r
is.double(7)
```

```
[1] TRUE
```

```r
is.integer(7)
```

```
[1] FALSE
```

Notice how the value 7 is also considered a *numeric* and *double* but not *integer*. This is because typing a whole number will be stored as a *double*. However, if we need to store an *integer*, we will need to type the letter "L" after the number. Try `is.numeric(7L)`, `is.double(7L)`, and `is.integer(7L)`:

```r
is.numeric(7L)
```

```
[1] TRUE
```

```r
is.double(7L)
```

```
[1] FALSE
```

```r
is.integer(7L)
```

```
[1] TRUE
```

1.3.2 Logical

A *logical* class are data where the only value is `TRUE` or `FALSE`. Sometimes the data is coded as `1` for `TRUE` and `0` for `FALSE`. The data may also be coded as `T` or `F`. To check if data belongs in the *logical* class, you will need the `is.logical()` function. Try `is.logical(3 < 4)`:

```r
is.logical(3 < 4)
```

```
[1] TRUE
```

This is same comparison from Section 1.2.2. The output was `TRUE`. Now R is checking whether the output is of a *logical* class. Since it it, R returns `TRUE`. Now try `is.logical(3 > 4)`:

```r
is.logical(3 > 4)
```

```
[1] TRUE
```

The output is `TRUE` as well even though the condition `3 > 4` is `FALSE`. Since the output is a *logical* data type, it is a *logical* variable.

1.3.3 POSIX

The *POSIX* class are date-time data. Where the data value is a time component. The *POSIX* class can be very complex in how it is formatted. IF you would like to learn more try `?POSIXct` or `?POSIClt`. First, lets run `Sys.time()` to check what is today's data and time:

```r
Sys.time()
```

```
[1] "2023-03-11 08:19:13 PST"
```

Now lets check if its of POSIX class, you can use the `class()` function to figure out which class is it. Try `class(Sys.time())`:

```r
class(Sys.time())
```

```
[1] "POSIXct" "POSIXt"
```

1.3.4 Character

A *character* value is where the data values follow a *string* format. Examples of *character* values are letters, words and even numbers. A *character* value is any value surrounded by quotation marks. For example, the phrase "Hello World!" is considered as one *character* value. Another example is if your data is coded with the actual words "yes" or "no". To check if you have *character* data, use the `is.character()` function. Try `is.character("Hello World!")`:

```r
is.character("Hello World!")
```

```
[1] TRUE
```

Notice that the output says `TRUE`. *Character* values can be created with single quotations. Try `is.character('Hello World!')`:

```r
is.character('Hello World!')
```

```
[1] TRUE
```

1.3.5 Complex Numbers

Complex numbers are data values where there is a real component and an imaginary component. The imaginary component is a number multiplied by $i = \sqrt{-1}$. To create a *complex* number, use the `complex()` function. To check if a number is complex, use the `is.complex()` function. Try the following to create a complex number `complex(1, 4, 5)`:

```r
complex(1, 4, 5)
```

```
[1] 4+5i
```

Now try `is.complex(complex(1, 4, 5))`:

```r
is.complex(complex(1, 4, 5))
```

```
[1] TRUE
```

1.3.6 Raw

You will probably never use raw data. I have never used raw data in R. To create a raw value, use the `raw()` or `charToRaw()` functions. Try `charToRaw('Hello World!')`:

```
charToRaw('Hello World!')
```

```
 [1] 48 65 6c 6c 6f 20 57 6f 72 6c 64 21
```

To check if you have raw data, use the `is.raw()` function. Try `is.raw(charToRaw('Hello World!'))`:

```
is.raw(charToRaw('Hello World!'))
```

```
[1] TRUE
```

1.3.7 Missing

The last data class in R is missing data. The table below provides a brief introduction of the different types of missing data

Value	Description	Functions
`NULL`	These are values indicating an object is empty. Often used for functions with values that are undefined.	`is.null()`
`NA`	Stands for "Not Available", used to indicate that the value is missing in the data.	`is.na()`
`NaN`	Stands for "Not an Number". Used to indicate a missing number.	`is.nan()`
`Inf` and `-Inf`	Indicating an extremely large value or a value divided by 0.	`is.infinite()`

1.4 R Functions

An R function is the procedure that R will execute to certain data. For example, the `log(x)` is an R function. It takes the value `x` and provides you the natural logarithm. Here `x` is known as an argument which needs to be specified to us the `log()` function. Find the `log(x = 5)`

```
log(x = 5)
```

```
[1] 1.609438
```

Another argument for the `log()` function is the `base` argument. With the previous code, we did not specify the `base` argument, so R makes the `base` argument equal to the number e. If you want to use the common log with base 10, you will need to set the `base` argument equal to 10.

Try `log(x = 5, base = 10)`

```
log(x = 5, base = 10)
```

```
[1] 0.69897
```

Now try `log(5,10)`

```
log(5,10)
```

```
[1] 0.69897
```

Notice that it provides the same value. This is because R can set arguments based on the values position in the function, regardless if the arguments are specified. For `log(5,10)`, R thinks that 5 corresponds to the first argument `x` and 10 is the second argument `base`.

To learn more about a functions, use the `?` operator on the function: `?log`.

1.5 R Objects

R objects are where most of your data will be stored. An R object can be thought of as a container of data. Each object will share some sort of characteristics that will make the unique for different types of analysis.

1.5.1 Assigning objects

To create an R object, all we need to do is assign data to a variable. The variable is the name of the R object. it can be called anything, but you can only use alphanumeric values, underscore, and periods. To assign a value to a variable, use the `<-` operator. This is known a left assignment. Kinda like an arrow pointing left. Try assigning 9 to 'x' (`x <- 9`):

```r
x <- 9
```

To see if `x` contains 9, type `x` in the console:

```r
x
```

```
[1] 9
```

Now `x` can be treated as data and we can perform data analysis on it. For example, try squaring it:

```r
x^2
```

```
[1] 81
```

You can use any mathematical operation from the previous sections. Try some other operations and see what happens.

The output R prints out can be stored in a variable using the asign operator, `<-`. Try storing `x^3` in a variable called `x_cubed`:

```r
x_cubed <- x^3
```

To see what is stored in `x_cubed` you can either type `x_cubed` in the console or use the `print()` function with `x_cubed` inside the parenthesis.

```r
x_cubed
```

```
[1] 729
```

```r
print(x_cubed)
```

```
[1] 729
```

1.5.2 Vectors

A vector is a set data values of a certain length. The R object `x` is considered as a numerical vector (because it contains a number) with the length 1. To check, try `is.numeric(x)` and `is.vector(x)`:

```r
is.numeric(x)
```

```
[1] TRUE
```

```r
is.vector(x)
```

```
[1] TRUE
```

Now let's create a logical vector that contains 4 elements (have it follow this sequence: T, F, T, F) and assign it to `y`. To create a vector use the `c()`[1] function and type all the values and separating them with columns. Type `y <- c(T, F, T, F)`:

```r
y <- c(T, F, T, F)
```

Now, lets see how `y` looks like. Type `y`:

```r
y
```

```
[1]  TRUE FALSE  TRUE FALSE
```

Now lets see if it's a logical vector:

```r
is.logical(y)
```

```
[1] TRUE
```

```r
is.vector(y)
```

```
[1] TRUE
```

Fortunately, this vector is really small to count how many elements it has, but what if the vector is really large? To find out how many elements a vector has, use the `length()` function. Try `length(y)`:

[1]The `c()` function allows you to put any data type and as many values as you wish. The only condition of a vector is that it must be the same data type.

```r
length(y)
```

```
[1] 4
```

1.5.3 Matrices

A matrix can be thought as a square or rectangular grid of data values. This grid can be constructed can be any size. Similar to vectors they must contain the same data type. The size of a matrix is usually denoted as $n \times k$, where n represents the number of rows and k represents the number of columns. To get a rough idea of how a matrix may look like, type `matrix(rep(1,12), nrow = 4, ncol = 3)`[2]:

```r
matrix(rep(1, 12), nrow = 4, ncol = 3)
```

```
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
[4,]    1    1    1
```

Notice that this is a 4×3 matrix. Each element in the matrix has the value 1. Now try this `matrix(rbinom(12,1.5), nrow = 4, ncol = 3)`[3]:

```r
matrix(rbinom(12, 1, .5), nrow = 4, ncol = 3)
```

```
     [,1] [,2] [,3]
[1,]    1    1    0
[2,]    0    1    1
[3,]    0    0    1
[4,]    0    0    0
```

[2]The function `rep()` creates a vector by repeating a value for a certain length. `rep(1,12)` creates a vector of length 12 with each element being 1. We use the `nrow` and `ncol` arguments in the function to specify the number of rows and columns, respectfully.

[3]The `rbinom()` function generates binomial random variables and stores them in a vector. `rbinom(12,1,5)` This creates 12 random binomial numbers with parameter $n = 1$ and $p = 0.5$.

Your matrix may look different, but that is to be expected. Notice that some elements in a matrix are 0's and some are 1's. Each element in a matrix can hold any value.

An alternate approach to creating matrices is with the use of `rbind()` and `cbind()` functions. Using 2 vectors, and matrices, of the same length, the `rbind()` will append the vectors together by each row. Similarly, the `cbind()` function will append vectors, and matrices, of the same length by columns.

```
x <- 1:4
y <- 5:8
z <- 9:12
cbind(x, y, z)
```

```
     x y  z
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
```

```
rbind(x, y, z)
```

```
  [,1] [,2] [,3] [,4]
x    1    2    3    4
y    5    6    7    8
z    9   10   11   12
```

If you want to create a matrix of a specific size without any data, you can use the `matrix()` function and only specify the `nrow` and `ncol` arguments. Here we are creating a 5×11 empty matrix:

```
matrix(nrow = 5, ncol = 11)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
[1,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA
[2,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA
[3,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA
[4,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA
[5,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA
```

Lastly, if you need to find out the dimensions of a matrix, you can use `dim()` function on a matrix:

```
dim(matrix(nrow = 5, ncol = 11))
```

```
[1]  5 11
```

This will return a vector of length 2 with the first element being the number of rows and the second element being the number of columns.

1.5.4 Arrays

Matrices can be considered as a 2-dimensional block of numbers. An array is an n-dimensional block of numbers. While you may never need to use an array for data analysis. It may come in handy when programming by hand. To create an array, use the `array()` function. Below is an example of a $3 \times 3 \times 3$ with the numbers 1, 2, and 3 representing the 3rd dimension stored in an R object called `first_array`[4].

```
(first_array <- array(c(rep(1, 9), rep(2, 9), rep(3, 9)),
                      dim=c(3,3,3)))
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1

, , 2

     [,1] [,2] [,3]
[1,]    2    2    2
[2,]    2    2    2
[3,]    2    2    2

, , 3
```

[4]Notice the code is surrounded by parenthesis. This tells R to store the array and print out the results. You can surround code with parenthesis every time you create an object to also print what is stored.

```
     [,1] [,2] [,3]
[1,]    3    3    3
[2,]    3    3    3
[3,]    3    3    3
```

1.5.5 Data Frames

Data frames are similar to data set that you may encounter in an excel file. However, there are a couple of differences. First, each row represents an observation, and each column represents a characteristic of the observation. Additionally, each column in a data frame will be the same data type. To get an idea of what a data frame looks like, try `head(iris)` [5]:

```
head(iris)
```

```
  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
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

In the data frame, the rows indicate a specific observation and the columns are the values of a variable. In terms of the `iris` data set, we can see that row 1 is a specific flower that has a sepal length of 5.1. We can also see that flower 1 has other characteristics such as sepal width and petal length. Lastly, there are results for the other flowers.

Now try `tail(iris)`:

```
tail(iris)
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
145          6.7         3.3          5.7         2.5 virginica
146          6.7         3.0          5.2         2.3 virginica
147          6.3         2.5          5.0         1.9 virginica
148          6.5         3.0          5.2         2.0 virginica
149          6.2         3.4          5.4         2.3 virginica
150          5.9         3.0          5.1         1.8 virginica
```

[5]The `head()` function just tells R to only print the top few components of the data frame.

The `tail()` function provides the last 6 rows of the data frame.

Lastly, if you are interested in viewing a specific variable (column) from a data frame, you can use the `$` operator to specify which variable from a specific data frame. For example, if we are interested in observing the `Sepal.Length` variable from the `iris` data frame, we will type `iris$Sepal.Length`:

```
iris$Sepal.Length
```

```
  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
 [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
 [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
 [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
 [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
 [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
[109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
[127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
[145] 6.7 6.7 6.3 6.5 6.2 5.9
```

1.5.6 Lists

To me a list is just a container that you can store practically anything. It is compiled of elements, where each element contains an R object. For example, the first element of a list may contain a data frame, the second element may contain a vector, and the third element may contain another list. It is just a way to store things.

To create a list, use the `list()` function. Create a list compiled of first element with the mtcars data set, second element with a vector of zeros of size 4, and a matrix 3×3 identity matrix[6]. Store the list in an object called `list_one`:

```
list_one <- list(mtcars, rep(0, 4),
                 diag(rep(1, 3)))
```

Type `list_one` to see what pops out:

```
list_one
```

[6]An identity matrix is a matrix where the diagonal elements are 1 and the non-diagonal elements are 0

```
[[1]]
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2

[[2]]
[1] 0 0 0 0

[[3]]
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Each element in the list is labeled as a number. It is more useful to have the elements named. An element is named by typing the name in quotes followed by the = symbol before your object in the `list()` function (`mtcars=mtcars`).

```
list_one <- list(mtcars = mtcars,
                 vector = rep(0, 4),
                 identity = diag(rep(1, 3)))
```

Here I am creating an object called `list_one`, where the first element is `mtcars` labeled `mtcars`, the second element is a vector of zeros labeled `vector` and the last element is the identity matrix labeled `identity`.'

Now create a new list called `list_two` and store `list_one` labeled as `list_one` and `first_array` labeled as `array`.

```
(list_two <- list(list_one = list_one,
                  array = first_array))
```

```
$list_one
$list_one$mtcars
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
```

```
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2

$list_one$vector
[1] 0 0 0 0

$list_one$identity
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1


$array
, , 1

     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1

, , 2

     [,1] [,2] [,3]
[1,]    2    2    2
[2,]    2    2    2
[3,]    2    2    2

, , 3

     [,1] [,2] [,3]
[1,]    3    3    3
[2,]    3    3    3
[3,]    3    3    3
```

1.6 R Packages

As I stated before, R can be extended to do more things, such as create this tutorial. This is done by installing R packages. An R package can be thought of as extra software. This allows you to do more with R. To install an R package, you will need to use `install.packages("NAME_OF_PACKAGE")`. Once you install it, you do not need to install it again. To use the R package, use `library("NAME_OF_PACKAGE")`. This allows you to load the package in R. You will need to load the package every time you start R. For more information, please watch the video: https://vimeo.com/203516241.

2 Control Flow

2.0.1 Vectors

In the Section 1.5, we discussed about different types of R objects. For example, a vector can be a certain data type with a set number of elements. Here we construct a vector called `x` increasing from -5 to 5 by one unit:

```
(x <- -5:5)
```

```
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

The vector `x` has 11 elements. If I want to know what the 6th element of `x`, I can index the 6th element from a vector. To do this, we use `[]` square brackets on `x` to index it. For example, we index the 6th element of `x`:

```
x[6]
```

```
[1] 0
```

When ever we use `[]` next to an R object, it will print out the data to a specific value inside the square brackets. We can index an R object with multiple values:

```
x[1:3]
```

```
[1] -5 -4 -3
```

```
x[c(3,9)]
```

```
[1] -3  3
```

Notice how the second line uses the `c()`. This is necessary when we want to specify non-contiguous elements. Now let's see how we can index a matrix

2.0.2 Matrices

A matrix can be indexed the same way as a vector using the `[]` brackets. However, since the matrix is a 2-dimensional objects, we will need to include a comma to represent the different dimensions: `[,]`. The first element indexes the row and the second element indexes the columns. To begin, we create the following 4×3 matrix:

```
(x <- matrix(1:12, nrow = 4, ncol = 3))
```

```
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Now to index the element at row 2 and column 3, use `x[2, 3]`:

```
x[2, 3]
```

```
[1] 10
```

We can also index a specific row and column:

```
x[2,]
```

```
[1]  2  6 10
```

```
x[,3]
```

```
[1]  9 10 11 12
```

2.0.3 Data Frames

There are several ways to index a data frame, since it is in a matrix format, you can index it the same way as a matrix. Here are a couple of examples using the `mtcars` data frame.

```r
mtcars[,2]
```

```
 [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

```r
mtcars[2,]
```

```
              mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

However, a data frame has labeled components, variables, we can index the data frame with the variable names within the brackets:

```r
mtcars[, "cyl"]
```

```
 [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

Lastly, a data frame can be indexed to a specific variable using the `$` notation as described in Section 1.5.5.

2.0.4 Lists

As described in Section 1.5.6, lists contain elements holding different R objects. To index a specific element of a list, you will use `[[]]` double brackets. Below is a toy list:

```r
toy_list <- list(mtcars = mtcars,
                 vector = rep(0, 4),
                 identity = diag(rep(1, 3)))
```

To access the second element, vector element, you can type `toy_list[[2]]`

```r
toy_list[[2]]
```

```
[1] 0 0 0 0
```

Since the elements are labeled within the list, you can place the label in quotes inside `[[]]`:

```
toy_list[["vector"]]
```

```
[1] 0 0 0 0
```

The element can be accessed using the `$` notation with a list:

```
toy_list$vector
```

```
[1] 0 0 0 0
```

Lastly, you can further index the list if needed, we can access the `mpg` variable in `mtcars` from the `toy_list`:

```
toy_list$mtcars$mpg
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

```
toy_list[["mtcars"]]$mpg
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

```
toy_list$mtcars[,'mpg']
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

2.1 If/Else Statements

In R, there are control flow functions that will dictate how a program will be executed. The first set of functions we will talk about are `if` and `else` statements. First, the `if` statement will evaluate a task, If the conditions is satisfied, yields `TRUE`, then it will conduct a certain task, if it fails, yields `FALSE`, the `else` statement will guide it to a different task. Below is a general format:

Important Concept

```
if (condition) {
  TRUE task
} else {
  FALSE task
}
```

2.1.1 Example

Below is an example where we generate `x` from a standard normal distribution and print the statement 'positive' or 'non-positive' based on the condition `x > 0`.

```r
x <- rnorm(1)

## if statements
if (x > 0){
  print("Positive")
} else {
  print("Non-Positive")
}
```

```
[1] "Non-Positive"
```

What if we want to print the statement 'negative' as well if the value is negative? We will then need to add another `if` statement after the `else` statement since `x > 0` only lets us know if the value is positive.

```r
x <- rnorm(1)

if (x > 0){
  print("Positive")
} else if (x < 0) {
  print("Negative")
}
```

```
[1] "Positive"
```

Above, we add the `if` statement with condition `(x < 0)` indicating if the number is negative. Lastly, if `x` is ever 0, we will want R to let us know it is 0. We can achieve this by adding one last `else` statement:

```
x <- rnorm(1)

if (x > 0){
  print("Positive")
} else if (x < 0) {
  print("Negative")
} else {
  print("Zero")
}
```

```
[1] "Negative"
```

2.2 `for` loops

A for `loop` is a way to repeat a task a certain amount of times. Every time a loop repeats a task, we state it is an iteration of the loop. For each iteration, we may change the inputs by a certain way, either from an indexed vector, and repeat the task. The general anatomy of a loop looks like:

Important Concept

```
for (i in vector){
  perform task
}
```

The `for` statement indicates that you will repeat a task inside the brackets. The `i` in the parenthesis controls how the task will be completed. The `in` statement tells R where `i` can look for the values, and `vectorr` is a vector R object that contains the values `i` can be. It also controls how many times the task will be repeated based on the length of the vector.

Learning about a loop is quite challenging, my recommendation is to read the section below and break the example code so you can understand how a `for` loop works.

2.2.1 Basic `for` loop

Let's say we want R to print one to five separately. We can achieve this by repeating the `print()` 5 times.

```r
print(1); print(2); print(3); print(4); print(5)
```

```
[1] 1
```

```
[1] 2
```

```
[1] 3
```

```
[1] 4
```

```
[1] 5
```

However, this takes quite awhile to type up. Let's try to achieve the same task using a `for` loop.

```r
for (i in 1:5){
  print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Here, `i` will take a value from the vector `1:5`,[1] Then, R will print out what the value of `i` is.

Now, let's try another example with letters. To begin, create a new vector called `letters_10` containing the first 10 letters of the alphabet. Use the vector `letters` to construct the neww vector.

```
letters_10 <- letters[1:10]
```

Now, we will use a loop to print out the first 10 letters:

```
for (i in 1:10) {
  print(letters_10[i])
}
```

```
[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] "e"
[1] "f"
[1] "g"
[1] "h"
[1] "i"
[1] "j"
```

Here, we have `i` take on the values 1 through 10. Using those values, we will index the vector `letters_10` by `i`. The resulting letter will then be printed. This task repeated 10 times.

Lastly, we can replace `1:10` by `letters_10` instead:

```
for (i in letters_10){
  print(i)
}
```

```
[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] "e"
[1] "f"
```

[1]Type this in the console to see what it is.

```
[1] "g"
[1] "h"
[1] "i"
[1] "j"
```

This is because `letters_10` are the values that we want to print and `i` takes on the value of `letters_10` each time.

2.2.2 Nested `for` loops

A nested `for` loop is a loop that contain a loop within. Below is an example of 3 `for` loops nested within each other. Below is a general example:

Important Concept

```
for (i in vector_1) {
  for (ii in vector_2) {
    for (iii in vector_3) {
      perform task
    }
  }
}
```

As an example, we will use the `greekLetter::`[2] and use the `greek_vector` vector to obtain greek letters in R. Lastly, create a vector called `greek_10`.

```
library(greekLetters)
greek_10 <- greek_vector[1:10]
```

For this example, we want R to print “a” and “α” together as demonstrated below[3]:

```
print(paste0(letters_10[1], greek_10[1]))
```

```
[1] "a "
```

Now let’s repeat this process to print all possible combinations of the first 3 letters and 3 greek letters:

[2] `install.packages(greekLetters)`
[3] We will need to use `paste0()` to combine the letters together.

```
for (i in 1:3){
  for (ii in 1:3){
    print(paste0(letters_10[i], greek_10[ii]))
  }
}
```

```
[1] "a "
[1] "a "
[1] "a "
[1] "b "
[1] "b "
[1] "b "
[1] "c "
[1] "c "
[1] "c "
```

2.3 `break`

A `break` statement is used to stop a loop midway if a certain condition is met. A general setup of `break` statement goes as follows:

Important Concept

```
for (i in vector){
  if (condition) {break}
  else {
    task
  }
}
```

As you can see there is an `if` statement in the loop. This is used to tell R when to break the loop. If the `if` statement was not there, then the loop will break without iterating.

