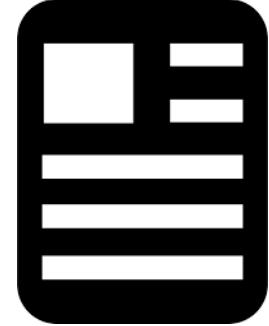


# Introduction to R in Data Science





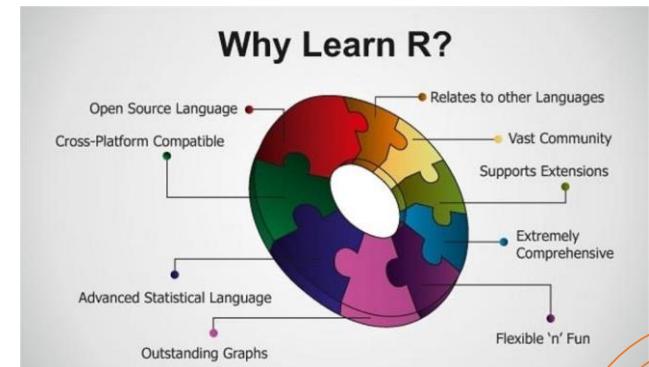
# Table of Contents



1. Introduction to R: 3

1. Data Visualization and Graphics: 109

1. Data Management: 216



# 1. Introduction to R



# Introduction to R

By the end of this lesson, you will be able to:

- Install R packages for use throughout the course
- Use R as a calculator for basic arithmetic
- Utilize different data structures
- Control program flow by writing if-else, for, and while loops
- Import and export data to and from CSV, Excel, and SQL



# Using R and RStudio, and Installing Useful Packages

- R is a programming language intended for use for statistical analysis.
- Additionally, it can be utilized in an object-oriented or functional way.
- Specifically, it is an implementation of S, an interactive statistical programming language.
- R was initially released in August 1993. It is maintained today by the R Development Core Team.



# Using R and RStudio

- Out of the box, R is completely usable.
- Open R on your machine.
- Let's use R for some basic arithmetic such as addition, multiplication, subtraction, and division.
- The following screenshot demonstrates this:

```
> 10 + 2  
[1] 12  
> 5 * 5  
[1] 25  
> 98 - 14  
[1] 84  
> 8 / 2  
[1] 4  
>
```

# Using R and RStudio

- It also provides functions such as `sum()` and `sqrt()` for addition and calculation of the square root.
- The following screenshot shows this in action:

```
> sqrt(25)
[1] 5
> sum(5, 6, 7, 8, 9)
[1] 35
>
```



# Executing Basic Functions in the R Console

Let's now try and execute the `sum()` and `sqrt()` functions in R. Follow the steps given below:

- Open the R console on your system.
- Type the code as follows:

```
sum(1, 2, 3, 4, 5)  
sqrt(144)
```



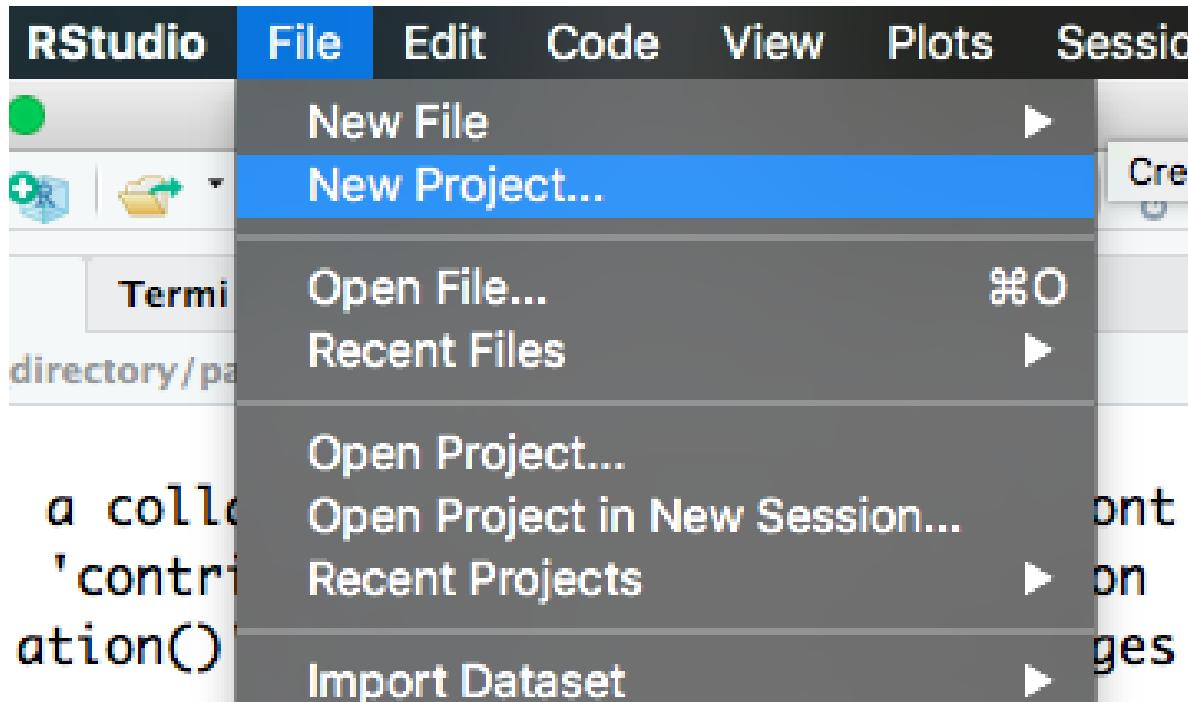
# Executing Basic Functions in the R Console