To demonstrate the break statement, we will simulate from a $N(1,1)$ until we have 30 positive numbers or we simulate a negative number.

```
x <- rep(NA,length = 30)

for (i in seq_along(x)){
  y <- rnorm(1,1)
  if (y<0) {
    break
  }
  else {
    x[i] <- y
  }
}
print(x)
```

```
 [1] 0.2822247         NA         NA         NA         NA         NA         NA
 [8]        NA         NA         NA         NA         NA         NA         NA
[15]        NA         NA         NA         NA         NA         NA         NA
[22]        NA         NA         NA         NA         NA         NA         NA
[29]        NA         NA
```

```
print(y)
```

```
[1] -0.06732581
```

Notice that the vector does not get filled up all the way, that is because the loop will break once a negative number is simulated

2.4 `next`

Similar to the `break` statement, the `next` statement is used in loops that will tell R to move on to the next iteration if a certain condition is met.

Important Note

```
for (i in vector){
  if (condition) {
    next
  } else {
    task
  }
}
```

The main difference here is that a **`next`** statement is used instead of a **`break`** statement.

Going back to simulating positive numbers, we will use the same setup but change it to a **`next`** statement.

```r
x <- rep(NA,length = 30)

for (i in seq_along(x)){
  y <- rnorm(1,1)
  if (y<0) {
    next
  }
  else {
    x[i] <- y
  }
}
print(x)
```

```
 [1] 1.7209191 0.1896949        NA 1.0792022        NA 2.1369064 0.6085387
 [8]        NA 0.5623279 2.4101566 3.2359326 1.4701157 0.6353537 2.6892720
[15]        NA 0.2331778 3.0216481        NA 1.6867428 1.0063384 0.6367926
[22]        NA 3.4886272 1.3408562 0.3545091 1.5495891 0.8707791        NA
[29] 1.5991112 1.3069543
```

As you can see, the vector contains missing values, these were the iterations that a negative number was simulated.

2.5 `while` loop

The last loop that we will discuss is a while loop. The while loop is used to keep a loop running until a certain condition is met. To construct a while loop, we will use the **`while`** statement with a condition attached to it. In general, a while loop will have the following format:

Important Concept

```
while (condition) {
  task
  update condition
}
```

Above, we see that the `while` statement is used followed by a condition. Then the loop will complete its task and update the condition. If the condition yields a `FALSE` value, then the loop will stop. Otherwise, it will continue.

2.5.1 Basic `while` loops

To implement a basic `while` loop, we will work on the previous example of simulating positive numbers. We want to simulate 30 positive numbers from $N(0,1)$ until we have 30 values. Here, our condition is that we need to have 30 numbers. Therefore we can use the following code to simulate the values:

```
x <- c()
size <- 0
while (size < 30){
  y <- rnorm(1)
  if (y > 0) {
    x <- c(x, y)
  }
  size <- length(x)
}
print(size)
```

```
[1] 30
```

```
print(x)
```

```
 [1] 0.27075614 0.68213351 0.64117300 0.09325178 0.25511193 0.84847289
 [7] 0.99696727 0.49154805 1.12825620 0.03624028 0.64491023 1.61245622
[13] 0.46394449 0.05552212 0.39188109 0.50643163 0.47071310 1.19085171
[19] 0.02597452 1.33588515 0.24634318 0.28013134 0.04718407 1.46137496
[25] 0.85088606 0.31027703 1.06482412 0.28022502 1.31905554 0.28745050
```

Notice that we do not use an `else` statement. This is because we do not need R to complete a task if the condition fails.

2.5.2 Infinite `while` loops

With while loops, we must be weary about potential infinite loops. This occurs when the condition will never yield a `FALSE` value. Therfore, R will never stop the loop because it does not know when to do this.

For example, let's say we are interest if $y = sin(x)$ will converge to a certain value. As you know it will not converge to a certain value; however, we can construct a while loop:

```r
x <- 1
diff <- 1
while (diff > 1e-20) {
  old_x <- x
  x <- x + 1
  diff <- abs(sin(x) - sin(old_x))
}
print(x)
print(diff)
```

My condition above is to see if the absolute difference between sequential values is smaller than 10^{-20}. As you may know, the absolute difference will never become that small. Therefore, the loop will continue on without stopping.

To prevent an infinite while loop, we can add a counter to the condition statement. This counter will also need to be true for the loop to continue. Therefore, we can arbitrarily stop it when the loop has iterated a certain amount of times. We just need to make sure to add one to the counter every time it iterates it. Below is the code that adds a counter to the while loop:

```r
x <- 1
counter <- 0
diff <- 1
while (diff > 1e-20 & counter < 10^3) {
  old_x <- x
  x <- x + 1
  diff <- abs(sin(x) - sin(old_x))
  counter <- counter + 1
}
print(x)
```

```
[1] 1001
```

```r
print(diff)
```

```
[1] 0.09311106
```

```r
print(counter)
```

```
[1] 1000
```

3 Functional Programming

3.1 Functions

The functionality in R is what makes it completely powerful compared to other statistical software. There are several pre-built functions, and you can extend R's functionality further with the use of R Packages.

3.1.1 Built-in Functions

There are several available functions in R to conduct specific statistical methods. The table below provides a set of commonly used functions:

Functions	Description
`aov()`	Fits an ANOVA Model
`lm()`	Fits a linear model
`glm()`	Fits a general linear model
`t.test()`	Conducts a t-test

Several of these functions have help documentation that provide the following sections:

Section	Description
Description	Provides a brief introduction of the function
Usage	Provides potential usage of the function
Arguments	Arguments that the function can take
Details	An in depth description of the function
Value	Provides information of the output produced by the function
Notes	Any need to know information about the function
Authors	Developers of the function
References	References to the model and function
See Also	Provide information of supporting functions
Examples	Examples of the function

To obtain the help documentation of each function, use the `?` operator and function name in the console pane.

3.1.2 Generic Functions

Commonly used functions, such as `summary()` and `plot()` functions, are considered generic functions where their functionality is determined by the class of an R object. For example, the `summary()` function is a generic function for several types of functions: `summary.aov()`, `summary.lm()`, `summary.glm()`, and many more. Therefore, the appropriate function is needed depending the type of R object. This is where generic functions come in. We can use a generic function, ie `summary()`, to read the type of object and then apply to correct procedure to the object.

3.1.3 User-built Functions

While R has many capable functions that can be used to analyze your data, you may need to create a custom function for specific needs. For example, if you find yourself writing the same to repeat a task, you can wrap the code into a user-built function and use it for analysis.

To create a user-built function, you will using the `function()` to create an R object that is a function. To use the function Inside the `funtion()` parentheses, write the arguments that need to specified for your function. These are arguments you choose for the function.

3.1.3.1 Anatomy

In general function we construct a function with the following anatomy:

```r
name_of_function <- function(data_1, data_2 = NULL,
                             argument_1, argument_2 = TRUE, argument_3 = NULL,
                             ...){
  # Conduct Task
  # Conduct Task
  output_object <- Tasks
  return(output_object)
}
```

Here, we are creating an R function called `name_of_function` that will take the following arguments: `data_1`, `data_2`, `argument_1`, `argument_2`, `argument_3`, and `...`. From this function, it requires us to supply data for `data_1` and `argument_1`. Arguments `data_2` and `argument_3` are not required, but can be utilized in the function if necessary. Argument `argument_2` is also required for the function, but it it has a default setting (in this case `TRUE`) if it is not specified. Lastly, the `...` argument allows you to pass other arguments to R built in functions if they are present. For example, we may use the `plot()` to create graphics and want to manipulate the output plot further, but do not want to specify the arguments in the user-based function. In the function itself, we will complete the necessary tasks and then use the `return()` to return the output.

3.1.3.2 Example

To begin, let's create a function that squares any value:

```
x_square <- function(x){x^2}
```

Above, I am creating a new function called `x_square` and it will take values of `x` and square it. Here are a couple of examples of `x_square()`:

```
x_square(4)
```

```
[1] 16
```

```
x_square(5)
```

```
[1] 25
```

The `mtcars` data set has several numeric variables that can be used for analysis. Let's say we want to apply a function (`x_square()`) to the sum of a specific variable and return the value. Then let's further complicate the function by allowing the sum of 2 variables, take the log of the sum and dividing the value if necessary. Below is the code for such function called `summing`:

```
summing <- function(vec1, vec2 = NULL, FUN, log_val = FALSE, divisor_val = NULL){
  FUN <- match.fun(FUN)
  wk_vec <- c(vec1, vec2)
  fun_sum_val <- FUN(sum(wk_vec))
  lval <- NULL
  if (isTRUE(log_val)){
    lval <- log(fun_sum_val)
  } else {
    lval <- fun_sum_val
  }
```

```r
  if (!is.null(divisor_val)){
    dval <- divisor_val
  } else {dval <- 1}
  output <- lval/dval
  return(output)
}
```

Now let's try obtaining the

```r
sum(mtcars$mpg)^2
```

```
[1] 413320.4
```

```r
summing(mtcars$mpg, FUN = x_square)
```

```
[1] 413320.4
```

```r
log(sum(c(mtcars$mpg,mtcars$disp))^2)
```

```
[1] 17.98088
```

```r
summing(mtcars$mpg, mtcars$disp, x_square, T)
```

```
[1] 17.98088
```

```r
log(sum(c(mtcars$mpg,mtcars$disp))^2)/5
```

```
[1] 3.596177
```

```r
summing(mtcars$mpg, mtcars$disp, x_square, T, 5)
```

```
[1] 3.596177
```

3.2 *apply Functions

*apply functions are used to iterate a function through a set of elements in a vector, matrix, or list. This will then return a vector or list depending on what is requested.

3.2.1 `apply()`

The `apply()` function is used to apply a function to the margins of an array or matrix. It will iterate between the elements, apply a function to the data, and return a vector, array or list if necessary. To use the `apply()` function, you will need to specify three arguments, `X` or the array, `MARGIN` which margin to apply the function on, and `FUN` the function.

Below we calculate the row means and column means using the apply function for a 5×4 matrix containing the elements 1 through 20:

```r
x <- matrix(1:20, nrow = 5, ncol = 4)

# Row Means
apply(x, 1, mean)
```

```
[1]  8.5  9.5 10.5 11.5 12.5
```

```r
# Col Means
apply(x, 2, mean)
```

```
[1]  3  8 13 18
```

3.2.2 `lapply()`

The `lapply()` function is used to apply a function to all elements in a vector or list. The `lapply()` function will then return a list as the output.

3.2.3 `sapply()`

The `sapply()` function is used to apply a function to all elements in a vector or list. Afterwards, the `sapply()` will return a "simplified" version of the list format. This could be a vector, matrix, or array.

3.3 Anonymous Functions

Anonymous functions are functions that R temporarily creates to conduct a task. They are commonly used in the *apply functions, piping or within functions. To create an anonymous function, we use the `function()` to create a function .

For example, let `x` be a vector with the values 1 through 15. Let's say we want to apply the function $f(x) = x^2 + \ln(x) + e^x/x!$. We can evaluate the function as the expression in the function:

```
x <- 1:15
x^2 + log(x) + exp(x)/factorial(x)
```

```
 [1]   3.718282   8.387675  13.446202  19.661217  27.846214  38.352077
 [7]  51.163496  66.153374  83.219555 102.308655 123.399395 146.485246
[13] 171.565020 198.639071 227.708053
```

Let's say we could not do that, we need to evaluate the function for each value of `x`. We can use the `sapply()` function with an anonymous function:

```
sapply(x, function(x) x^2 + log(x) + exp(x) / factorial(x))
```

```
 [1]   3.718282   8.387675  13.446202  19.661217  27.846214  38.352077
 [7]  51.163496  66.153374  83.219555 102.308655 123.399395 146.485246
[13] 171.565020 198.639071 227.708053
```

In R 4.1.0, developers introduce a shortcut approach to create functions. You can create a function using `\()` expression, and specify the arguments for your function within the parenthesis. Reworking the previous code, we can use `\()` instead of `function()`:

```
sapply(x, \(x) x^2 + log(x) + exp(x)/factorial(x))
```

```
 [1]   3.718282   8.387675  13.446202  19.661217  27.846214  38.352077
 [7]  51.163496  66.153374  83.219555 102.308655 123.399395 146.485246
[13] 171.565020 198.639071 227.708053
```

```
sapply(x, \(.) .^2 + log(.) + exp(.)/factorial(.))
```

```
 [1]   3.718282   8.387675  13.446202  19.661217  27.846214  38.352077
 [7]  51.163496  66.153374  83.219555 102.308655 123.399395 146.485246
[13] 171.565020 198.639071 227.708053
```

Notice that the argument in the anonymous function can be anything.

4 Scripting and Piping in R

4.1 Commenting

A comment is used to describe your code within an R Script. To comment your code in R, you will use the # key, and R will not execute any code after the symbol. The # key can be used to anywhere in the line, from beginning to midway. It will not execute any code coming after the #.

Additionally, commenting is a great way to debug long scripts of code or functions. You comment certain lines to see if any errors are being produced. It can be used to test code line by line with out having to delete everything.

4.2 Scripting

When writing a script, it is important to follow a basic structure for you to follow your code. While this structure can be anything, the following sections below has my main recommendations for writing a script. The most important part is the **Beginning of the Script** section.

4.2.1 Beginning of the Script

Load any R packages, functions/scripts, and data that you will need for the analysis. I always like to get the date and time of the

```r
## Todays data
analysis_data <- format(Sys.time(),"%Y-%m-%d-%H-%M")

## R Packages
library(tidyverse)
library(magrittr)

## Functions
source("fxs.R")
```

```r
Rcpp::sourceCpp("fxs.cpp")

## Data
df1 <- read_csv("file.csv")
df2 <- load("file.RData") %>% get
```

4.2.2 Middle of the Script

Run the analysis, including pre and post analysis.

```r
## Pre Analysis
df1_prep <- Prep_data(df1)
df2_prep <- Prep_data(df2)

## Analysis
df1_analysis <- analyze(df1_prep)
df2_analysis <- analyze(df2_prep)

## Post Analysis
df1_post <- Prep_post(df1_anlysis)
df2_post <- Prep_post(df2_anlysis)
```

4.2.3 End of the Script

Save your results in an R Data file:

```r
## Save Results
res <- list(df1 = list(pre = df1_prep,
                       analysis = df1_analysis,
                       post = df1_post),
            df2 = list(pre = df2_prep,
                       analysis = df2_analysis,
                       post = df2_post))
file_name <- paste0("results_", analysis_data, ".RData")
save(res, file = file_name)
```

4.3 Pipes

In R, pipes are used to transfer the output from one function to the input of another function. Piping will then allow you to chain functions to run an analysis. Since R 4.1.0, there are two version of pipes, the base R pipe and the pipes from the magrittr package. The table below provides a brief description of each type pipes

Pipe	Name	Package	Description
`|>`	R Pipe	Base	This pipe will use the output of the previous function as the input for the first argument following function.
`%>%`	Forward Pipe	magrittr	The forward pipe will use the output of the previous function as the input of the following function.
`%$5`	Exposition Pipe	magrittr	The exposition function will expose the named elements of an R object (or output) to the following function.
`%T>%`	Tee Pipe	magrittr	The Tee pipe will evaluate the next function using the output of the previous function, but it will not retain the output of the next function and utilize the output of the previous function.
`%<>%`	Assignment Pipe	magrittr	The assignment pipe will rewrite the object that is being piped into the next function.

When choosing between Base or magrittr's pipes, I recommend using magrittr's pipes due to the extended functionality. However, when writing production code or developing an R package, I recommend using the Base R pipe.

Lastly, when using the pipe, I recommend only stringing a limited amount of functions (~10) to maintain code readability and conciseness. Any more functions may make the code incoherent.

If you plan to use magrittr's pipe, I recommend loading `magrittr::` package instead of `tidyverse::` package.

```r
library(magrittr)
```

4.3.1 |>

The base pipe will use the output from the first function and use it as the input of the first argument in the second function. Below, we obtain the `mpg` variable from `mtcars` and pipe it in the `mean()` function.

```
mtcars$mpg |> mean()
```

```
[1] 20.09062
```

4.3.2 %>%

4.3.2.1 Uses

Magrittr's pipe is the equivalent of Base R's pipe, with some extra functionality. Below we repeat the same code as before:

```
mtcars$mpg %>% mean()
```

```
[1] 20.09062
```

Alternatively, we do not have to type the parenthesis in the second function:

```
mtcars$mpg %>% mean
```

```
[1] 20.09062
```

Below is another example where we will pipe the value `3` into the `rep()` with `times=5`, this will repeat the value `3` five times:

```
3 %>% rep(5)
```

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

If we are interested in piping the output to another argument other than the first argument, we can use the `(.)` placeholder in the second function to indicate which argument should take the previous output. Below, we repeat the vector `c(1, 2)` three times because the `.` is in the second argument:

```
3 %>% rep(c(1,2), .)
```

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

4.3.2.2 Creating Unary Functions

You can use `%>%` and `.` to create unary functions, a function with one argument, can be created. The following code will create a new function called `logsqrt()` which evaluates $\sqrt{\log(x)}$:

```
logsqrt <- . %>% log(base = 10) %>% sqrt
logsqrt(10000)
```

```
[1] 2
```

```
sqrt(log10(10000))
```

```
[1] 2
```

4.3.3 `%$%`

The exposition pipe will expose the named elements of an object or output to the following function. For example, we will pipe the `mtcars` into the `lm()` function. However, we will use the `%$%` pipe to access the variables in the data frame for the `formula=` argument without having to specify the `data=` argument:

```
mtcars %$% lm(mpg ~ hp)
```

```


Call:
lm(formula = mpg ~ hp)

Coefficients:
(Intercept)           hp
   30.09886     -0.06823
```

4.3.4 %T>%

The Tee pipe will pipe the contents of the previous function into the following function, but will retain the previous functions output instead of the current function. For example, we use the Tee pipe to push the results from the `lm()` function to print out the summary table, then use the same `lm()` function results to print out the model standard error:

```
x_lm <- mtcars %$% lm(mpg ~ hp) %T>%
  (\(x) print(summary(x))) %T>%
  (\(x) print(sigma(x)))
```

```

Call:
lm(formula = mpg ~ hp)

Residuals:
    Min      1Q  Median      3Q     Max 
-5.7121 -2.1122 -0.8854  1.5819  8.2360 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 30.09886    1.63392  18.421  < 2e-16 ***
hp          -0.06823    0.01012  -6.742 1.79e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.863 on 30 degrees of freedom
Multiple R-squared:  0.6024,	Adjusted R-squared:  0.5892 
F-statistic: 45.46 on 1 and 30 DF,  p-value: 1.788e-07

[1] 3.862962
```

4.4 Keyboard Shortcuts

Below is a list of recommended keyboard shortcuts:

Shortcut	Windows/Linux	Mac
%>%	Ctrl+Shift+M	Cmd+Shift+M
Run Current Line	Ctrl+Enter	Cmd+Return
Run Current Chunk	Ctrl+Shift+Enter	Cmd+Shift+Enter

Shortcut	Windows/Linux	Mac
Knit Document	Ctrl+Shift+K	Cmd+Shift+K
Add Cursor Below	Ctrl+Alt+Down	Cmd+Alt+Down
Comment Line	Ctrl+Shift+C	Cmd+Shift+C

I recommend modify these keyboard shortcuts in RStudio

Shortcut	Windows/Linux	Mac
`%in%`	Ctrl+Shift+I	Cmd+Shift+I
`%$%`	Ctrl+Shift+D	Cmd+Shift+D
`%T>%`	Ctrl+Shift+T	Cmd+Shift+T

Note you will need to install the `extraInserts` package:

```r
remotes::install_github('konradzdeb/extraInserts')
```

Part II

Simulations

5 Random Variables

6 Models

6.1 Normal Model

6.2 Binomial Model

6.3 Poisson Model

6.4 Gamma Model

6.5 Survival Model

Part III

Randomizations

7 Permutation Tests

8 Permutation Regression

Part IV

Monte Carlo Methods

9 Monte Carlo Integration

10 Monte Carlo Hypothesis Testing

11 Monte Carlo Methods Case Study 1

12 Monte Carlo Methods Case Study 2

13 Monte Carlo Methods Case Study 3

Part V

Bootstrapping

14 Parametric Bootrapping

15 Nonparametric Boostrapping

Part VI

Data Manipulation, Summarization, and Graphics

Resources

How to read this section.

Through out this Section, we use certain notations for different components in R. To begin, when something is in a gray block, `_`, this indicates that R code is being used. When I am talking about an R Object, it will be displayed as a word. For example, we will be using the R object `mtcars`. When I am talking about an R function, it will be displayed as a word followed by an open and close parentheses. For example, we will use the mean function denoted as `mean()` (read this as "mean function"). When I am talking about an R argument for a function, it will be displayed as a word following by an equal sign. For example, we will use the data argument denoted as `data=` (read this as "data argument"). When I am referencing an R package, I will use `::` (two colons) after the name. For example, in this Section, I will use the `ggplot2::` (read this as "ggplot2 package") Lastly, if I am displaying R code for your reference or to run, it will be displayed on its own line. There are many components in R, and my hope is that this will help you understand what components am I talking about.

16 Importing Data

16.1 Directories

16.2 Importing Data

```
# Reading Data -----

## RData ----
load("~/x.RData")

## CSV ----
library(readr)
data_3_1_csv <- read_csv("student/stat_147/data/data_3_1.csv")
View(data_3_1_csv)

## Excel ----
library(readxl)
data_3_1 <- read_excel("student/stat_147/data/data_3_1.xlsx")
View(data_3_1)

## txt ----
library(readr)
data_3_1_s <- read_table2("student/stat_147/data/data_3_1_s.txt")
View(data_3_1_s)

## Semi-colon ----
library(readr)
data_3_1_sc <- read_delim("student/stat_147/data/data_3_1_sc.txt", ";", escape_double = FA
View(data_3_1_sc)
## SPSS ----
library(haven)
data_3_1 <- read_sav("student/stat_147/data/data_3_1.sav")
View(data_3_1)
```

```r
## SAS -----
library(haven)
data_3_1 <- read_sas("student/stat_147/data/data_3_1.sas7bdat", NULL)
View(data_3_1)

## Stata ----
library(haven)
data_3_1 <- read_dta("student/stat_147/data/data_3_1.dta")
View(data_3_1)

data_3_1 <- read.csv("~/student/stat_147/data/data_3_1.csv", header=FALSE)
View(data_3_1)


# Reading Data -----
setwd("~/Repos/s147/files/Week_2")

## Base R -----

# CSV
data.csv <- read.csv("data.csv")

# STATA File
library(foreign)
read.dta("data.dta")

## RStudio packages
library(readr)
read_csv("data.csv")

library(readxl)
read_excel("data.xlsx")

library(haven)
read_dta("data.dta")
```

17 Data Manipulation

17.1 Introduction

Data manipulation consists of transforming a data set to be analyzed. Certain statistical methods require data sets to be formatted in a certain way before you can apply a certain function[1]. Other times, you will need to transform the data set to present to stakeholder. Therefore, being able to transform a data set is essential.