- Output: The preceding code provides the following output:

```
[1] 15  
[1] 12
```



# Executing Basic Functions in the R Console



# Executing Basic Functions in the R Console

New Project

Create Project

---

 **New Directory**  
Start a project in a brand new working directory >

---

 **Existing Directory**  
Associate a project with an existing working directory >

---

 **Version Control**  
Checkout a project from a version control repository >

---

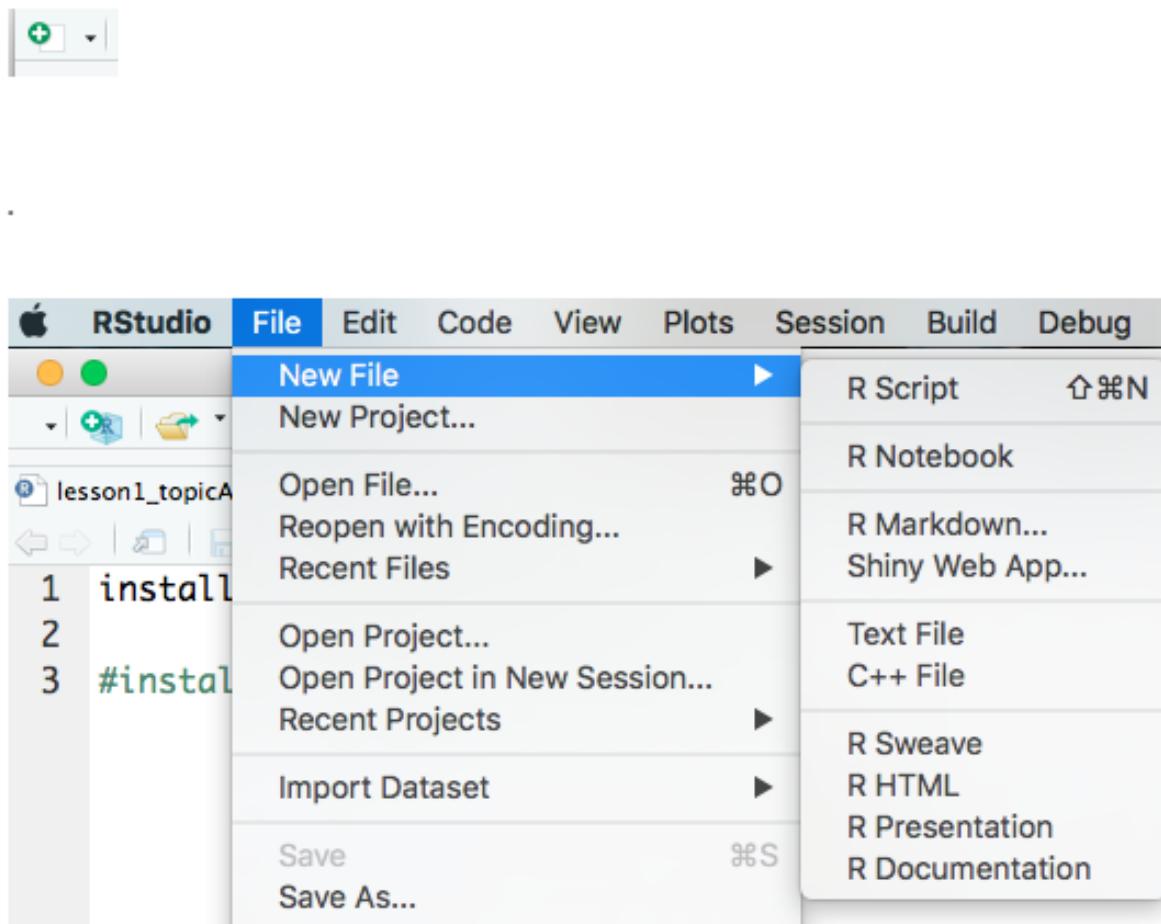
[Cancel](#)

# Executing Basic Functions in the R Console

- To verify the working directory at any time, use the `getwd()` function.
- It will print the working directory as a character string (you can tell because it has quotation marks around it).
- The working directory can be changed at any time by using the following syntax:

```
setwd("new location/on the/computer")
```





# Executing Basic Functions in the R Console

- Custom functions are fairly straightforward to create in R.
- Generally, they should be created using the following syntax:

```
name_of_function <- function(input1, input2){  
  operation to be performed with the inputs  
}
```

- The example custom function is as follows:

```
area_triangle <- function(base, height){  
  0.5 * base * height  
}
```



# Executing Basic Functions in the R Console

- Once the custom function code has been run, it will display in the Global Environment in the upper right corner and is now available for use in your RStudio project, as shown in the following screenshot:

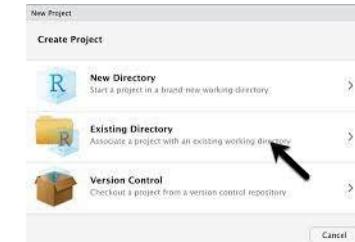
The screenshot shows the RStudio interface with the Global Environment tab selected. Below the tabs, there are several icons: a folder with a green arrow, a blue document, an 'Import Dataset' button, a paintbrush, a 'List' dropdown, and a 'C' icon. A search bar is also present. The main pane is titled 'Functions' and lists two entries:

area_rectang...	function (length, width)
area_triangle	function (base, height)

# Setting up a New Project

- Write a custom function, `area_rectangle()`, which calculates the area of a rectangle, with the following code:

```
area_rectangle <- function(length, width){  
  length * width  
}
```



# Setting up a New Project

Try out `area_rectangle()` with the following sets of lengths and widths:

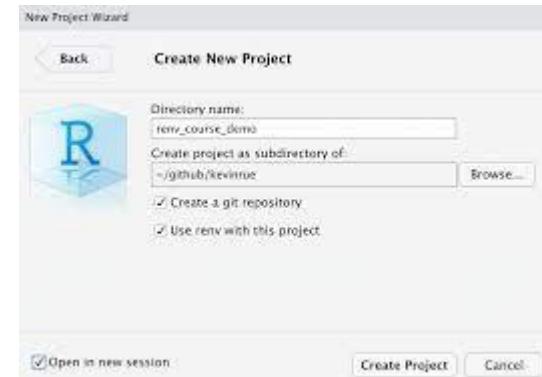
- 5, 10
- 80, 7
- 48209302930, 4

The code will be as follows:

`area_rectangle(5, 10)`

`area_rectangle(80, 70)`

`area_rectangle(48209302930, 4)`



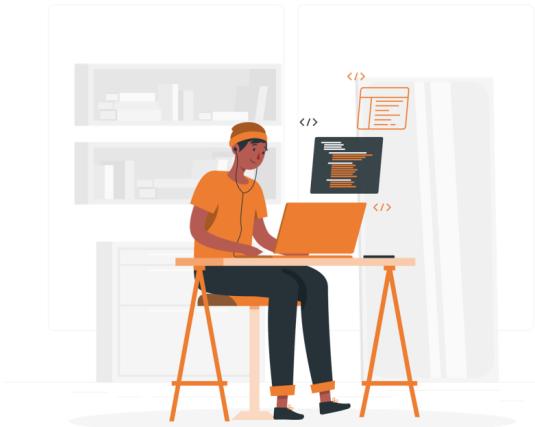
# Setting up a New Project

- The area of the rectangle with different lengths will be provided as follows:

[1] 50

[1] 5600

[1] 192837211720



# Installing Packages

Install Packages

Install from: [? Configuring Repositories](#)

Repository (CRAN)

Packages (separate multiple with space or comma):

mice

Install to Library:

/Library/Frameworks/R.framework/Versions/3.4/Resources/libi

Install dependencies

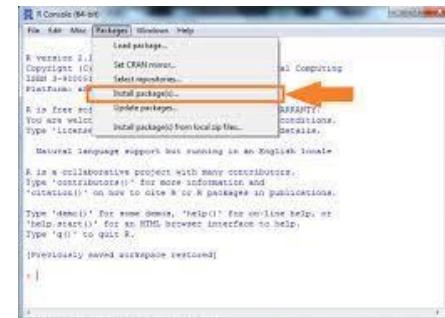
Install Cancel

# Installing Packages

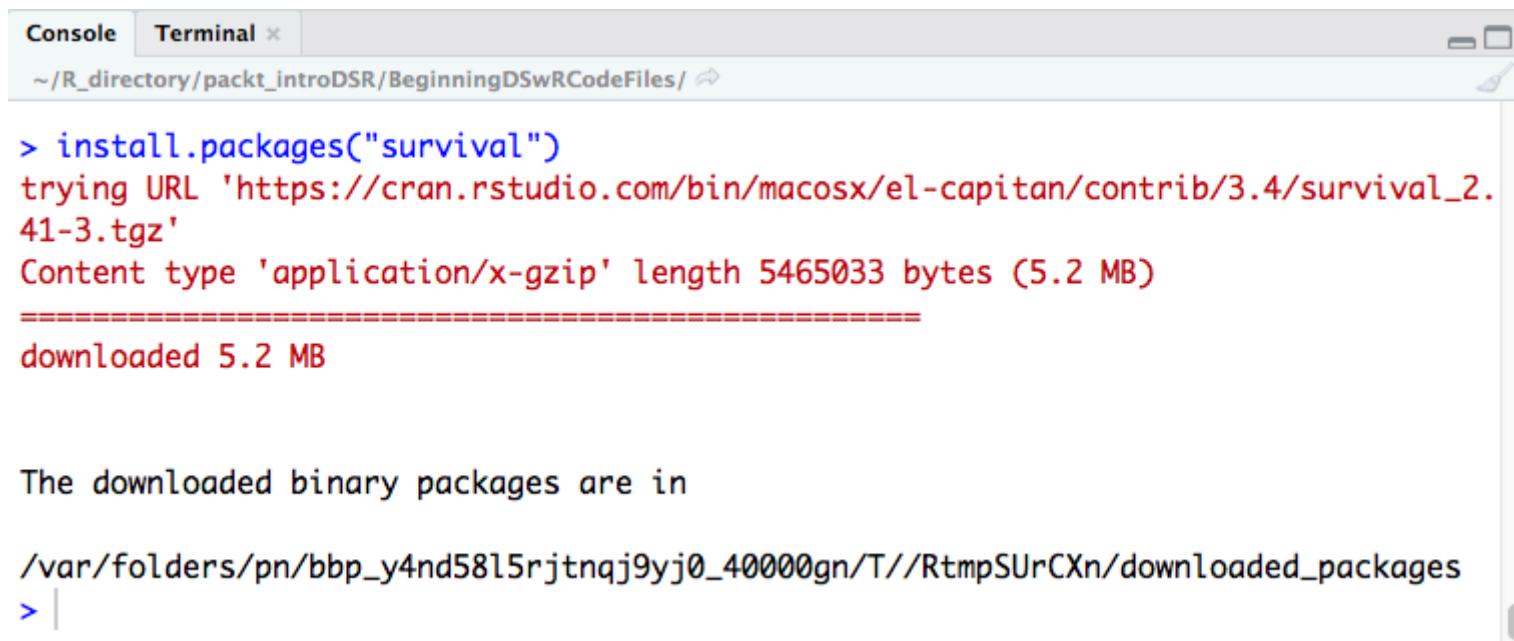
Let's now use two different methods to install R packages. Follow these steps:

- Install the survival package using the following code:

```
install.packages("survival")
```