17.1.1 Notes

Warnings Suppressed

In order to keep the page concise, the warning messages have been suppressed. These warnings were produced because functions were applied to incorrect inputs, ie $\sqrt{A}$. Therefore, you may see `NA` as the output. There is nothing wrong with the code, it is just that the input was not valid, but R still completed the task.

17.2 Tidyverse

Tidyverse is a set of packages that make data manipulation much easier. These are functions that many individuals from the R community find useful to use for data analysis. In my opinion, once you have understand how Tidyverse packages function, it makes it much easier to work with than Base R. Many of the functions are descriptively named for easy remembrance. For the most part, you can do almost everything that Base R can do. There are just a few things you can't do, but it is rare that you will use them. One last thing is that the output from tidyverse is always formatted as a tibble class, the Tidyverse version of the data frame. This can have some ups and downs. However, think of a tibble as a lazier data frame. If you haven't done so, install tidyverse:

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

[1] Linear Mixed-Effects Models.

Then load tidyverse into R:

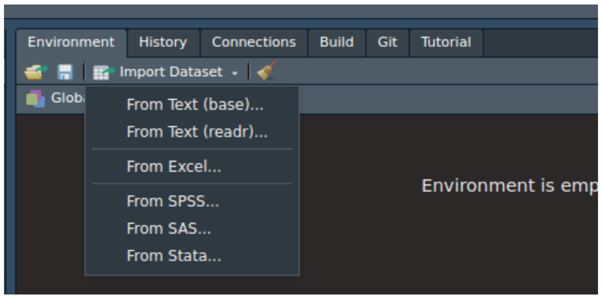
```r
library(tidyverse)
```

This will load the main Tidyverse packages: `ggplot2::`, `tibble::`, `tidyr::`, `readr::`, `purr::`, `dplyr::`, `stringr::`, and `forcats::`.

17.3 Loading Data

There are three methods to load a data set in R: using base R, using Tidyverse, or using RStudio. While it is important to understand how the code works to load a data set, I recommend using RStudio to import the data. It does all the work for you. Additionally, if you decide to use Tidyverse packages, RStudio will provide corresponding code for a particular file.

To import a data set using RStudio, head over to the environment tab (usually in the upper right hand pane) and click on the Import Dataset button. A pop-up window should look something like below.



Notice how there are several options to load a data set. Depending on the format, you may want to choose one of those options. Next, notice how there are 2 “From Text”. This is because it will load text data using either Base R packages or the `readr::` package from tidyverse. Either works, but the `readr::` package provides the necessary code to load the data set in the window. The other one provides the code in the console.

17.3.1 CSV Files

A CSV file is a type of text file that where the values are separated from commas. It is very common file that you will work with. Here I will provide the code necessary to import a CSV file using either Base R or `readr::` code.

17.3.1.1 Base R

```
read.csv("FILE_NAME_AND_LOCATION")
```

17.3.1.2 `readr::`

```
read_csv("FILE_NAME_AND_LOCATION")
```

Notice that the functions are virtually the same.

17.3.2 For This Chapter

You will need to download and extract this zip file to conduct the analysis in the chapter. The code below will load the data sets you need:

```
data1 <- read_csv("data/data_3_1.csv")
data2 <- read_csv("data/data_3_2.csv")
data3 <- read_csv("data/data_3_3.csv")
data4 <- read_csv("data/data_3_6.csv")
data5 <- read_csv("data/data_3_7.csv")
data6 <- read_csv("data/data_3_5.csv")
data7 <- read_csv("data/data_3_4.csv")
```

Make sure to change the file location as needed.

17.4 The Pipe Operator %>%

The main benefit of the pipe operator is to make the code easier to read. The pipe operator is from the `magrittr::`. It is usually loaded when you load the `tidyverse::`. What the pipe operator, `%>%`, does is that it will take the output from a previous function and it will use it as the input for the next function. This prevents us from nesting functions together and overwhelm us with numerous parentheses and commas. To practice, pipe `data` into the `glimpse()`.

```
data1 %>% glimpse()
```

```
Rows: 1,000
Columns: 10
$ ID1  <chr> "A2b6115fd", "Ac51c9cf1", "A7534d3a0", "A73fc5642", "Ae020e4bd", ~
$ cat1 <chr> "A", "A", "A", "A", "C", "A", "A", "C", "B", "C", "B", "B", "A", ~
$ cat2 <chr> "E", "D", "F", "F", "E", "D", "E", "F", "E", "E", "F", "E", "E", ~
$ var1 <dbl> 1.1541672, -0.3667030, -0.4203357, -2.0006336, 0.6970417, 0.46690~
$ var2 <dbl> 4, 3, 6, 5, 3, 4, 5, 0, 5, 3, 3, 8, 4, 5, 2, 5, 7, 7, 5, 1, 4, 3,~
$ var3 <dbl> 2.87981553, 0.06397162, -1.04021753, -0.31355281, 0.52613439, 2.4~
$ var4 <dbl> 3.53845785, 1.07279559, 0.22632480, 0.02128418, 2.97936180, 1.853~
$ var5 <dbl> -3.1827969, -3.1827969, -3.1827969, -3.1827969, 6.0601967, -3.182~
$ var6 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ var7 <dbl> 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1,~
```

The `glimpse()` provides basic variable information about `data1`. I recommend practice reading the code in plain English to help you understand how these functions.

17.5 Data Transformation

This section focuses on manipulating the data to obtain basic statistics, such as obtaining the mean for different categories. Many of the functions used here are from the `dplyr::`.

17.5.1 Summarizing Data

Summarizing Data is one of the most important thing in statistics. First, let's get the mean for all the variables in `data1`. This is done by using the `summarize_all()`. All you need to do is provide the function you want R to provide. Pipe `data1` into the `summarize_all()` and specify `mean` in the function.

```
data1 %>% summarise_all(mean)
```

```
# A tibble: 1 x 10
    ID1  cat1  cat2    var1  var2  var3  var4  var5  var6  var7
  <dbl> <dbl> <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1    NA    NA    NA -0.0335  4.94 0.936  1.96 0.774     0 0.394
```

Notice how some values are `NA`, this is because the variables are character data types. Therefore, it will not be able to compute the mean. Now find the standard deviation for the data set.

```
data1 %>% summarise_all(sd)
```

```
# A tibble: 1 x 10
    ID1  cat1  cat2  var1  var2  var3  var4  var5  var6  var7
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1    NA    NA    NA  1.00  2.22  1.02  1.00  3.93     0 0.489
```

Now lets create a frequency table for the `cat1` variable in `data1`. use the `count()` and specify the variable you are interested in:

```
data1 %>% count(cat1)
```

```
# A tibble: 3 x 2
  cat1      n
  <chr> <int>
1 A       332
2 B       328
3 C       340
```

Now, repeat for `cat2` in `data1`:

```
data1 %>% count(cat2)
```

```
# A tibble: 3 x 2
  cat2      n
  <chr> <int>
1 D       322
2 E       337
3 F       341
```

17.5.2 Grouping

Summarizing data is great, but sometimes you may want to group data and obtain summary statistics for those groups. This is done by using the `group_by()` and specify which variable you want to group. Try grouping `data1` by `cat1`:

```
data1 %>% group_by(cat1)
```

```
# A tibble: 1,000 x 10
# Groups:   cat1 [3]
   ID1       cat1  cat2    var1  var2    var3   var4   var5  var6  var7
   <chr>     <chr> <chr>  <dbl> <dbl>   <dbl>  <dbl>  <dbl> <dbl> <dbl>
 1 A2b6115fd A     E      1.15      4  2.88   3.54   -3.18      0     1
 2 Ac51c9cf1 A     D     -0.367     3  0.0640 1.07   -3.18      0     0
 3 A7534d3a0 A     F     -0.420     6 -1.04   0.226  -3.18      0     0
 4 A73fc5642 A     F     -2.00      5 -0.314  0.0213 -3.18      0     0
 5 Ae020e4bd C     E      0.697     3  0.526  2.98    6.06      0     0
 6 Ac0d3b0fe A     D      0.467     4  2.45   1.85   -3.18      0     0
 7 A2edfed41 A     E      1.36      5  0.514  0.529  -3.18      0     1
 8 Ad38a4bbe C     F      0.369     0  1.98   2.36    6.06      0     1
 9 A5ee0f97f B     E      1.80      5  0.147  2.22   -0.701     0     0
10 Ad791c03d C     E      1.25      3 -1.05   0.0289  6.06      0     1
# ... with 990 more rows
```

Great! You now have grouped data; however, this is not helpful. We can use this output and summarize the groups. All we need to do is pipe the output to the `summarise_all()`. Group `data1` by `cat1` and find the mean:

```
data1 %>% group_by(cat1) %>% summarise_all(mean)
```

```
# A tibble: 3 x 10
  cat1    ID1  cat2    var1  var2  var3  var4   var5  var6  var7
  <chr> <dbl> <dbl>   <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl> <dbl>
1 A        NA    NA -0.0369  4.79 0.877  1.97 -3.18      0 0.401
2 B        NA    NA -0.0345  4.98 1.01   1.97 -0.701     0 0.390
3 C        NA    NA -0.0292  5.05 0.922  1.94  6.06      0 0.391
```

If we want to group by two variables, all we need to do is specify both variables in the `group_by()`. Group `data1` by `cat1` and `cat2` then find the mean:

```
data1 %>% group_by(cat1,cat2) %>% summarise_all(mean)
```

```
# A tibble: 9 x 10
# Groups:   cat1 [3]
  cat1  cat2    ID1      var1  var2  var3  var4   var5  var6  var7
  <chr> <chr> <dbl>     <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl> <dbl>
1 A     D        NA -0.0186    4.87 0.858  1.93 -3.18      0 0.374
2 A     E        NA -0.0265    4.70 0.882  1.94 -3.18      0 0.393
```

```
3 A     F        NA -0.0661    4.79 0.890  2.03 -3.18      0 0.436
4 B     D        NA -0.152     5.21 1.01   1.96 -0.701     0 0.333
5 B     E        NA  0.0890    5.04 1.04   1.94 -0.701     0 0.417
6 B     F        NA -0.0659    4.72 0.979  2.03 -0.701     0 0.411
7 C     D        NA -0.160     5.01 0.927  1.87  6.06      0 0.378
8 C     E        NA -0.000228  5.07 0.910  2.00  6.06      0 0.345
9 C     F        NA  0.0662    5.07 0.930  1.94  6.06      0 0.445
```

Now, instead of finding the mean for all variables in a data set, we are only interested in viewing `var1`. We can use the `summarise()` and type the R code for finding the mean for the particular variable. Group `data1` by `cat1` and find the mean for `var1`:

```
data1 %>% group_by(cat1) %>% summarise(mean(var1))
```

```
# A tibble: 3 x 2
  cat1  `mean(var1)`
  <chr>        <dbl>
1 A          -0.0369
2 B          -0.0345
3 C          -0.0292
```

17.5.3 Subsets

On occasion, you may need to create a subset of your data. You may only want to work with one part of your data. To create a subset of your data, use the `filter()` to create the subset. This will select the rows that satisfy a certain condition. Create a subset of `data1` where only the positive values of `var1` are present. Use the `filter()` and state `var1>0`.

```
data1 %>% filter(var1>0)
```

```
# A tibble: 484 x 10
   ID1       cat1  cat2   var1  var2   var3   var4   var5  var6  var7
   <chr>     <chr> <chr> <dbl> <dbl>  <dbl>  <dbl>  <dbl> <dbl> <dbl>
 1 A2b6115fd A     E     1.15      4  2.88   3.54   -3.18      0     1
 2 Ae020e4bd C     E     0.697     3  0.526  2.98    6.06      0     0
 3 Ac0d3b0fe A     D     0.467     4  2.45   1.85   -3.18      0     0
 4 A2edfed41 A     E     1.36      5  0.514  0.529  -3.18      0     1
 5 Ad38a4bbe C     F     0.369     0  1.98   2.36    6.06      0     1
 6 A5ee0f97f B     E     1.80      5  0.147  2.22   -0.701     0     0
 7 Ad791c03d C     E     1.25      3 -1.05   0.0289  6.06      0     1
 8 Af88d3ab5 B     E     2.10       8  3.07   3.29    -0.701      0     1
 9 A429b65a6 A     E     1.46       4  0.0638 2.36    -3.18       0     0
10 A3638155a A     F     0.429      5  1.76   1.55    -3.18       0     1
# ... with 474 more rows
```

If you know which rows you want, you can use the `slice()` and specify the rows as a vector. Create a subset of `data1` and select the rows 100 to 200 and 300 to 400.

```
data1 %>% slice(c(100:200, 300:400))
```

```
# A tibble: 202 x 10
   ID1       cat1  cat2    var1  var2   var3    var4   var5  var6  var7
   <chr>     <chr> <chr>  <dbl> <dbl>  <dbl>   <dbl>  <dbl> <dbl> <dbl>
 1 A568e9a48 A     E      1.88      6  2.59  4.02    -3.18      0     1
 2 Aa77dca83 B     D      1.32      5  0.889 3.12    -0.701     0     1
 3 A901d56c2 C     F     -0.406     6  0.496 3.67     6.06      0     0
 4 Ad66ce513 A     D      1.04      2  0.331 0.00523 -3.18      0     0
 5 A897a230d B     E     -0.616     9  1.61  3.08    -0.701     0     0
 6 Afbc693a4 B     F     -0.404     7 -0.604 0.149   -0.701     0     1
 7 Ae7269323 C     F     -0.176     4  1.06  2.26     6.06      0     0
 8 A182729af A     D      1.11      7 -0.297 1.45    -3.18      0     1
 9 A1a06950b C     E     -1.29      5  1.78  3.07     6.06      0     1
10 A569c9d81 A     F      1.28      6  0.683 1.53    -3.18      0     0
# ... with 192 more rows
```

If you need random sample of your `data1`, use the `sample_n()` and specify the number you want. It will create a data set of randomly selected rows. Create a random sample of `data1` of 100 rows.

```
data1 %>%  sample_n(100)
```

```
# A tibble: 100 x 10
   ID1       cat1  cat2    var1  var2    var3    var4   var5  var6  var7
   <chr>     <chr> <chr>  <dbl> <dbl>   <dbl>   <dbl>  <dbl> <dbl> <dbl>
 1 D9f856b96 A     F     -1.49      1 -0.175   1.65   -3.18      0     1
 2 B9a648d3b A     E      0.978     7  1.38    1.17   -3.18      0     0
 3 B44e4fc90 A     E      1.40      3  1.65    3.87   -3.18      0     1
 4 C1baa25f5 B     F      0.377     5 -0.0618 -0.0663 -0.701     0     0
 5 Da3cb7424 B     D      0.457     9  1.72    3.77   -0.701     0     1
 6 D269f438f A     E     -0.851    15  0.744   2.73   -3.18      0     0
 7 Bfe326236 C     F     -0.301     6  0.173   1.56    6.06      0     0
 8 Ca77dca83 B     F      0.636     7 -0.0266  2.36   -0.701     0     0
 9 A3235290a A     F     -0.418     5  1.83    2.85   -3.18      0     1
10 Cfd7b14fe C     F     -0.972     5  1.25    3.01    6.06      0     0
# ... with 90 more rows
```

If you want a random sample that is proportion of your original data size, use the `sample_frac()`. Specify the proportion that you want from the data. Create a random sample of `data1` that is only 2/7th of the original size.

```
data1 %>% sample_frac(2/7)
```

```
# A tibble: 286 x 10
   ID1       cat1  cat2    var1  var2   var3  var4   var5  var6  var7
   <chr>     <chr> <chr>  <dbl> <dbl>  <dbl> <dbl>  <dbl> <dbl> <dbl>
 1 D7534d3a0 C     D      0.214     4  1.12  1.46   6.06      0     0
 2 C39e0959a A     E     -1.27      8  0.766 0.796 -3.18      0     0
 3 Df48d4b9d C     D     -0.354     3 -0.913 1.01   6.06      0     1
 4 Aff01dea3 B     F     -1.23      3  1.45  1.46  -0.701     0     0
 5 B6bad4423 B     F      0.904     7  1.04  1.49  -0.701     0     1
 6 Dd3d286c0 C     D     -0.522    10  0.512 2.29   6.06      0     0
 7 Da5532cc3 C     E      0.744     7  0.839 1.67   6.06      0     0
 8 Be2341c24 C     E      0.596     5  1.78  2.90   6.06      0     0
 9 D30c73efd C     E      0.0447    5  0.689 2.46   6.06      0     0
10 C1ef06cb4 C     F     -0.644     4  0.437 2.72   6.06      0     1
# ... with 276 more rows
```

17.5.4 Creating Variables

Some times you may need to transform variables to a new variable. This can be done by using the `mutate()` where you specify the name of the new variable and set equal to the transformation of other variables. Using the `data2` data set, create a new variable called `logvar1` and set that to the log of `va1`.

```
data2 %>% mutate(logvar1 = log(va1))
```

```
# A tibble: 1,000 x 6
   ID1       ID_1     ID_2       va1    va2 logvar1
   <chr>     <chr>    <chr>    <dbl>  <dbl>   <dbl>
```

```
 1 A2b6115fd 2b6115fd A     0.458   81.4    -0.782
 2 Ac51c9cf1 c51c9cf1 A     0.236   -1.15   -1.44
 3 A7534d3a0 7534d3a0 A     0.254    1.16   -1.37
 4 A73fc5642 73fc5642 A     0.0411  -1.21   -3.19
 5 Ae020e4bd e020e4bd A     0.266   -2.31   -1.32
 6 Ac0d3b0fe c0d3b0fe A     0.00992 -0.882  -4.61
 7 A2edfed41 2edfed41 A     0.293   -0.375  -1.23
 8 Ad38a4bbe d38a4bbe A     0.261   -1.09   -1.34
 9 A5ee0f97f 5ee0f97f A     0.186   -6.14   -1.68
10 Ad791c03d d791c03d A     0.0368  -0.258  -3.30
# ... with 990 more rows
```

The `mutate()` allows you to create multiple new variables at once. Id addition to `logvar1`, create a new variable called `sqrtvar2` and set that equal to the square root of `va2`.

```
data2 %>% mutate(logvar1 = log(va1), sqrtvar2 = sqrt(va2))
```

```
# A tibble: 1,000 x 7
   ID1       ID_1     ID_2      va1    va2 logvar1 sqrtvar2
   <chr>     <chr>    <chr>   <dbl>  <dbl>   <dbl>    <dbl>
 1 A2b6115fd 2b6115fd A     0.458   81.4    -0.782     9.02
 2 Ac51c9cf1 c51c9cf1 A     0.236   -1.15   -1.44    NaN
 3 A7534d3a0 7534d3a0 A     0.254    1.16   -1.37      1.08
 4 A73fc5642 73fc5642 A     0.0411  -1.21   -3.19    NaN
 5 Ae020e4bd e020e4bd A     0.266   -2.31   -1.32    NaN
 6 Ac0d3b0fe c0d3b0fe A     0.00992 -0.882  -4.61    NaN
 7 A2edfed41 2edfed41 A     0.293   -0.375  -1.23    NaN
 8 Ad38a4bbe d38a4bbe A     0.261   -1.09   -1.34    NaN
 9 A5ee0f97f 5ee0f97f A     0.186   -6.14   -1.68    NaN
10 Ad791c03d d791c03d A     0.0368  -0.258  -3.30    NaN
# ... with 990 more rows
```

If you want to create categorical variables, use the `mutate()` and the `if_else()`. The `if_else()` requires three arguments: `condition=`, `true=`, and `false=`. The first argument requires a condition that will return a logical value. If true, then R will assign what is stated in the `true=`, otherwise R will assign what is in the `false=`. To begin, find the median of `va1` from `data2` and assign it to `medval`.

```
medval <- data2$va1 %>% median()
```

No create a new variable called `diva1` where if `va1` is greater than the median of `va1`, assign it "A", otherwise assign it "B".

```
data2 %>% mutate(diva1=if_else(va1>medval,"A","B"))
```

```
# A tibble: 1,000 x 6
   ID1       ID_1     ID_2      va1    va2 diva1
   <chr>     <chr>    <chr>   <dbl>  <dbl> <chr>
 1 A2b6115fd 2b6115fd A     0.458   81.4   A
 2 Ac51c9cf1 c51c9cf1 A     0.236   -1.15  B
 3 A7534d3a0 7534d3a0 A     0.254    1.16  B
 4 A73fc5642 73fc5642 A     0.0411  -1.21  B
 5 Ae020e4bd e020e4bd A     0.266   -2.31  B
 6 Ac0d3b0fe c0d3b0fe A     0.00992 -0.882 B
 7 A2edfed41 2edfed41 A     0.293   -0.375 A
 8 Ad38a4bbe d38a4bbe A     0.261   -1.09  B
 9 A5ee0f97f 5ee0f97f A     0.186   -6.14  B
10 Ad791c03d d791c03d A     0.0368  -0.258 B
# ... with 990 more rows
```

17.5.5 Merging Datasets

One of the last thing is to go over how to merge data sets together. To merge the data sets, we use the `full_join()`. The `full_join()` needs two data sets (separated by commas) and the `by=` which provides the variables needed (must be the same name for each data set) to merge the data sets. Merge `data1` and `data2` with the variable `ID1`.

```
full_join(data1, data2, by = "ID1")
```

```
# A tibble: 1,000 x 14
   ID1    cat1  cat2    var1  var2   var3   var4   var5  var6  var7 ID_1  ID_2
   <chr>  <chr> <chr>  <dbl> <dbl>  <dbl>  <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
 1 A2b61~ A     E      1.15      4  2.88   3.54   -3.18     0     1 2b61~ A
 2 Ac51c~ A     D     -0.367     3  0.0640 1.07   -3.18     0     0 c51c~ A
 3 A7534~ A     F     -0.420     6 -1.04   0.226  -3.18     0     0 7534~ A
 4 A73fc~ A     F     -2.00      5 -0.314  0.0213 -3.18     0     0 73fc~ A
 5 Ae020~ C     E      0.697     3  0.526  2.98    6.06     0     0 e020~ A
 6 Ac0d3~ A     D      0.467     4  2.45   1.85   -3.18     0     0 c0d3~ A
 7 A2edf~ A     E      1.36      5  0.514  0.529  -3.18     0     1 2edf~ A
 8 Ad38a~ C     F      0.369     0  1.98   2.36    6.06     0     1 d38a~ A
 9 A5ee0~ B     E      1.80      5  0.147  2.22   -0.701     0     0 5ee0~ A
10 Ad791~ C     E      1.25      3 -1.05   0.0289  6.06      0     1 d791~ A
# ... with 990 more rows, and 2 more variables: va1 <dbl>, va2 <dbl>
```

The `full_join()` allows you to merge data sets using two variables instead of one. All you need to do is specify `by=` with a vector specifying the arguments. Merge `data2` and `data3` by `ID_1` and `ID_2`.

```
full_join(data2, data3, by = c("ID_1","ID_2"))
```

```
# A tibble: 1,000 x 8
   ID1.x     ID_1     ID_2      va1    va2 ID1.y         v1     v2
   <chr>     <chr>    <chr>   <dbl>  <dbl> <chr>      <dbl>  <dbl>
 1 A2b6115fd 2b6115fd A     0.458   81.4   A2b6115fd 0.361  0.278
 2 Ac51c9cf1 c51c9cf1 A     0.236   -1.15  Ac51c9cf1 0.273  2.64
 3 A7534d3a0 7534d3a0 A     0.254    1.16  A7534d3a0 1.17   0.119
 4 A73fc5642 73fc5642 A     0.0411  -1.21  A73fc5642 0.879  0.705
 5 Ae020e4bd e020e4bd A     0.266   -2.31  Ae020e4bd 0.0268 0.297
 6 Ac0d3b0fe c0d3b0fe A     0.00992 -0.882 Ac0d3b0fe 1.18   3.16
 7 A2edfed41 2edfed41 A     0.293   -0.375 A2edfed41 0.356  0.174
 8 Ad38a4bbe d38a4bbe A     0.261   -1.09  Ad38a4bbe 0.430  0.130
 9 A5ee0f97f 5ee0f97f A     0.186   -6.14  A5ee0f97f 0.643  0.0231
10 Ad791c03d d791c03d A     0.0368  -0.258 Ad791c03d 0.183  0.311
# ... with 990 more rows
```

17.6 Reshaping Data

This section focuses on reshaping the data to prepare it for analysis. For example, to conduct longitudinal data analysis, you will need to have long data. Reshaping data may be with converting data from wide to long, converting back from long to wide, splitting variables, splitting rows and merging variable. The functions used in this lesson are from the `tidy::`

17.6.1 Wide to Long Data

Converting data from wide to long is necessary when the data looks like `data4`, view `data4`:

```
data4
```

```
# A tibble: 1,000 x 5
   ID1           X1     X2    X3    X4
   <chr>      <dbl>  <dbl> <dbl> <dbl>
 1 Ad9131ee9  0.800  4.68   1.46  5.35
 2 A9c5988ea  1.17   1.50   4.83  3.75
 3 A28a5479d  1.85   2.64   2.39  4.34
 4 Aaf5537cc  1.55   2.28   3.35  3.76
 5 A370958bd -1.36   2.48   2.06  4.70
 6 Aea997e13  2.37   3.27   3.11  3.31
 7 A3563646f  2.10  -0.902  2.49  2.75
 8 A9b3cfdba -0.513  0.271  2.97  2.97
 9 A32b6737a  1.28   2.02   3.48  4.87
10 A30e96748  1.30   1.72   2.11  2.04
# ... with 990 more rows
```

Let's say `data4` represents biomarker data. Variable `ID1` represents a unique identifier for the participant. Then `X1`, `X2`, `X3`, and `X4` represents a value collected for a participant at different time point. This is know as repeated measurements. This data is considered wide because the repeated measurements are on the same row. To make it long, the repeated measurements must be on the same column.

To convert data from long to wide, we will use the `pivot_longer()` with the first argument taking variables of the repeated measurements, `c(X1:X4)` or `X1:X4`, the second argument asks for the name for the variable that contains the stored repeated measurements, the variable names, and the last variable asks for the name for all the values, the data collected. Convert the `data4` to long and name the variable names column `"measurement"`, and values column `"value"`.

```
data4 %>% pivot_longer(X1:X4, "measurement", "value")
```

```
# A tibble: 4,000 x 3
   ID1       measurement value
   <chr>     <chr>       <dbl>
 1 Ad9131ee9 X1          0.800
 2 Ad9131ee9 X2          4.68
 3 Ad9131ee9 X3          1.46
 4 Ad9131ee9 X4          5.35
 5 A9c5988ea X1          1.17
 6 A9c5988ea X2          1.50
 7 A9c5988ea X3          4.83
 8 A9c5988ea X4          3.75
 9 A28a5479d X1          1.85
```

```
10 A28a5479d X2          2.64
# ... with 3,990 more rows
```

17.6.2 Long to Wide

If you need to convert data from long to wide, use the `pivot_wider()`. You will need to specify the `names_from=` which specifies the variable names for the wide data set, and you will need to specify the `values_from=` that specifies variable that contains the values in the long data set. Convert `data5` from long to wide data. Note, you must specify the arguments for this function.

```
data5 %>% pivot_wider(names_from = measurement,
                      values_from = value)
```

```
# A tibble: 1,000 x 5
   ID1           X1     X2    X3    X4
   <chr>      <dbl>  <dbl> <dbl> <dbl>
 1 Ad9131ee9  0.800  4.68   1.46  5.35
 2 A9c5988ea  1.17   1.50   4.83  3.75
 3 A28a5479d  1.85   2.64   2.39  4.34
 4 Aaf5537cc  1.55   2.28   3.35  3.76
 5 A370958bd -1.36   2.48   2.06  4.70
 6 Aea997e13  2.37   3.27   3.11  3.31
 7 A3563646f  2.10  -0.902  2.49  2.75
 8 A9b3cfdba -0.513  0.271  2.97  2.97
 9 A32b6737a  1.28   2.02   3.48  4.87
10 A30e96748  1.30   1.72   2.11  2.04
# ... with 990 more rows
```

17.6.3 Spliting Variables

Before we begin, look at `data6`:

```
data6
```

```
# A tibble: 1,000 x 4
   ID1       merge          X3    X4
   <chr>     <chr>       <dbl> <dbl>
 1 Ad9131ee9 -1.23/2.64   2.12  3.56
 2 A9c5988ea 1.74/3.02    4.09  4.88
 3 A28a5479d 0.87/3.56   3.47  4.47
 4 Aaf5537cc 1.05/2.01   3.61  5.03
 5 A370958bd -1.47/1.26  3.98  6.59
 6 Aea997e13 1.66/3.51   1.65  2.72
 7 A3563646f 1.81/1.7    4.29  3.13
 8 A9b3cfdba 1.8/2.26    1.94  5.23
 9 A32b6737a 2.38/1.68   3.06  3.3
10 A30e96748 1/2.17      2.59  3.03
# ... with 990 more rows
```

Notice how the `merge` variable has two values separated by "/". If we need to split the variable into two variables, we need to specify the `separate()`. All you need to specify is the variable you need to split, the name of the 2 new variables, in a character vector, and how to split the variable `"/"`. Split the variable `merge` in `data6` to two new variables called `X1` and `X2`.

```
data6 %>% separate(merge, c("X1", "X2"), "/")
```

```
# A tibble: 1,000 x 5
   ID1       X1    X2       X3    X4
   <chr>     <chr> <chr> <dbl> <dbl>
 1 Ad9131ee9 -1.23 2.64   2.12  3.56
 2 A9c5988ea 1.74  3.02   4.09  4.88
 3 A28a5479d 0.87  3.56   3.47  4.47
 4 Aaf5537cc 1.05  2.01   3.61  5.03
 5 A370958bd -1.47 1.26   3.98  6.59
 6 Aea997e13 1.66  3.51   1.65  2.72
 7 A3563646f 1.81  1.7    4.29  3.13
 8 A9b3cfdba 1.8   2.26   1.94  5.23
 9 A32b6737a 2.38  1.68   3.06  3.3
10 A30e96748 1     2.17   2.59  3.03
# ... with 990 more rows
```

17.6.4 Splitting Rows

The variable `merge` in `data6` was split into different variables before, now instead of variables, let's split it into different rows instead. To do this, use the `separate_rows()`. All you need to specify the variable name and the `sep=` (must state the argument). Split the `merge` variable from `data6` into different rows.

```
data6 %>% separate_rows(merge, sep = "/")
```

```
# A tibble: 2,000 x 4
   ID1       merge    X3    X4
   <chr>     <chr> <dbl> <dbl>
 1 Ad9131ee9 -1.23  2.12  3.56
 2 Ad9131ee9 2.64   2.12  3.56
 3 A9c5988ea 1.74   4.09  4.88
 4 A9c5988ea 3.02   4.09  4.88
 5 A28a5479d 0.87   3.47  4.47
 6 A28a5479d 3.56   3.47  4.47
 7 Aaf5537cc 1.05   3.61  5.03
 8 Aaf5537cc 2.01   3.61  5.03
 9 A370958bd -1.47  3.98  6.59
10 A370958bd 1.26   3.98  6.59
# ... with 1,990 more rows
```

17.6.5 Merging Rows

If you need to merge variables together, similar to the `merge` variable, use the `unite()`. All you need to do is specify the variables to merge, the `col=` which specifies the name of the new variable (as a character), and the `sep=` which indicates the symbol for separate value, as a character. Note, you need to specify the bot the `col=` and `sep=`. Merge variable `X3` and `X4` in `data6` to a new variable called `merge2` and have the separator be a hyphen.

```
data6 %>% unite(X3, X4, col = "merge2", sep="-")
```

```
# A tibble: 1,000 x 3
   ID1       merge      merge2
   <chr>     <chr>      <chr>
 1 Ad9131ee9 -1.23/2.64 2.12-3.56
 2 A9c5988ea 1.74/3.02  4.09-4.88
 3 A28a5479d 0.87/3.56  3.47-4.47
 4 Aaf5537cc 1.05/2.01  3.61-5.03
 5 A370958bd -1.47/1.26 3.98-6.59
 6 Aea997e13 1.66/3.51  1.65-2.72
 7 A3563646f 1.81/1.7   4.29-3.13
 8 A9b3cfdba 1.8/2.26   1.94-5.23
 9 A32b6737a 2.38/1.68  3.06-3.3
10 A30e96748 1/2.17     2.59-3.03
# ... with 990 more rows
```

17.7 Applied Example

Here is an applied example where you will use what you learned from the previous lesson and convert `data7` into `data8`. `data7` has a wide data format which contains time points labeled as `vX`, where `X` represents the time point number. At each time point, the mean, sd, and median was taken. You will need to convert the data to long where each row represents a new time point, and each row will have 3 variables representing the mean, sd, and median. View both `data7` and `data8` to have a better idea on what is going on. Remember you need to convert `data7` to `data8`.

```
data7
```

```
# A tibble: 1,000 x 13
   ID1       `v1/mean` `v1/sd` v1/med~1 v2/me~2 `v2/sd` v2/med~3 v3/me~4 `v3/sd`
   <chr>         <dbl>   <dbl>    <dbl>   <dbl>   <dbl>    <dbl>   <dbl>   <dbl>
 1 Ad9131ee9     3.11    2.86      4.50   1.93    3.21   3.27      2.65    -0.383
 2 A9c5988ea     2.03    2.90      2.08   0.709   2.27   4.13      1.45     2.01
 3 A28a5479d    -0.415   2.42      2.47   2.38   -0.820  1.22      3.44     1.63
 4 Aaf5537cc     1.25    2.24      3.71   4.00    0.456  4.32      1.54     0.789
 5 A370958bd    -0.984   0.972     3.73   2.19   -0.184  2.14      4.32    -0.804
 6 Aea997e13     1.42    1.34      2.35   2.77    4.16  -0.00874 -3.02     4.25
 7 A3563646f    -0.149   3.26      4.49   5.07    2.44   3.85      0.0388   1.92
 8 A9b3cfdba     0.270   1.57      3.25   2.89    0.422  5.01     -0.218    0.545
 9 A32b6737a     0.714   3.39      5.66   2.52    3.15   3.16     -0.784    1.39
10 A30e96748     0.467   2.47      2.64   3.97    1.76   4.00      2.21     2.34
# ... with 990 more rows, 4 more variables: `v3/median` <dbl>, `v4/mean` <dbl>,
#   `v4/sd` <dbl>, `v4/median` <dbl>, and abbreviated variable names
#   1: `v1/median`, 2: `v2/mean`, 3: `v2/median`, 4: `v3/mean`
```

```
data8
```

```
# A tibble: 4,000 x 5
   ID1       time    mean     sd  median
   <chr>     <chr>  <dbl>  <dbl>   <dbl>
 1 Ad9131ee9 v1     3.11   2.86   4.50
 2 Ad9131ee9 v2     1.93   3.21   3.27
 3 Ad9131ee9 v3     2.65  -0.383  3.23
 4 Ad9131ee9 v4     0.605  0.883  4.65
 5 A9c5988ea v1     2.03   2.90   2.08
 6 A9c5988ea v2     0.709  2.27   4.13
 7 A9c5988ea v3     1.45   2.01   2.84
 8 A9c5988ea v4     0.710  3.03  -0.0898
 9 A28a5479d v1    -0.415  2.42   2.47
10 A28a5479d v2     2.38  -0.820  1.22
# ... with 3,990 more rows
```

Now that you viewed the data set, type the code to convert `data7` to `data8`. Try working it out before you look at the solution.

```
data7 %>% pivot_longer(`v1/mean`:`v4/median`,"measurement","value") %>%
          separate(measurement,c("time","stat"),sep="/") %>%
          pivot_wider(names_from = stat,values_from = value)
```

```
# A tibble: 4,000 x 5
   ID1       time    mean     sd  median
   <chr>     <chr>  <dbl>  <dbl>   <dbl>
 1 Ad9131ee9 v1     3.11   2.86   4.50
 2 Ad9131ee9 v2     1.93   3.21   3.27
 3 Ad9131ee9 v3     2.65  -0.383  3.23
 4 Ad9131ee9 v4     0.605  0.883  4.65
 5 A9c5988ea v1     2.03   2.90   2.08
 6 A9c5988ea v2     0.709  2.27   4.13
 7 A9c5988ea v3     1.45   2.01   2.84
 8 A9c5988ea v4     0.710  3.03  -0.0898
 9 A28a5479d v1    -0.415  2.42   2.47
10 A28a5479d v2     2.38  -0.820  1.22
# ... with 3,990 more rows
```

18 Data Summarization

18.1 Descriptive Statistics

Here, we will go over some of the basic syntax to obtain basic statistics. We will use the variables `mpg` and `cyl` from the `mtcars` data set. To view the data set use the `head()`:

```r
head(mtcars)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

The variable `mpg` would be used as a continuous variable, and the variable `cyl` would be used as a categorical variable.

18.1.1 Point Estimates

The first basic statistic you can compute are point estimates. These are your means, medians, etc. Here we will calculate these estimates.

18.1.1.1 Mean

To obtain the mean, use the `mean()`, you only need to specify `x=` for the data to compute the mean:

```r
mean(mtcars$mpg)
```

```
[1] 20.09062
```

18.1.1.2 Median

To obtain the median, use the `median()`, you only need to specify `x=` for the data to compute the median:

```r
median(mtcars$mpg)
```

```
[1] 19.2
```

18.1.1.3 Frequency

To obtain a frequency table, use the `table()`, you only need to specify the data as the first argument to compute the frequency table:

```r
table(mtcars$cyl)
```

```

 4  6  8 
11  7 14 
```

18.1.1.4 Proportion

To obtain a the proportions for the frequency table, use the `prop.table()`. However the first argument must be the results from the `table()`. Use the `table()` inside the `prop.table()` to get the proportions:

```r
prop.table(table(mtcars$cyl))
```

```

      4       6       8 
0.34375 0.21875 0.43750 
```

18.1.2 Variability

In addition to point estimates, variability is an important statistic to report to let a user know about the spread of the data. Here we will calculate certain variability statistics.

18.1.2.1 Variance

To obtain the variance, use the `var()`, you only need to specify `x=` for the data to compute the variance:

```r
var(mtcars$mpg)
```

```
[1] 36.3241
```

18.1.2.2 Standard deviation

To obtain the standard deviation, use the `sd()`, you only need to specify `x=` for the data to compute the standard deviation:

```r
sd(mtcars$mpg)
```

```
[1] 6.026948
```

18.1.2.3 Max and Min

To obtain the max and min, use the `max()` and `min()`, respectively. You only need to specify the data as the first argument to compute the max and min:

```r
max(mtcars$mpg)
```

```
[1] 33.9
```

```r
min(mtcars$mpg)
```

```
[1] 10.4
```

18.1.2.4 Q1 and Q3

To obtain the Q1 and Q3, use the `quantile()` and specify the desired quantile with `probs=`. You only need to specify the data as the first argument and `probs=` (as a decimal) to compute the Q1 and Q3:

```
quantile(mtcars$mpg, .25)
```

```
   25%
15.425
```

```
quantile(mtcars$mpg, .75)
```

```
 75%
22.8
```

18.1.3 Associations

In statistics, we may be interested on how different variables are related to each other. These associations can be represented in a numerical value.

18.1.3.1 Continuous and Continuous

When we measure the association between to continuous variables, we tend to use a correlation statistic. This statistic tells us how linearly associated are the variables are to each other. Essentially, as one variable increases, what happens to the other variable? Does it increase (positive association) or does it decrease (negative association). To find the correlation in R, use the `cor()`. You will need to specify the `x=` and `y=` which represents vectors for each variable. Find the correlation between `mpg` and `hp` from the `mtcars` data set.

```
cor(mtcars$mpg, mtcars$hp)
```

```
[1] -0.7761684
```

18.1.3.2 Categorical and Continuous

When comparing categorical variables, it becomes a bit more nuanced in how to report associations. Most of time you will discuss key differences in certain groups. Here, we will talk about how to get the means for different groups of data. Our continuous variable is the `mpg` variable, and our categorical variable is the `cyl` variable. Both are from the `mtcars` data set. The `tapply()` allows us to split the data into different groups and then calculate different statistics. We only need to specify `X=` of the R object to split, `INDEX=` which is a list of factors or categories indicating how to split the data set, and `FUN=` which is the function that needs to be computed. Use the `tapply()` and find the mean `mpg` for each `cyl` group: 4, 5, and 6.

```
tapply(mtcars$mpg, list(mtcars$cyl), mean)
```

```
       4        6        8 
26.66364 19.74286 15.10000 
```

18.1.3.3 Categorical and Categorical

Reporting the association between two categorical variables is may be challenging. If you have a 2 × 2 table, you can report a ratio of association. However, any other case may be challenging. You can report a hypothesis test to indicate an association, but it does not provide much information about the effect of each variable. You can also report row, column, or table proportions. Here we will talk about creating cross tables and report these proportions. To create a cross table, use the `table()` and use the first two arguments to specify the two categorical variables. Create a cross tabulation between `cyl` and `carb` from the `mtcars` data set.

```
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
```

Notice how the first argument is represented in the rows and the second argument is in the columns. Now create table proportions using both of the variables. You first need to create the table and store it in a variable and then use the `prop.table()`.

```
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
```

To get the row proportions, use the argument `margin = 1` within the `prop.table()`.

```r
prop.table(table(mtcars$cyl, mtcars$carb),
           margin = 1)
```

```
             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
```

To get the column proportions, use the argument `margin = 2` within the `prop.table()`.

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

```
            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
```

18.2 Summarizing with Tidyverse

```r
library(magrittr)
library(tidyverse)
```

```
-- Attaching packages --------------------------------------- tidyverse 1.3.2 --
v ggplot2 3.4.0      v purrr   1.0.0
v tibble  3.1.8      v dplyr   1.0.10
v tidyr   1.2.1      v stringr 1.5.0
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x tidyr::extract()   masks magrittr::extract()
x dplyr::filter()    masks stats::filter()
x dplyr::lag()       masks stats::lag()
x purrr::set_names() masks magrittr::set_names()
```

```r
f <- function(x){
  mtcars %>% split(~.$cyl) %>% map(~shapiro.test(.$mpg))
  return(1)}
g <- function(x){
  mtcars %>% group_by(cyl) %>% nest() %>% mutate(shapiro = map(data, ~shapiro.test(.$mpg))
  return(1)}
bench::mark(f(1),g(1))
```

```
# A tibble: 2 x 6
  expression      min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 f(1)        402.6us  432.6us    2258.   134.23KB    16.9
2 g(1)         11.6ms   11.7ms      83.5    3.65MB     9.03
```

19 Graphics

Through out this chapter, we use certain notations for different components in R. To begin, when something is in a gray block, `_`, this indicates that R code is being used. When I am talking about an R Object, it will be displayed as a word. For example, we will be using the R object `mtcars`. When I am talking about an R function, it will be displayed as a word followed by an open and close parentheses. For example, we will use the mean function denoted as `mean()` (read this as "mean function"). When I am talking about an R argument for a function, it will be displayed as a word following by an equal sign. For example, we will use the data argument denoted as `data=` (read this as "data argument"). When I am referencing an R package, I will use `::` (two colons) after the name. For example, in this tutorial, I will use the `ggplot2::` (read this as "ggplot2 package") Lastly, if I am displaying R code for your reference or to run, it will be displayed on its own line. There are many components in R, and my hope is that this will help you understand what components am I talking about.

19.1 Base R Plotting

19.1.1 Introduction

This tutorial provides an introduction on how to create different graphics in R. For this tutorial, we will focus on plotting different components from the `mtcars` data set.

19.1.2 Contents

1. Basic
2. Grouping
3. Tweaking

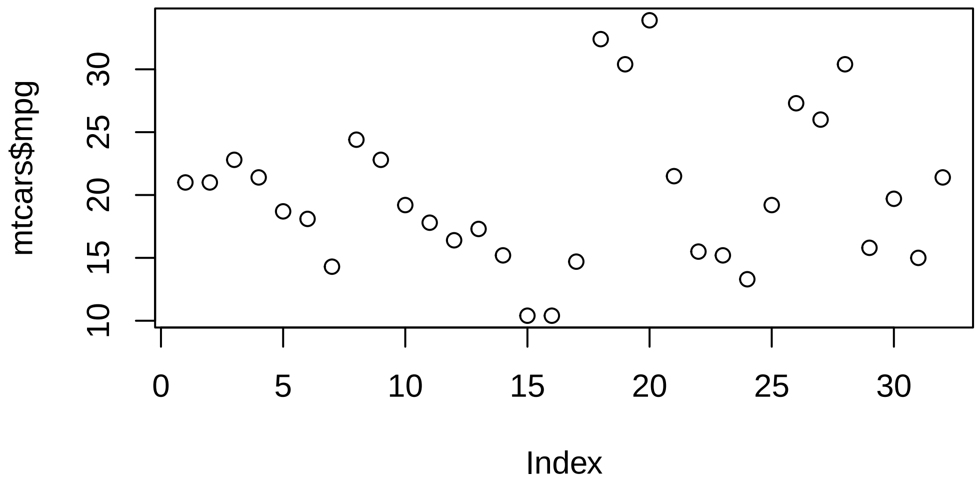
19.1.3 Basic Graphics

Here we will use the built-in R functions to create different graphics. The main function that you will use is the `plot()`. It contains much of the functionality to create many different plots in R. Additionally, it works well for different classes of R objects. It will provide many important plots that you will need for a certain statistical analysis.

19.1.4 Scatter Plot

Let's first create a scatter plot for one variable using the `mpg` variable. This is done using the `plot()` and setting the first argument `x=` to the vector.

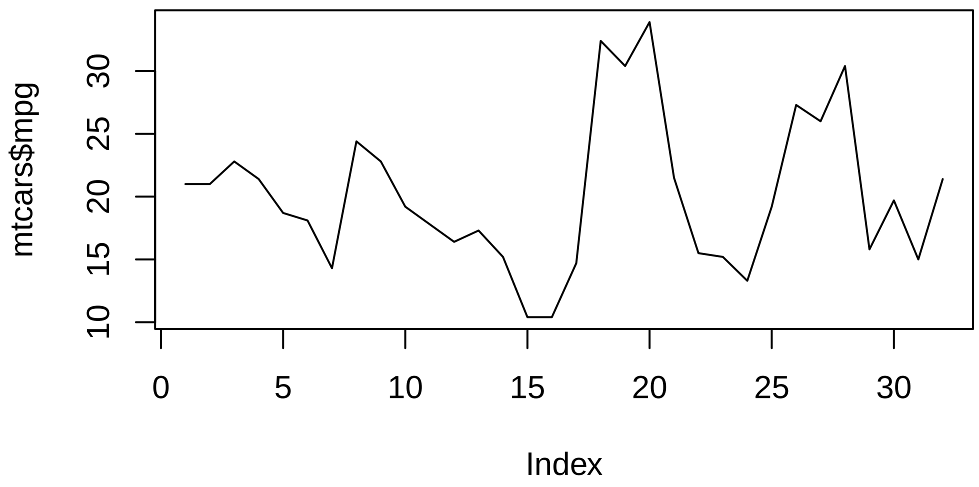
```r
plot(mtcars$mpg)
```



Notice that the x-axis is the index (which is not informative) and the y-axis is the `mpg` values.

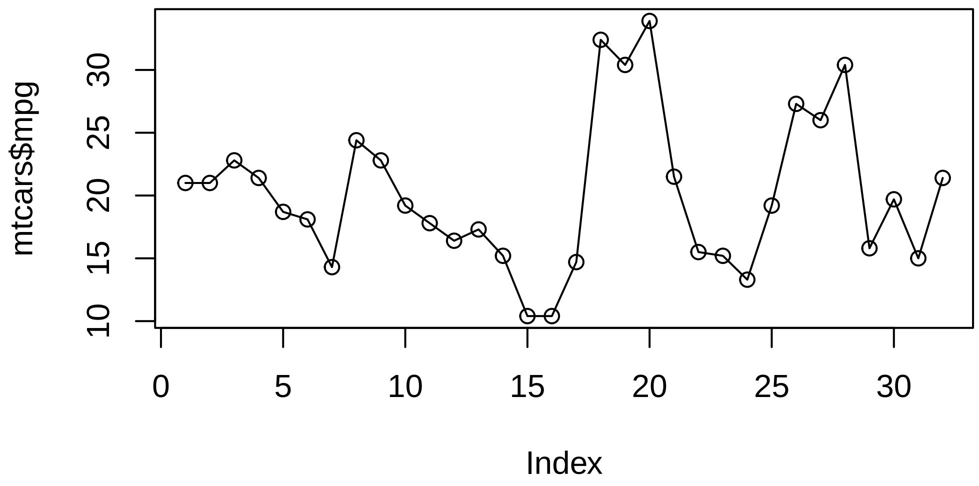
Let's connect the points with a line. This is done by setting the `type=` to `"l"`.

```r
plot(mtcars$mpg, type = "l")
```



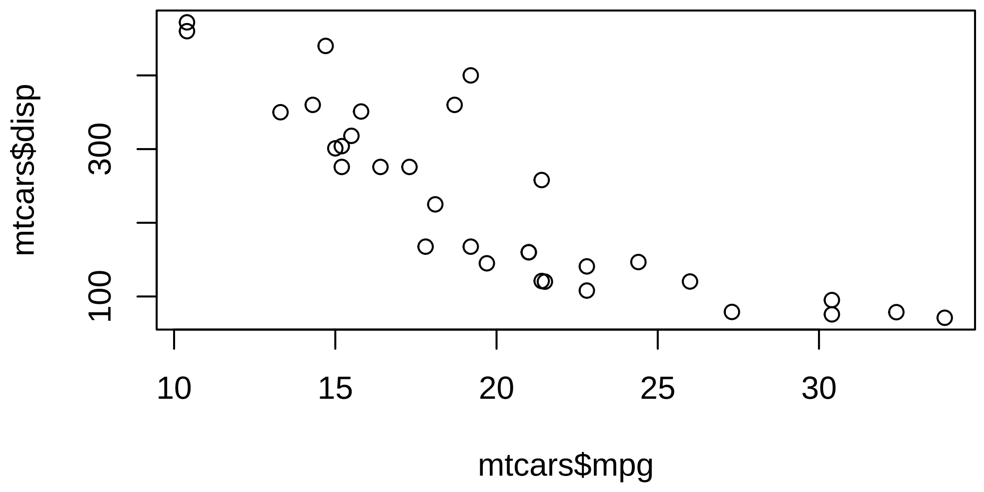
Let's add the points back to the plot and keep the lines. What we are going to do is first create the scatter plot as we did before, but we will also use the `lines()` to add the lines. The `lines()` needs the `x=` which is a vector of points (`mpg`). The two lines of code must run together.

```
plot(mtcars$mpg)
lines(mtcars$mpg)
```



Now, let's create a more realistic scatter plot with 2 variables. This is done by specifying the `y=` with another variable in addition to the `x=` in the `plot=`. Plot a scatter plot between `mpg` and `disp`.

```
plot(mtcars$mpg,mtcars$disp)
```

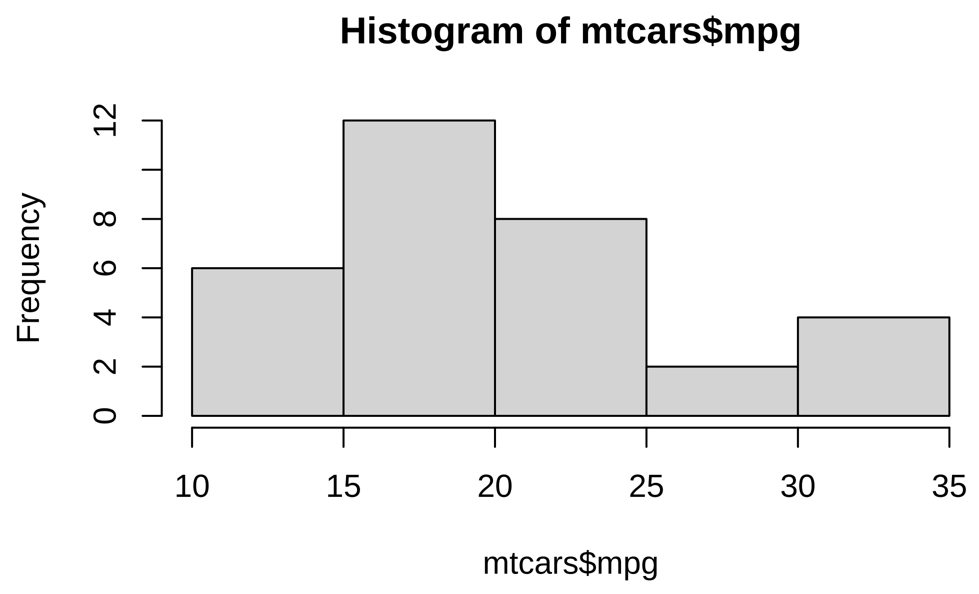


Now, let's change the the axis labels and plot title. This is done by using the arguments `main=`, `xlab=`, and `ylab`. The `main=` changes the title of the plot.

19.1.5 Histogram

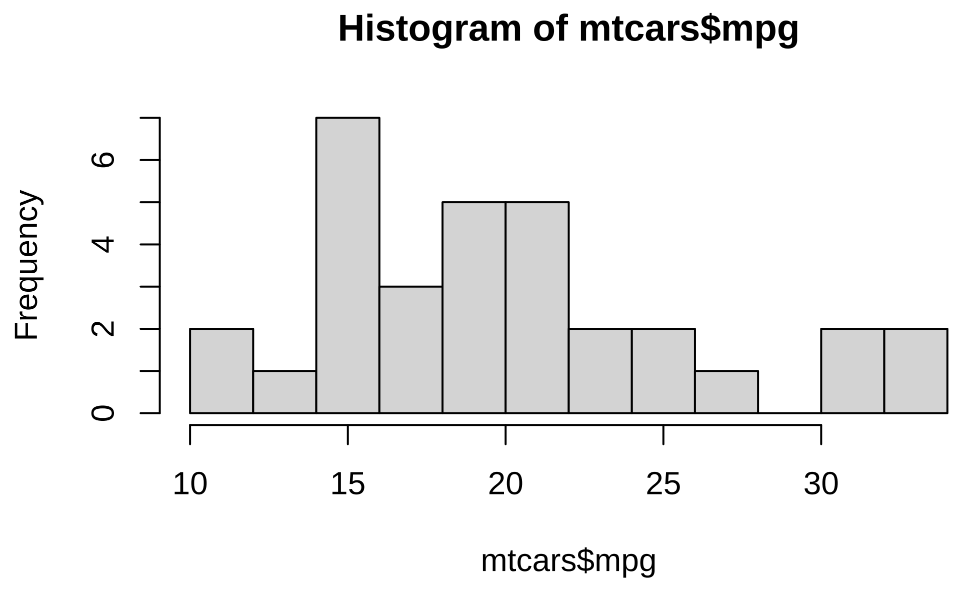
To create a histogram, use the `hist()`. The `hist()` only needs `x=` which is numerical vector. Create a histogram with the `mpg` variable.

```
hist(mtcars$mpg)
```



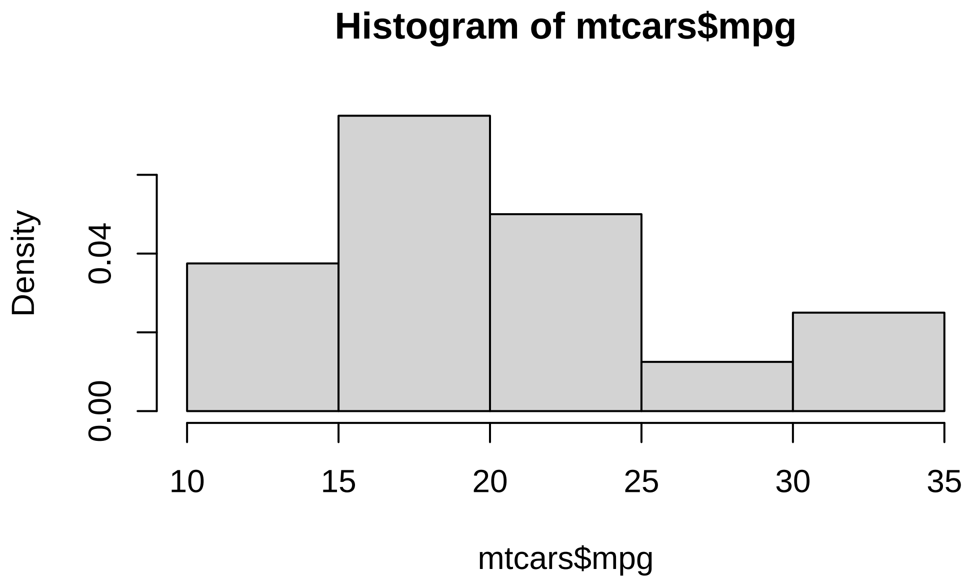
If you want to change the number of breaks in the histogram, use the `breaks=`. Create a new histogram of the `mpg` variable with ten breaks.

```r
hist(mtcars$mpg, breaks = 10)
```



The above histograms provide frequencies instead of relative frequencies. If you want relative frequencies, use the `freq=` and set it equal to `FALSE` in the `hist()`.

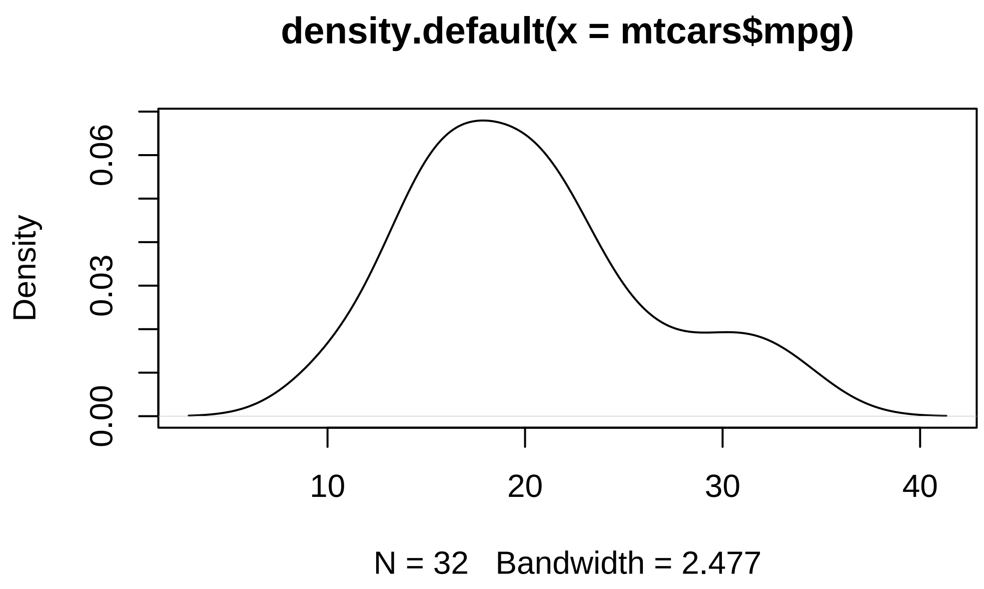
```r
hist(mtcars$mpg, freq = FALSE)
```



19.1.6 Density Plot

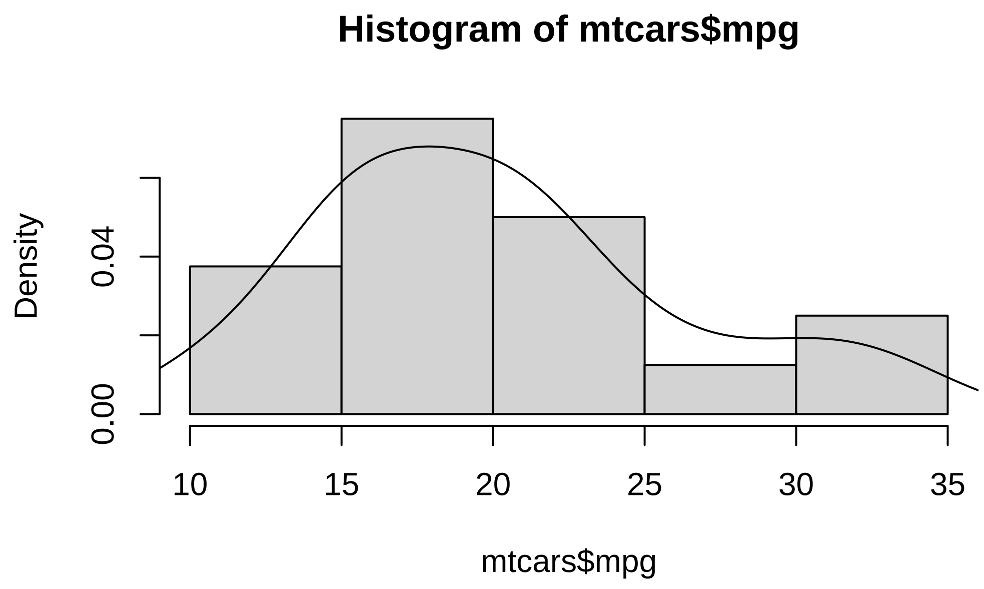
A density plot can be used instead of a histogram. This is done by using the `density()` to create an object containing the information to create density function. Then, use the `plot()` to display the plot. The only argument the `density()` needs is the `x=` which is the data to be used. Create a density plot the `mpg` variable.

```r
plot(density(mtcars$mpg))
```



Now, if we want to overlay the density function over a histogram, use the `lines()` with the output from the `density()` as its main input. First create the histogram using the `hist()` and setting the `freq=` to `FALSE`. Then use the `lines()` to overlay the density. Make sure to run both lines together.

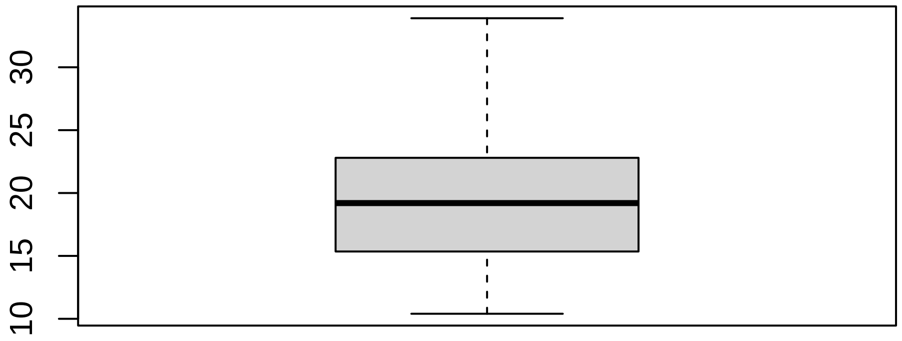
```
hist(mtcars$mpg, freq = FALSE)
lines(density(mtcars$mpg))
```



19.1.7 Box Plots

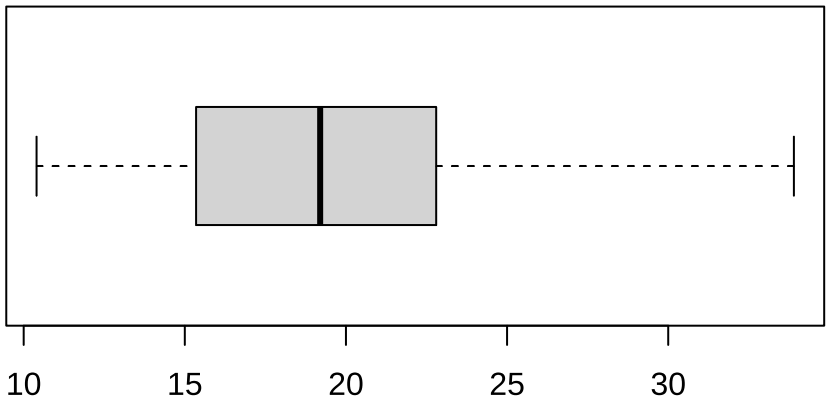
A commonly used plot to display relevant statistics is the box plot. To create a box plot use the `boxplot()`. The function only needs the `x=` which specifies the data to create the box plot. Use the box plot function to create a box plot on for the variable `mpg`.

```
boxplot(mtcars$mpg)
```



If you want to make the box plot horizontal, use `horizontal=` and set it equal to `TRUE`.

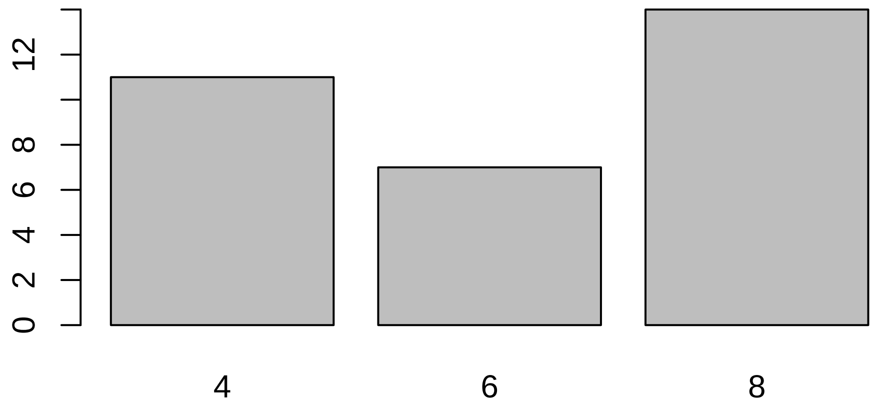
```
boxplot(mtcars$mpg, horizontal = TRUE)
```



19.1.8 Bar Chart

A histogram shows you the frequency for a continuous variable. A bar chart will show you the frequency of a categorical or discrete variable. To create a bar chart, use the `barplot()`. The main argument it needs is the `height=` which needs to an object from the `table()`. Create a bar chart for the `cyl` variable.

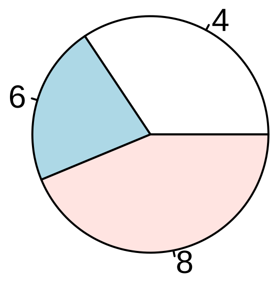
```
barplot(table(mtcars$cyl))
```



19.1.9 Pie Chart

While I do not recommend using a pie chart, R is capable of creating one using the `pie()`. It only needs the `x=` which is a vector numerical quantities. This could be the output from the `table()`. Create a pie chart with the `cyl` variable.

```
pie(table(mtcars$cyl))
```



19.1.10 Grouping

Similar to obtaining statistics for certain groups, plots can be grouped to reveal certain trends. We will look at a couple of methods to visualize different groups.

19.1.10.1 One Variable Grouping

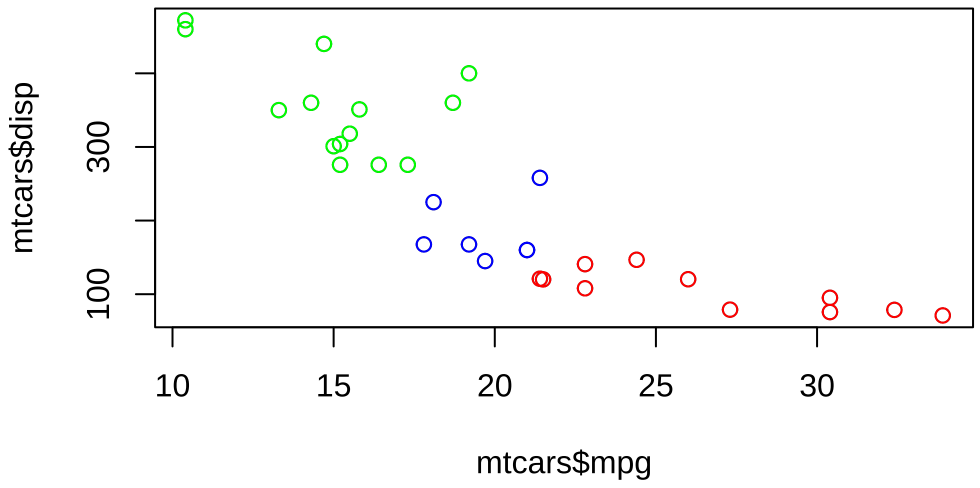
Two ways to display groups is by using color coding or panels. I will show you what I think is the best way to group variables. There may be better ways to do this, such as using the ggplot2 package. Before we begin, create three new R objects that are a subset of the `mtcars` data set into 3 different data sets with for the three different values of the `cyl` variable: "4", "6", and "8". use the `subset()` to create the different data sets. Name the new R objects `mtcars_4`, `mtcars_6`, and `mtcars_8`, respectively.

```
mtcars_4 <- subset(mtcars, cyl == 4)
mtcars_6 <- subset(mtcars, cyl == 6)
mtcars_8 <- subset(mtcars, cyl == 8)
```

19.1.10.1.1 Scatter Plot

To create different colors points for their respective label associated `cyl` variable. First create a base scatter plot using the `plot()` to set up the plot. Then one by one, overlay a set of new points on the base plot using the `points()`. The first two arguments should be the vectors of data from their respective R object subset. Also, use the `col=` to change the color of the points. The `col=` takes either a string or a number.

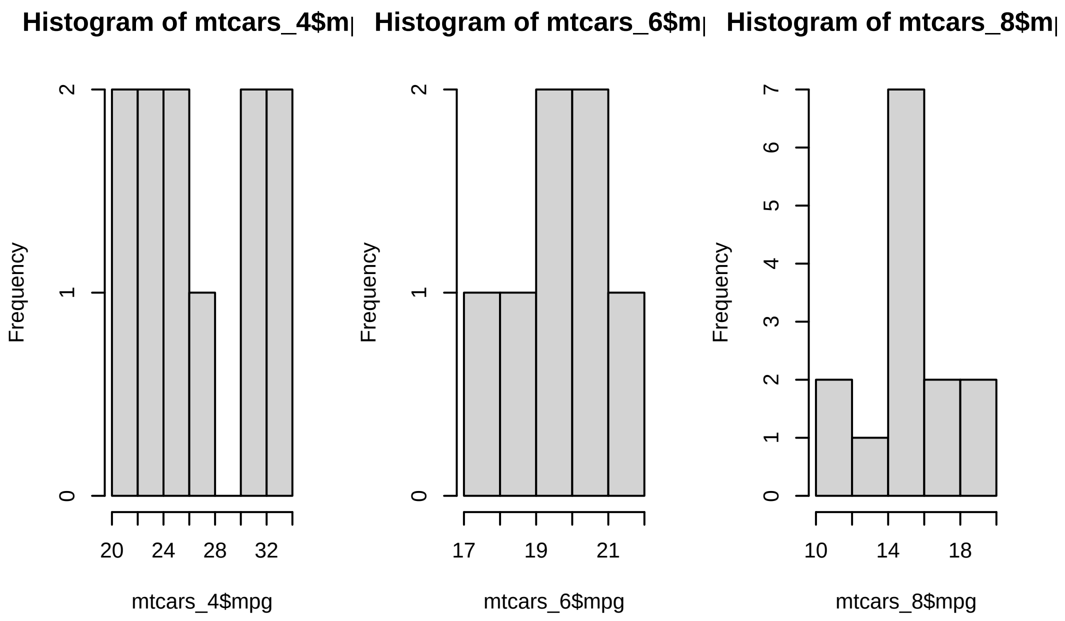
```
plot(mtcars$mpg, mtcars$disp)
points(mtcars_4$mpg, mtcars_4$disp, col = "red")
points(mtcars_6$mpg, mtcars_6$disp, col = "blue")
points(mtcars_8$mpg, mtcars_8$disp, col = "green")
```



19.1.10.1.2 Histogram

Now, it us more difficult to overlay histograms on a plot to different colors. Therefore, a panel approach may be more beneficial. This can be done by setting up R to plot a grid of plots. To do this, use the `par()` to tell R how to set up the grid. Then use the `mfrow=`, which is a vector of length two, to set up a grid. The `mfrow=` usually has an input of `c(ROWS,COLS)` which states the number of rows and the number of columns. Once this is done, the next plots you create will be used to populate the grid.

```
par(mfrow=c(1,3))
hist(mtcars_4$mpg)
hist(mtcars_6$mpg)
hist(mtcars_8$mpg)
```

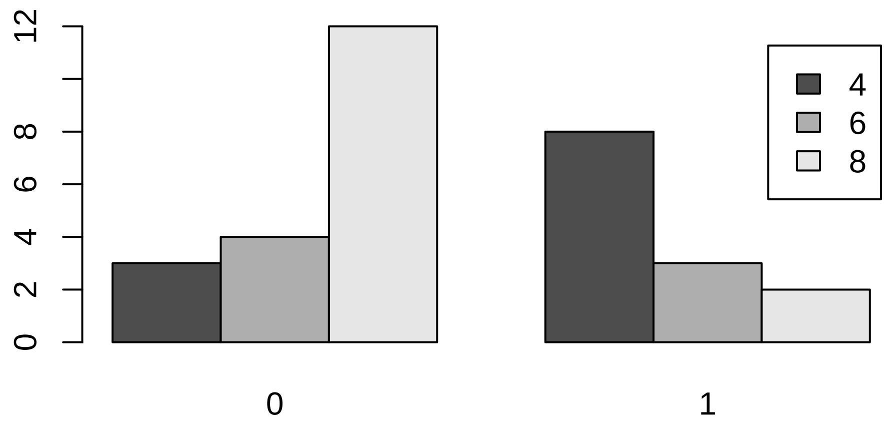


Every time you use the `par()`, it will change how graphics are created in an R session. Therefore, all your plots will follow the new graphic parameters. You will need to reset it by typing `dev.off()`.

19.1.10.1.3 Bar Chart

To visualize two categorical variables, we can use a color-coded bar chart to compare the frequencies of the categories. This is simple to do with the `barplot()`. First, use the `table()` to create a cross-tabulation of the frequencies for two variables. Then use the `boxplot()` to visualize both variables. Then use `legend=` to create a label when the bar chart is color-coded. Additionally, use the `beside=` argument to change how the plot looks. Use the code below to compare the variables `cyl` and `am` variable.

```
barplot(table(mtcars$cyl, mtcars$am), beside = TRUE, legend = rownames(table(mtcars$cyl, m
```



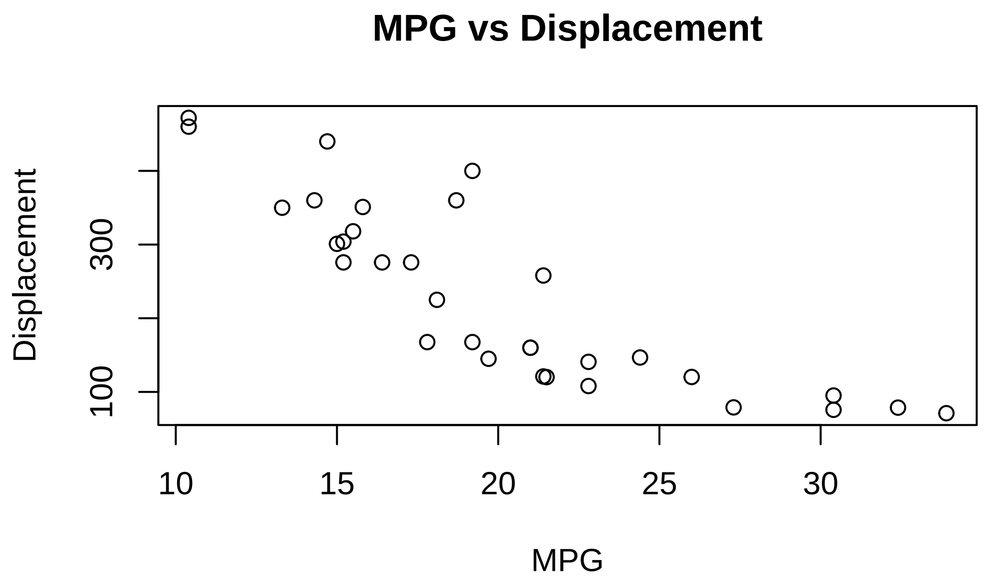
Notice that I use the `rownames()` to label the legend.

19.1.11 Tweaking

19.1.11.1 Labels

The main tweaking of plots I will talk about is changing the the axis label and titles. For the most part, each function allows you to use the `main=`, `xlab=`, and `ylab=`. The `main=` allows you to change the title. The `xlab=` and `ylab=` allow you to change the labels for the x-axis and y-axis, respectively. Create a scatter plot for the variables `mpg` and `disp` and change the labels.

```
plot(mtcars$mpg, mtcars$disp, main = "MPG vs Displacement", xlab = "MPG", ylab = "Displace
```



19.2 ggplot2

19.2.1 Introduction

The `ggplot2::` provides a set of functions to create different graphics. For more information on plotting in `ggplot2::`, please visit the this excellent resource. Here we will discuss some of the basics to the `ggplot2::``. To me,`ggplot2::‘`creates a plot by adding layers to a base plot. The syntax is designed for you to change different components of a plot in an intuitive manner. For this tutorial, we will focus on plotting different components from the`mpg‘ data set.

19.2.1.1 Contents

1. Basic
2. Grouping
3. Themes/Tweaking

19.2.2 Basics

To begin, the `ggplot2::` really works well when you are using data frames. If you have any output that you want to plot, convert into to a data frame. Once we have our data set, the first thing you would want to do is specify the main components of your base plot. This will be what will be plotted on your x-axis, and what will be plotted on your y-axis. Next, you will create the the type of plot. Lastly, you will add different layers to tweak the plot for your needs. This can be changing the layout or even overlaying another plot. The ‘ggplot2::“ provides you with tools to do almost everything you need to create a plot easily.

Before we begin plotting, load the `ggplot2::` in R.

```r
library(ggplot2)
```

Now, when we create a base plot, we will use the `ggplot()`. This will initialize the data that we need to use with the `data=` and how to map it on the x and y axis with the `mapping=`. With the `mapping=`, you will need to use the `aes()` which constructs the mapping function for the base plot. The `aes()` requires the `x=` and optionally uses the `y=` to set which values represents the x and y axis. The `aes()` also accepts other arguments for grouping or other aesthetics.

Before we begin, create a new variable in `mtcars` called `ind` and place a numeric vector which contains integers from 1 to 32.

```r
mtcars$ind <- c(1:32)
```

Now, let’s create the base plot and assign it to `gg_1`. Use the `ggplot()` and set `mtcars` as its data and the variable `ind` as `x=` and `mpg` as the `y=`

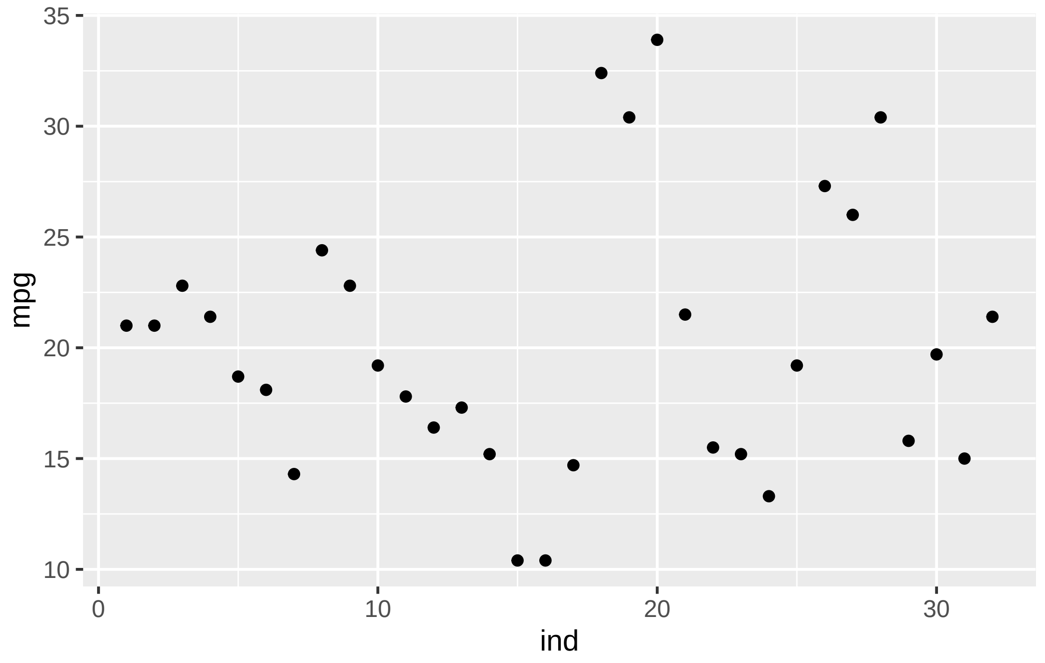
```r
gg_1 <- ggplot(mtcars, aes(ind, mpg))
```

This base plot is now used to create certain plots. Plots are created by adding functions to the base plot. This is done by using the `+` operator and then a specific `ggplot2::` function. Below we will go over some of the functions necessary.

19.2.3 Scatter Plot

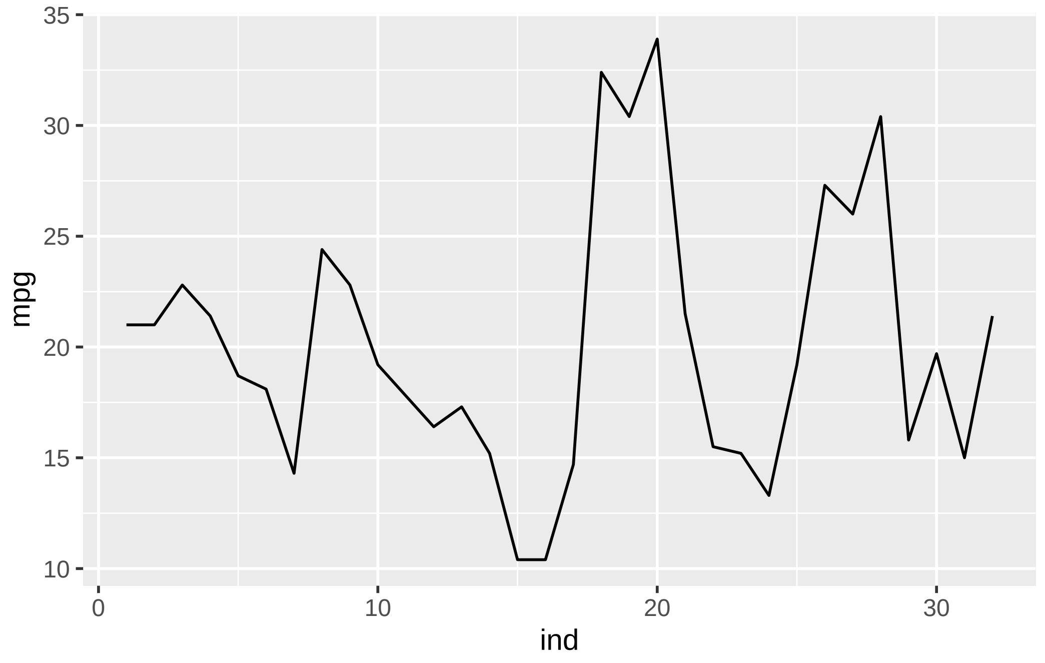
To create a scatter plot in `ggplot2::`, add the `geom_point()` to the base plot. You do not need to specify any arguments in the function. Create a scatter plot to `gg_1`

```r
gg_1 + geom_point()
```



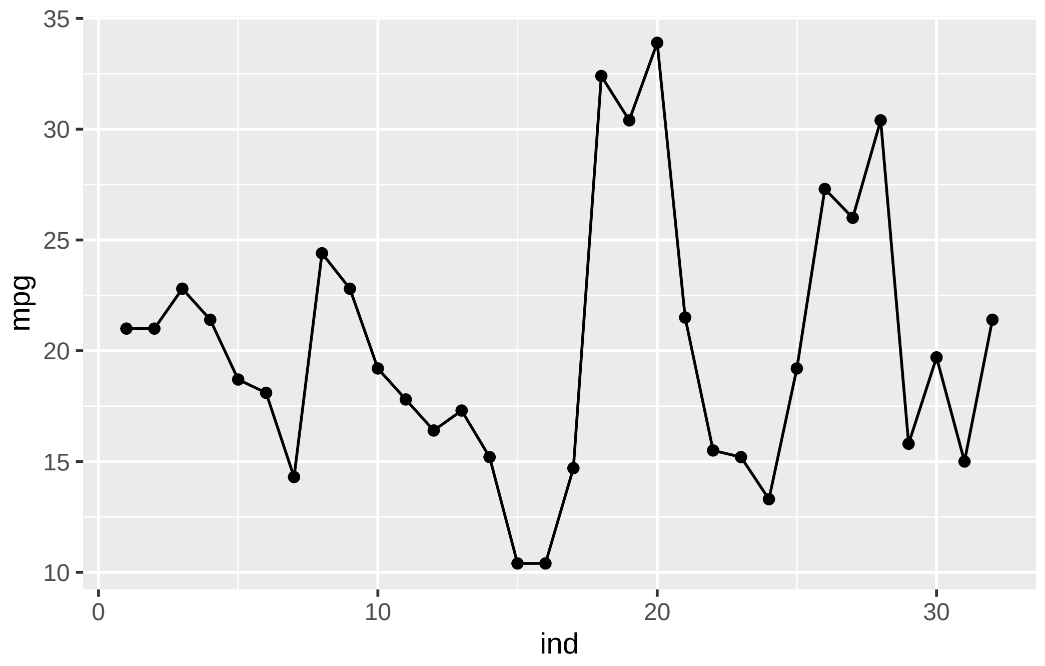
If we want to put lines instead of points, we will need to use the `geom_point()`. Change the points to a line.

```
gg_1 + geom_line()
```



To overlay points to the plot, add `geom_point()` as well as `geom_line()`. Add points to the plot above.

```
gg_1 + geom_point() + geom_line()
```

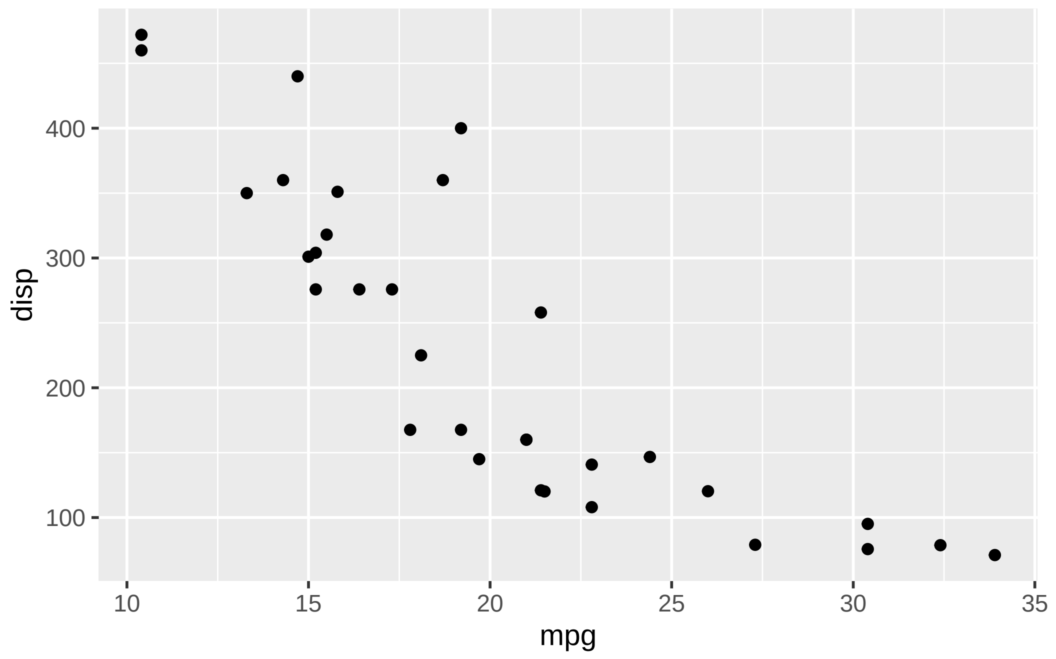


To create a 2 variable scatter plot. You will just need to specify the `x=` and `y=` in the `aes()`. Create a base plot using the `mtcars` data set and use the `mpg` and `disp` as the x and y variables, respectively, and assign in it to `gg_2`

```
gg_2 <- ggplot(mtcars, aes(mpg, disp))
```

Now create a scatter plot using `gg_2`.

```
gg_2 + geom_point()
```



19.2.4 Histogram and Density Plot

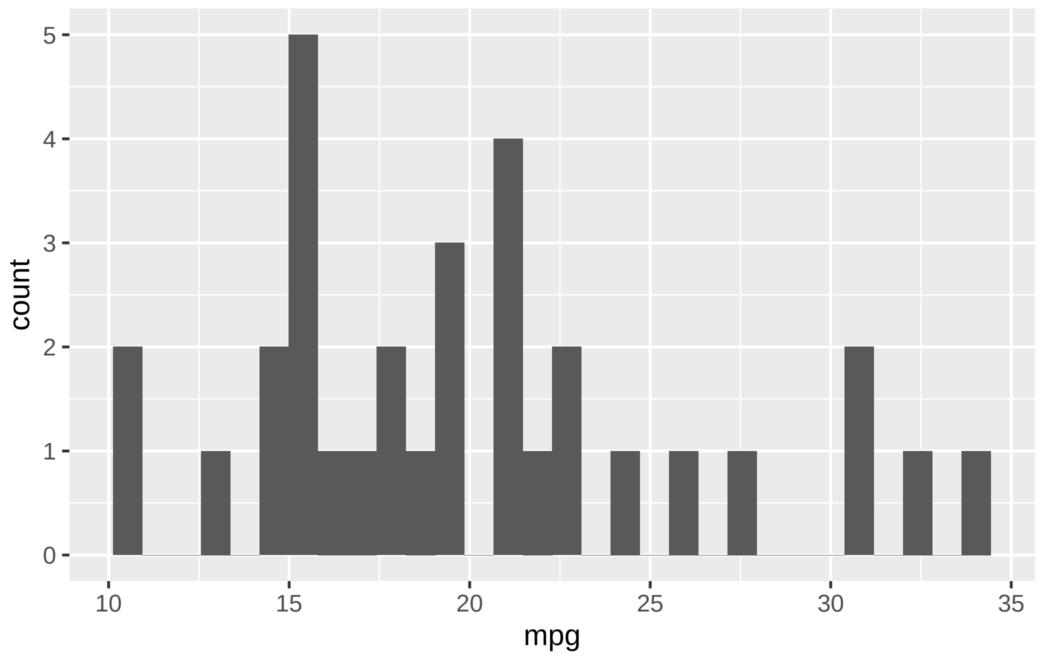
To create a histogram and density plots, create a base plot and specify the variable of interest in the `aes()`, only specify one variable. Create a base plot using the `mtcars` data set and the `mpg` variable. Assign it to `gg_3`.

```
gg_3 <- ggplot(mtcars, aes(mpg))
```

To create a histogram, use the `geom_histogram()`.

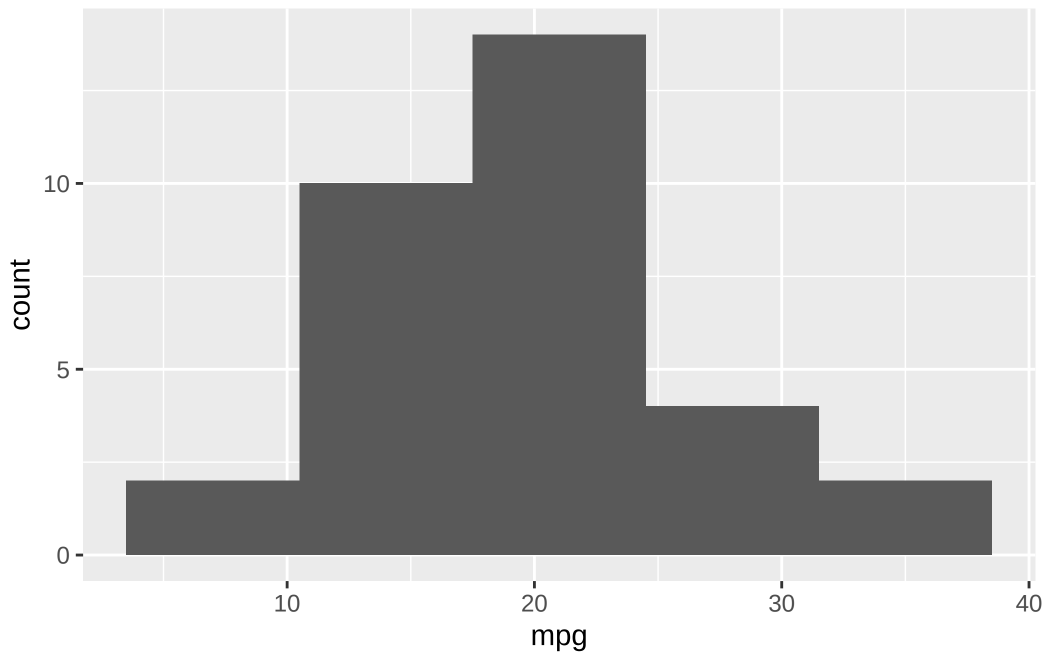
```
gg_3 + geom_histogram()
```

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



The above plot shows a histogram, but the number of bins is quite large. We can change the bin width argument, `binwidth=`, the the `geom_histogram()`. Change the bin width to seven.

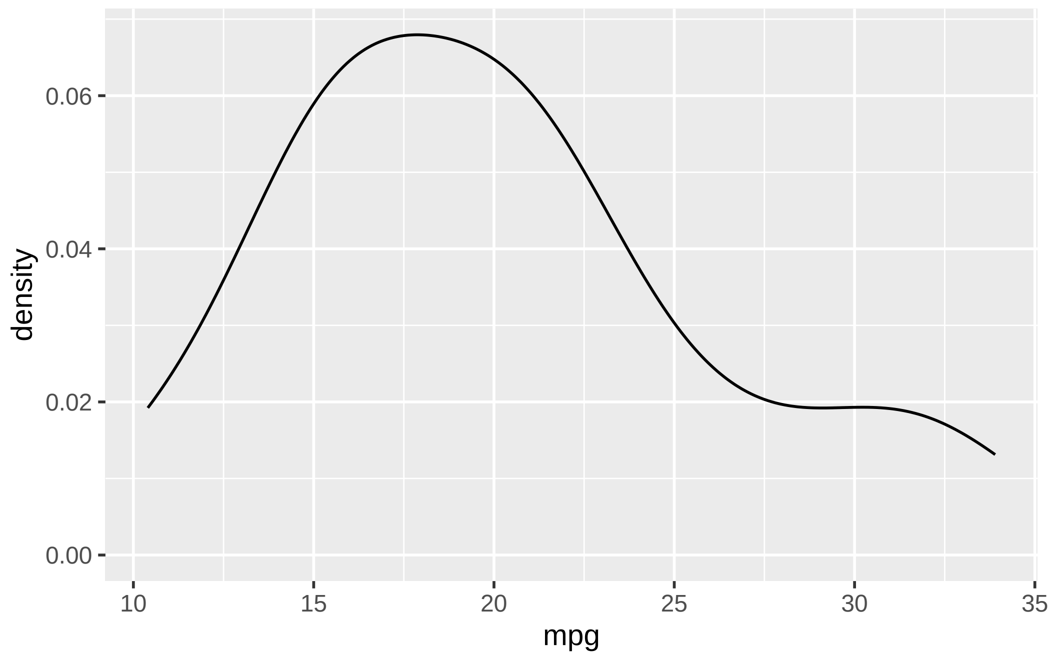
```
gg_3 + geom_histogram(binwidth = 7)
```



19.2.4.1 Density Plot

To create a density plot, use the `geom_density()`. Create a density plot for the `mpg` variable.

```r
gg_3 + geom_density()
```

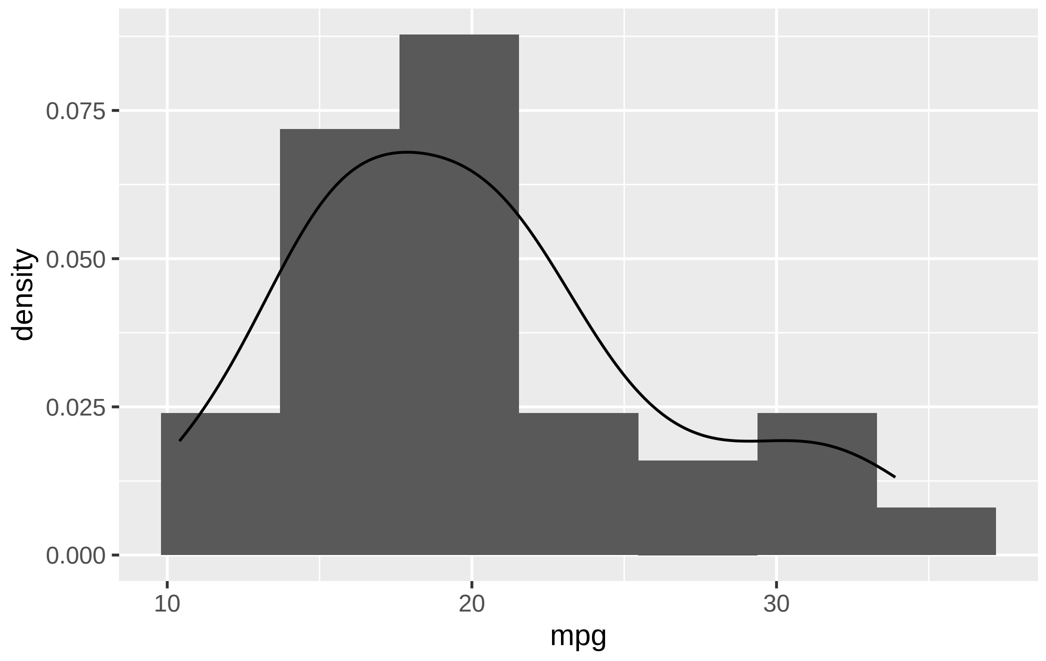


19.2.4.2 Both

Similar to adding lines and points in the same plot, you can add a histogram and a density plot by adding both the `geom_histogram()` and `geom_density()`. However, in the `geom_histogram()`, you must add `aes(y=..density..)` to create a frequency histogram. Create a plot with a histogram and a density plot.

```r
gg_3 + geom_histogram(aes(y=..density..),bins=7) +
  geom_density()
```

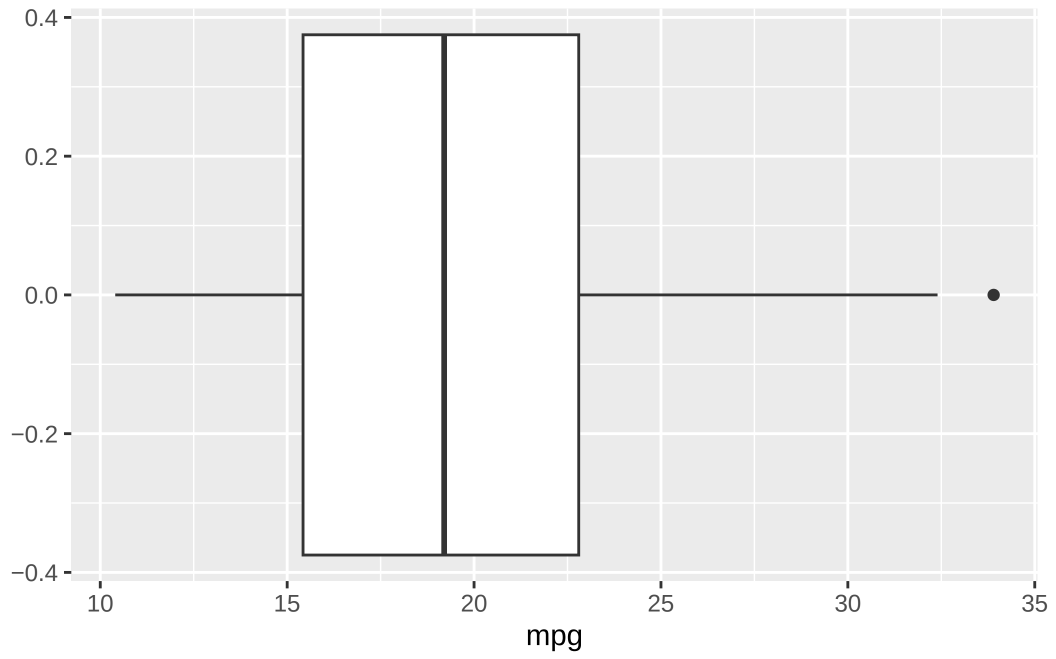
```
Warning: The dot-dot notation (`..density..`) was deprecated in ggplot2 3.4.0.
i Please use `after_stat(density)` instead.
```



19.2.5 Box Plots

If you need to create a box plot, use the `stat_boxplot()`. Create a boxplot for the variable `mpg`. All you need to do is add `stat_boxplot()`.

```
gg_3 + stat_boxplot()
```



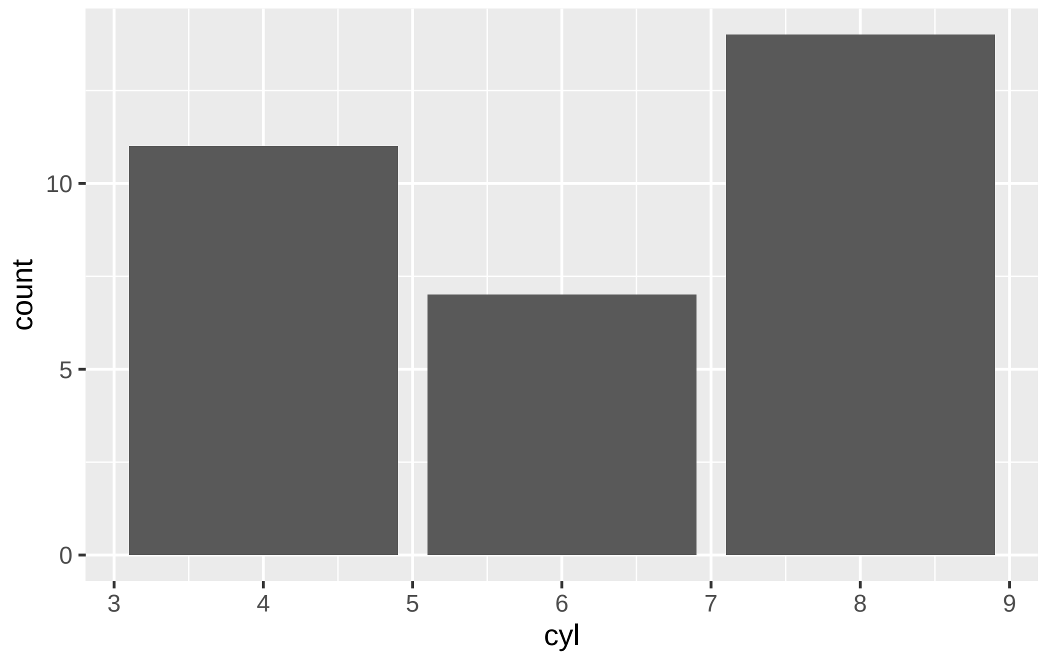
19.2.6 Bar Charts

Creating a bar chart is similar to create a box plot. All you need to do is use the `stat_count()`. First create a base plot using the `mtcars` data sets and the `cyl` variable for the mapping and assign it to `gg_4`.

```
gg_4 <- ggplot(mtcars, aes(cyl))
```

Now create the bar plot by adding the `stat_count()`.

```
gg_4 + stat_count()
```



19.2.7 Grouping

The 'ggplot2::" easily allows you to create plots from different groups. We will go over some of the arguments and functions to do this.

19.2.7.1 One Variable Grouping

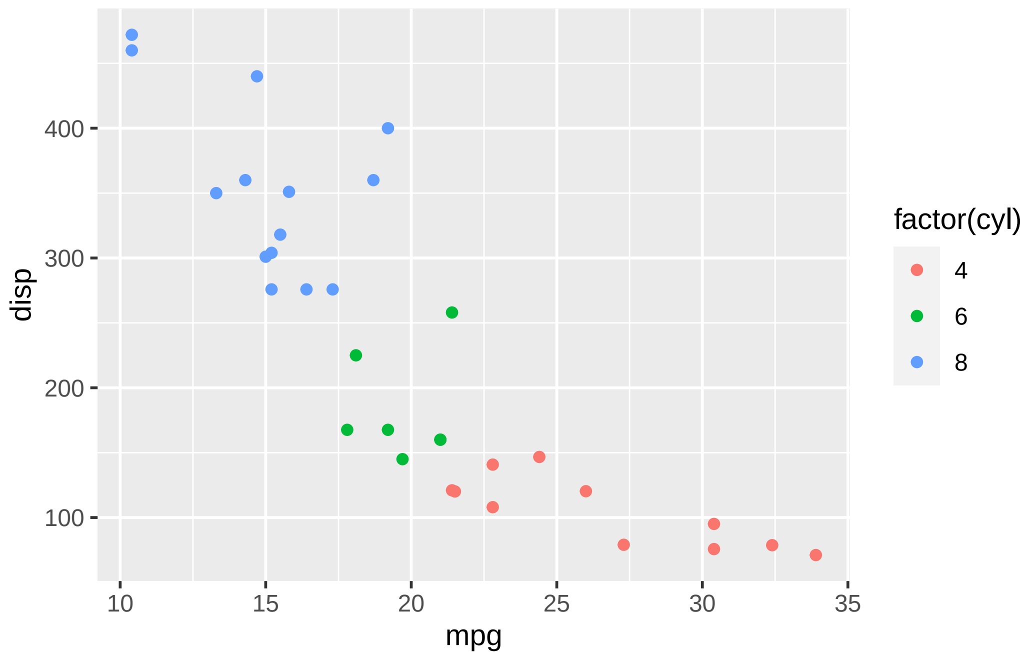
19.2.7.1.1 Scatter Plot

To begin, we want to specify the grouping variable within the `aes()` with the `color=`. Additionally, the argument works best with a factor variable, so use the `factor()` to create a factor variable. Create a base plot from the `mtcars` data set using `mpg` and `disp` for the x and y axis, respectively, and set the `color=` equal to the `factor(cyl)`. Assign it the R object `gg_5`.

```r
gg_5 <- ggplot(mtcars, aes(mpg, disp, color=factor(cyl)))
```

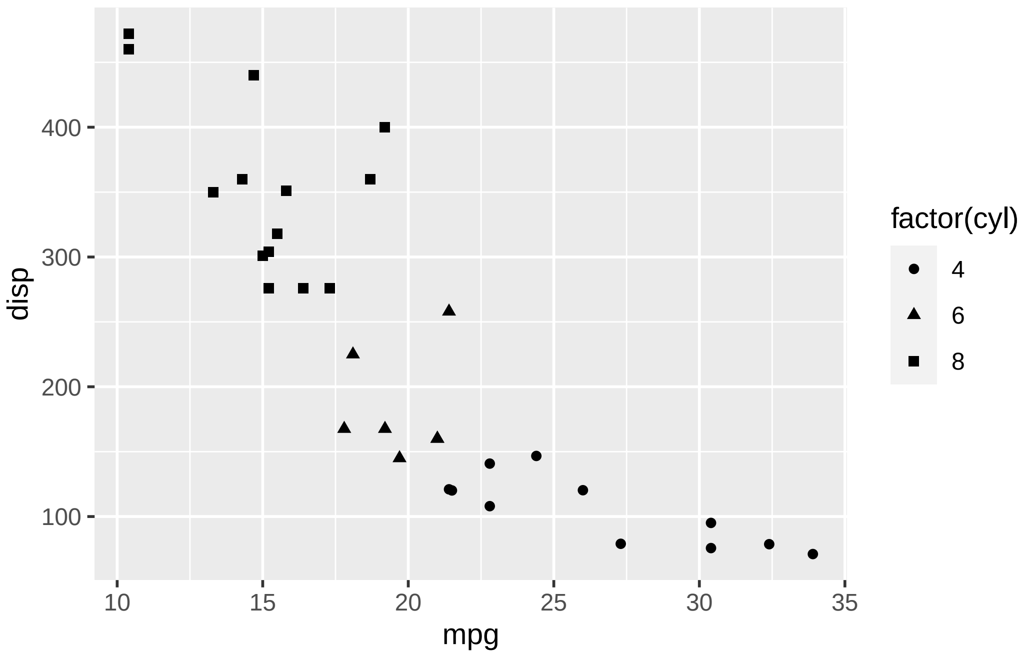
Once the base plot is created, 'ggplot2::" will automatically group the data in the plots. Create the scatter plot from the base plot.

```r
gg_5 + geom_point()
```



If you want to change the shapes instead of the color, use the `shape=`. Create a base plot from the `mtcars` data set using `mpg`, and `disp` for the x and y axis, respectively, and group it by `cyl` with the `shape=`. Assign it the R object `gg_6`.

```
gg_6 <- ggplot(mtcars, aes(mpg, disp, shape=factor(cyl)))
gg_6 + geom_point()
```



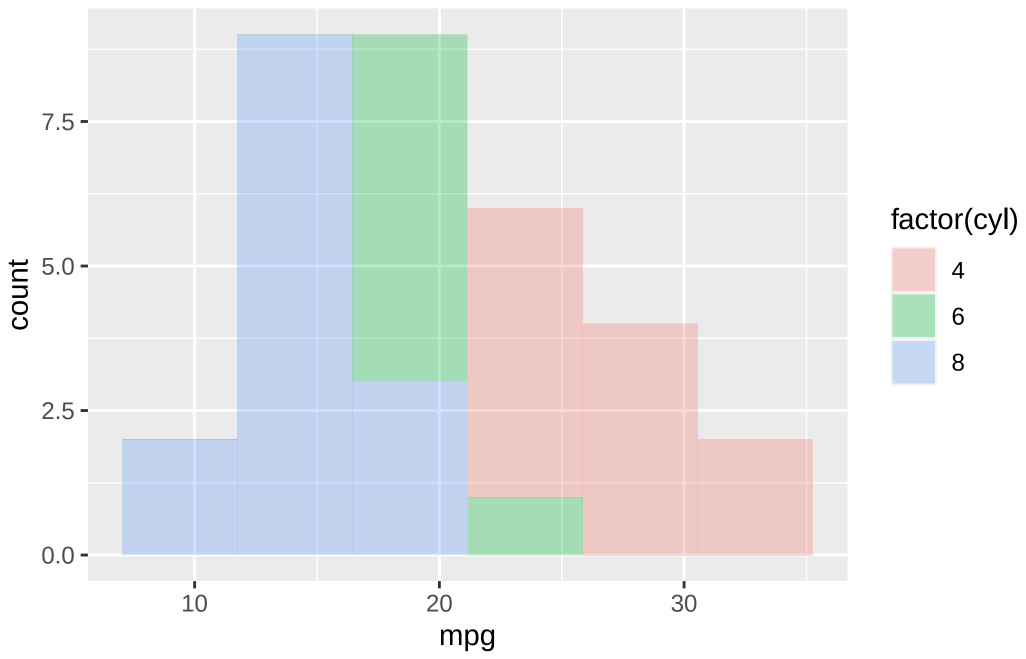
19.2.7.1.2 Histograms

Histograms can be grouped by different colors. This is done by using the `fill=` within the `aes()` in the base plot. Assign it the R object `gg_7`.

```
gg_7 <- ggplot(mtcars, aes(mpg, fill = factor(cyl)))
```

Now create a histogram from the base plot `gg_7`.

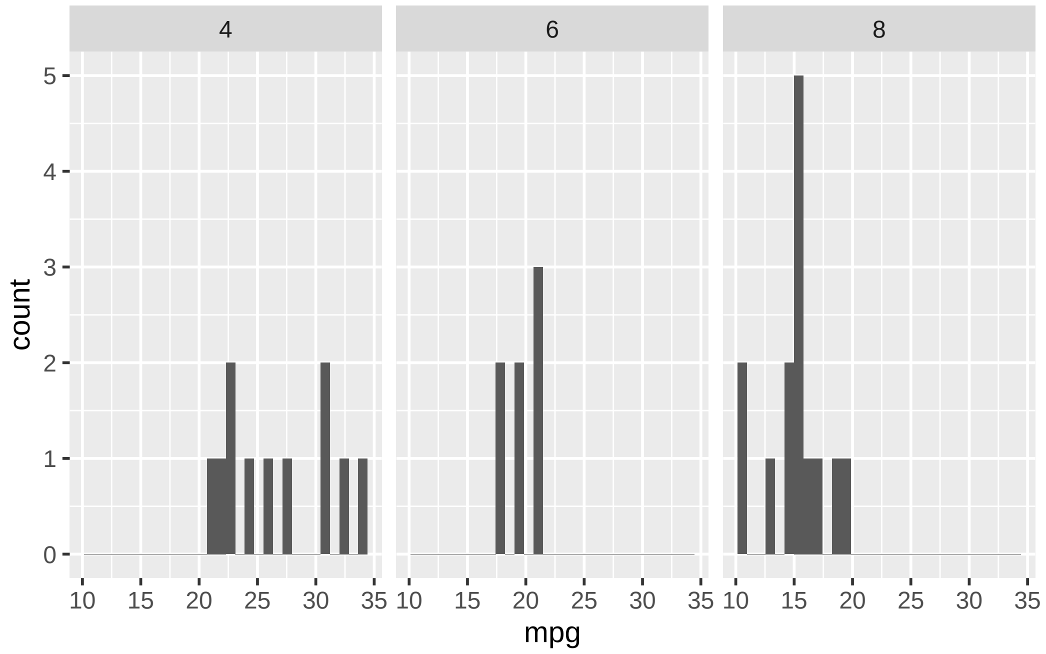
```
gg_7 + geom_histogram(bins = 6, alpha = 0.3)
```



Sometimes we would like to view the histogram on separate plots. The `facet_wrap()` and the `flact_grid()` allows this. Using either function, you do not need to specify the grouping factor in the `aes()`. You will add `facet_wrap()` to the plot. It needs a formula argument with the grouping variable. Using the R object `gg_3` create side by side plots using the `cyl` variable. Remember to add `geom_histogram()`.

```
gg_3+geom_histogram() + facet_wrap( ~ cyl)
```

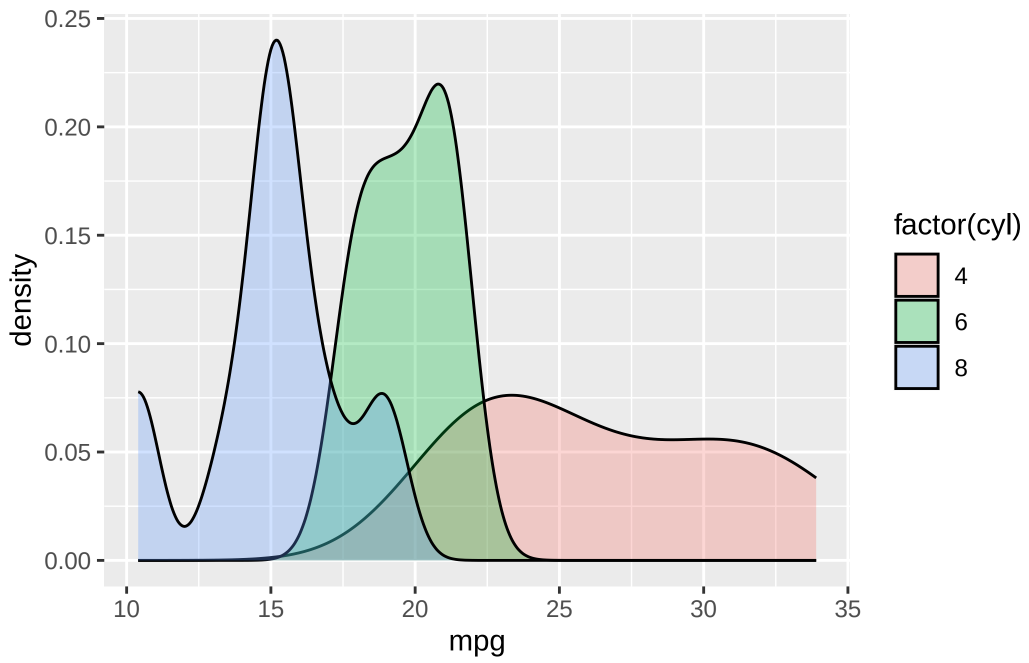
```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



19.2.7.1.3 Density Plot

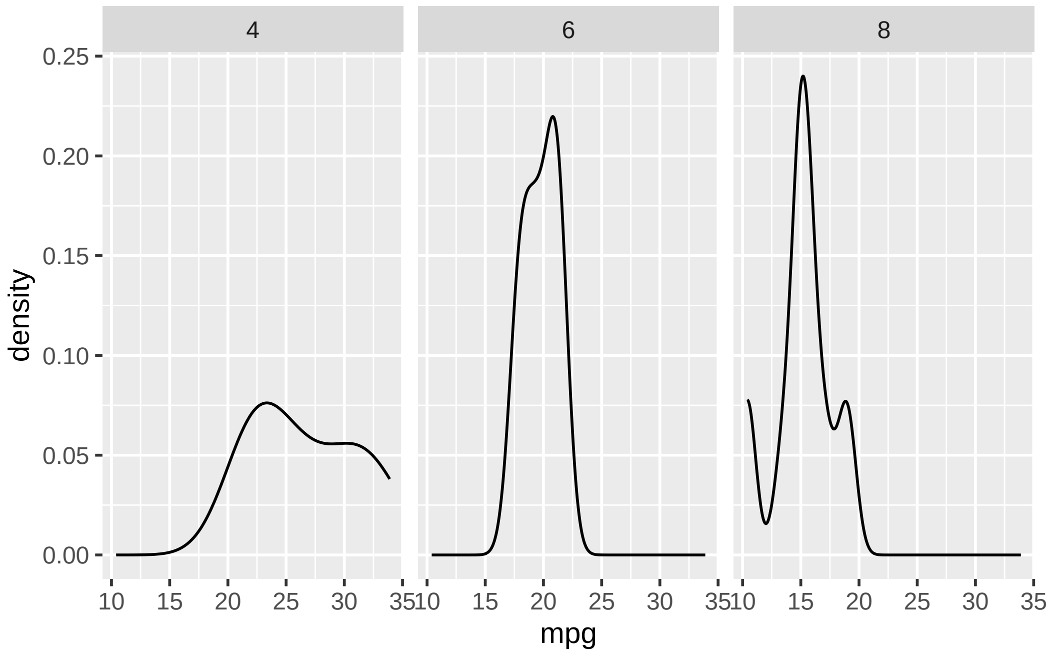
Similar to histograms, density plots can be grouped by variables the same way. Using `gg_7`, create color-coded density plots. All you need to do is add `geom_density()`.

```r
gg_7 + geom_density(alpha=0.3)
```



Using `gg_3`, create side by side density plots. You need to do is add `geom_density()` and `facet_wrap()` to group with the `cyl` variable.

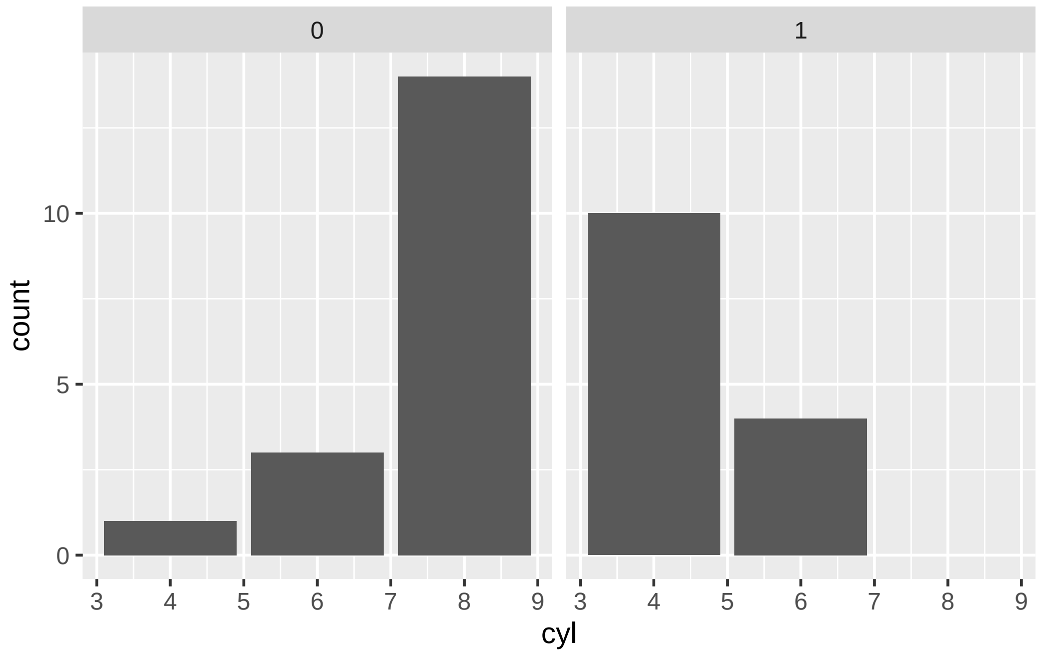
```
gg_3 + geom_density() + facet_wrap( ~ cyl)
```



19.2.7.1.4 Bar Chart

To create a side by side bar plot, you can use the facet_wrap() with a grouping variable. Using gg_4, create a side by side bar plot using vs as the grouping variable. Remember to add stat_count() as well.

```
gg_4 + stat_count() + facet_wrap(~vs)
```

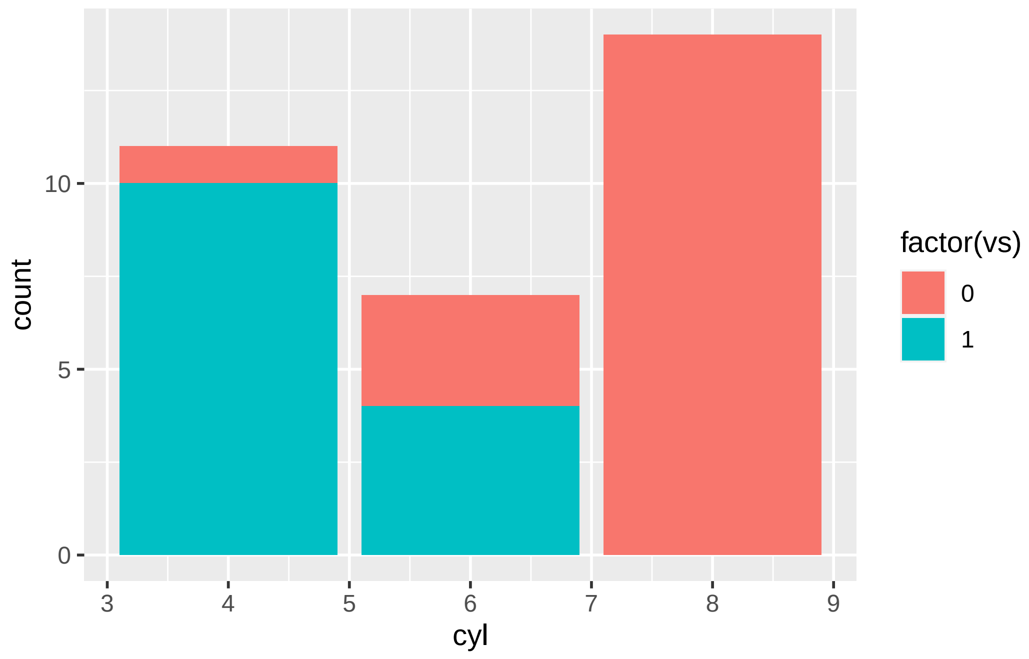


If you want to compare the bars from different group in one plot, you can use the fill= from the aes(). The fill= just needs a factor variable (use factor()). First create a base plot using the data mtcars, variable cyl and grouping variable vs. Assign it to gg_8.

```
gg_8 <- ggplot(mtcars, aes(cyl, fill = factor(vs)))
```

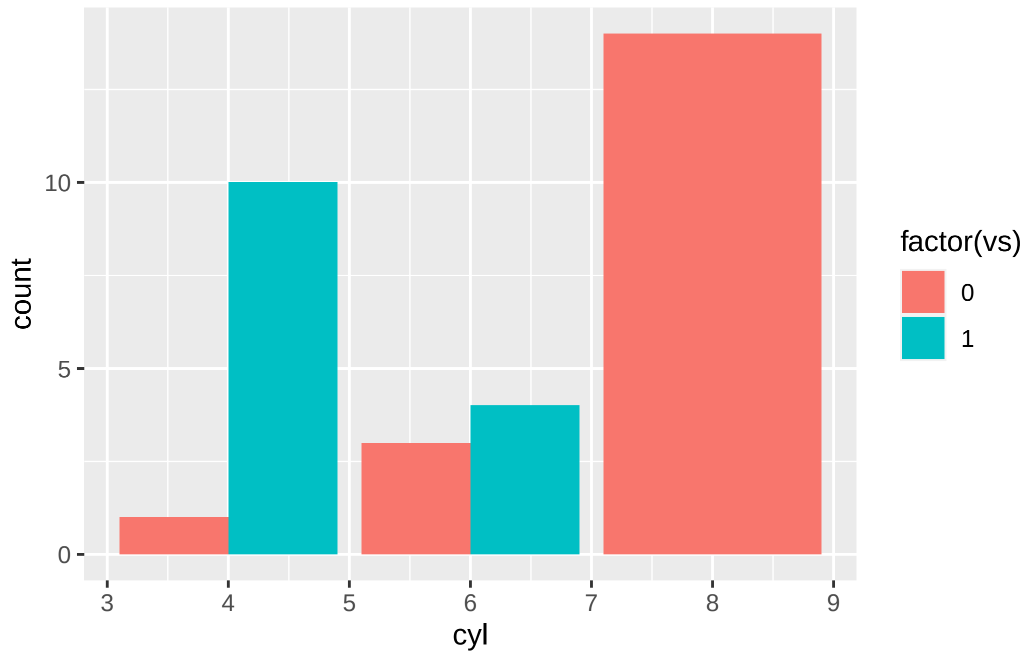
Now create a bar chart by adding stat_count().

```
gg_8 + stat_count()
```



If you want to grouping bars to be side by side, use the `position=` in the `stat_count()` and set it equal to `"dodge"`. Create the bar plot using the `position = "dodge"`.

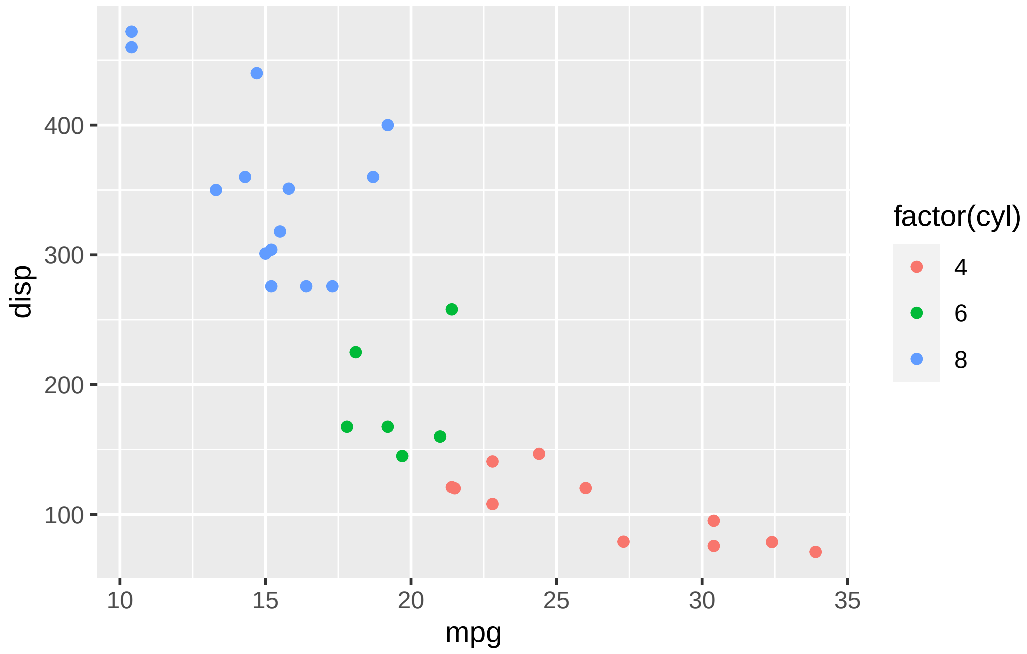
```
gg_8 + stat_count(position = "dodge")
```



19.2.8 Themes/Tweaking

In this section, we will talk about the basic tweaks and themes to `ggplot2::`. However. `ggplot2::` is much more powerful and can do much more. Before we begin, lets look at object `gg_9` to understand the plot. To view a plot, use the `plot()`.

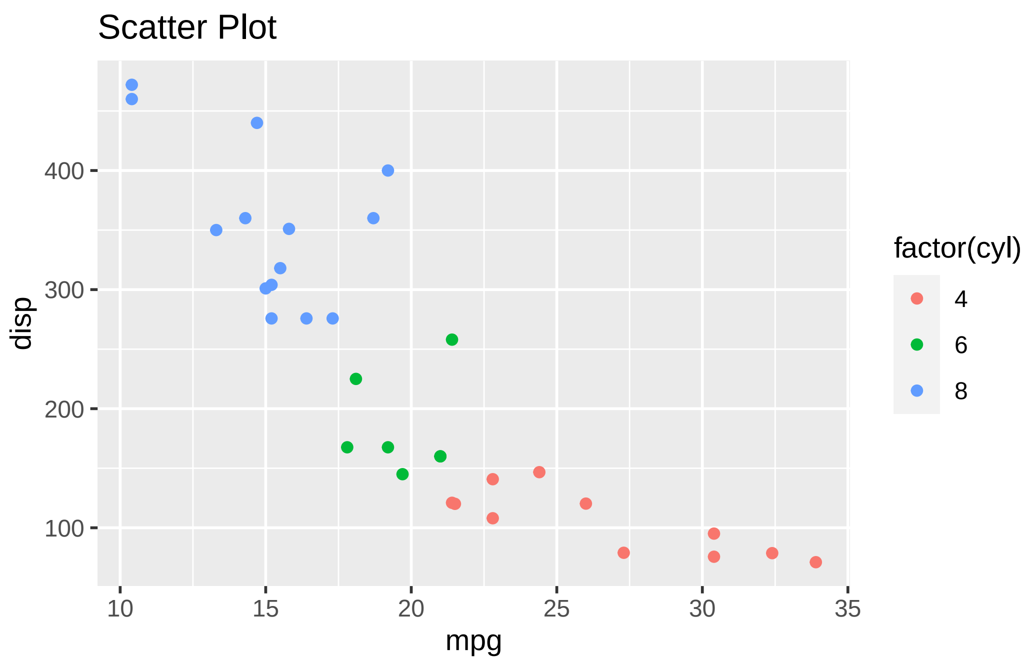
```
plot(gg_9)
```



19.2.8.1 Title

To change the title, add the `ggtitle()` to the plot. Put the new title in quotes as the first argument. Change the title for `gg_9`.

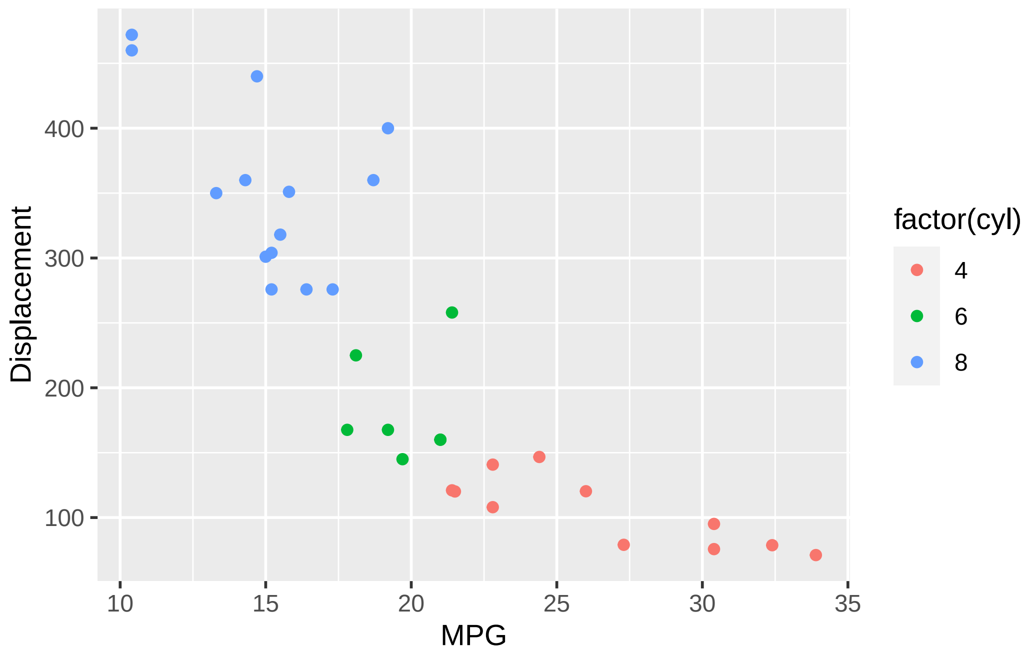
```
gg_9 + ggtitle("Scatter Plot")
```



19.2.8.2 Axis

Changing the labels for a plot, add the `xlab()` and `ylab()`, respectively. The first argument contains the phrase for the axis. Change the axis labels for `gg_9`.

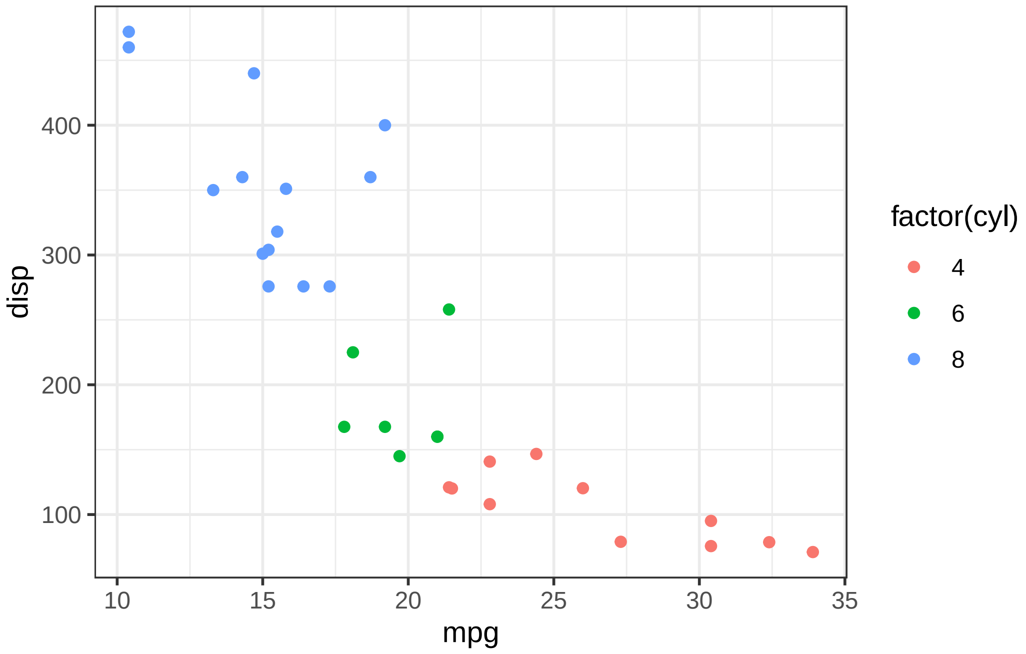
```
gg_9 + xlab("MPG") + ylab("Displacement")
```



19.2.8.3 Themes

If you don't like how the plot looks, `ggplot2::` has custom themes you can add to the plot to change it. These functions usually are formatted as `theme_*()`, where the `*` indicates different possibilities. I personally like how `theme_bw()` looks. Change the theme of `gg_9`.

```
gg_9 + theme_bw()
```



Additionally, you can change certain part of the theme using the `theme()`. I encourage you to look at what are other possibilities.

19.2.9 Saving plot

If you want to save the plot, use the `ggsave()`. Read the help documentation for the functions capabilities.

Part VII

Reporting Data

20 Markdown Reports

21 Presentations

Part VIII

Debugging and Efficient Porgramming

22 Debugging Code

23 Efficient Programming and Profiling

24 Vectorizing Code

25 Introduction to Rcpp