# Installing Packages



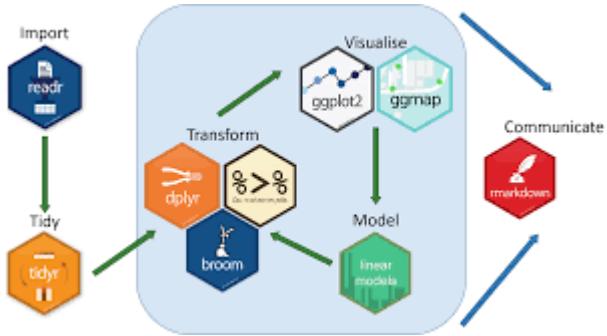
The screenshot shows the RStudio interface with the 'Console' tab selected. The current working directory is displayed as `~/R_directory/packt_introDSR/BEGINNINGDSWRCodeFiles/`. The console output shows the command `> install.packages("survival")` being run, followed by the download process details: URL, content type, length (5.2 MB), and the final message "downloaded 5.2 MB". Below this, a message states "The downloaded binary packages are in" followed by the path `/var/folders/pn/bbp_y4nd58l5rjtnqj9yj0_40000gn/T//RtmpSURCXn/downloaded_packages`.

```
Console Terminal ×
~/R_directory/packt_introDSR/BEGINNINGDSWRCodeFiles/ ⌂

> install.packages("survival")
trying URL 'https://cran.rstudio.com/bin/macosx/el-capitan/contrib/3.4/survival_2.41-3.tgz'
Content type 'application/x-gzip' length 5465033 bytes (5.2 MB)
=====
downloaded 5.2 MB

The downloaded binary packages are in

/var/folders/pn/bbp_y4nd58l5rjtnqj9yj0_40000gn/T//RtmpSURCXn/downloaded_packages
> |
```



# Complete Activity: Installing the Tidyverse Packages



# Variable Types and Data Structures

## Variable Types

- Variable types exist in all programming languages and will tell the computer how to store and access a variable.
- First, know that all variables created in R will have a class and a type.
- You can look at the class or type of anything in R using the `class()` and `typeof()` functions, respectively.

# Variable Types and Data Structures

- Let's examine the following code snippet:

```
x <- 4.2  
class(x)  
typeof(x)
```



- The preceding code provides the following output:

```
[1] "numeric"  
[1] "double"
```

# Numeric and Integers

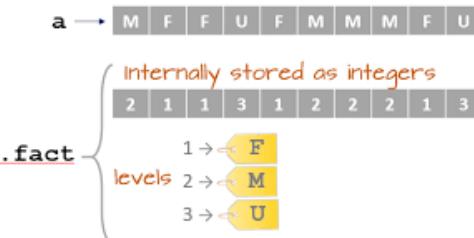
Let's now create and check the class() and typeof() of different numeric objects in R. Follow the steps given below:

- Create the following numeric objects:

`x <- 12.7`

`y <- 8L`

`z <- 950`



# Numeric and Integers

- Check the class and type of each using `class()` and `typeof()`, respectively, as follows:

`class(x)`

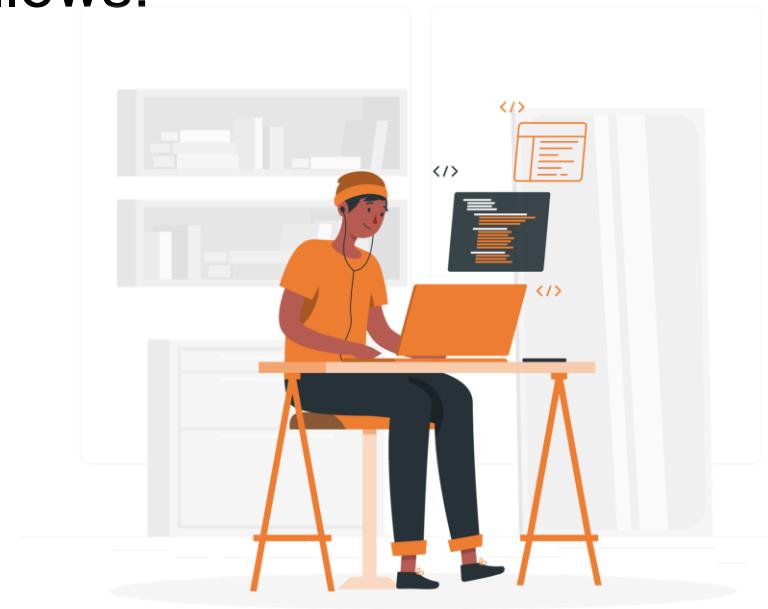
`typeof(x)`

`class(y)`

`typeof(y)`

`class(z)`

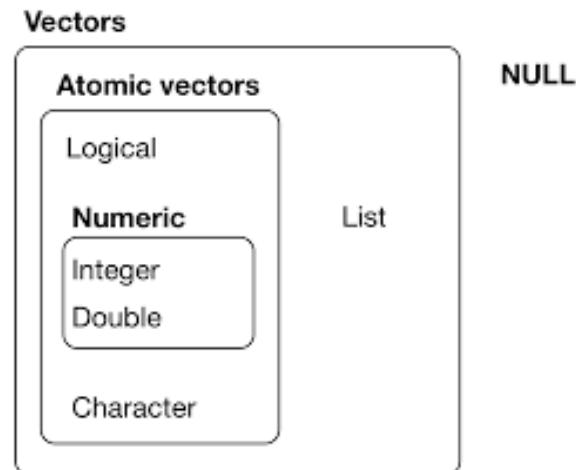
`typeof(z)`



# Numeric and Integers

- Output: The preceding code provides the following output:

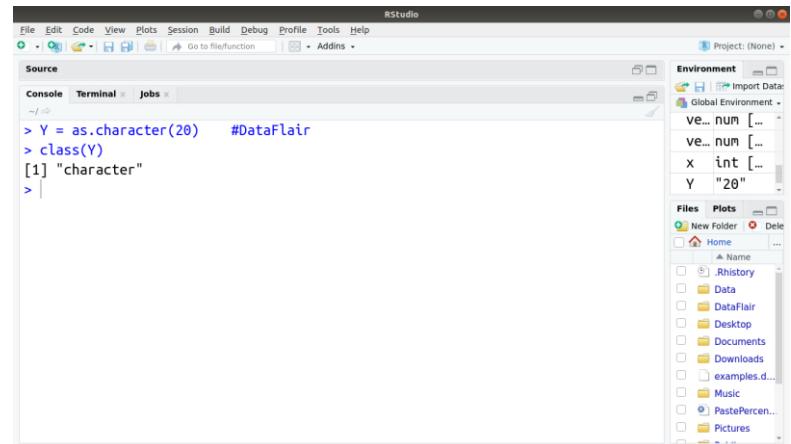
```
[1] "numeric"  
[1] "double"  
[1] "integer"  
[1] "integer"  
[1] "numeric"  
[1] "double"
```



# Character

Let's create and check the class() and typeof() of different character objects in R. Follow the steps given below:

- Create the following objects:  
a <- "apple"  
b <- "7"  
c <- "9-5-2016"



The screenshot shows the RStudio interface. In the top menu bar, 'File', 'Edit', 'Code', 'View', 'Plots', 'Session', 'Build', 'Debug', 'Profile', 'Tools', and 'Help' are visible. Below the menu is a toolbar with various icons. The main area has tabs for 'Console', 'Terminal', and 'Jobs'. The 'Console' tab is active, displaying the following R session:

```
> Y = as.character(20) #DataFlair
> class(Y)
[1] "character"
>
```

To the right of the console is the 'Environment' pane, which lists objects: 've...', 've...', 'x', and 'Y'. The 'Y' object is highlighted. Below the environment is the 'Files' pane, showing a file tree with 'Rhistory', 'Data', 'Desktop', 'Documents', 'Downloads', 'examples.d...', 'Music', 'PastePercen...', and 'Pictures'. The 'Name' column is expanded.

# Character

- Check the class and type of each using `class()` and `typeof()`, respectively, as follows:

`class(a)`

`typeof(a)`

`class(b)`

`typeof(b)`

`class(c)`

`typeof(c)`



# Numeric and Integers

- Output: The preceding code provides the following output:

```
[1] "character"  
[1] "character"  
[1] "character"  
[1] "character"  
[1] "character"  
[1] "character"  
[1] "character"
```



# Dates

Let's create and check the class() and typeof() of different date objects in R. Follow these steps:

- Create the objects using the following code:

```
e <- as.Date("2016-09-05")  
f <- as.POSIXct("2018-04-05")
```

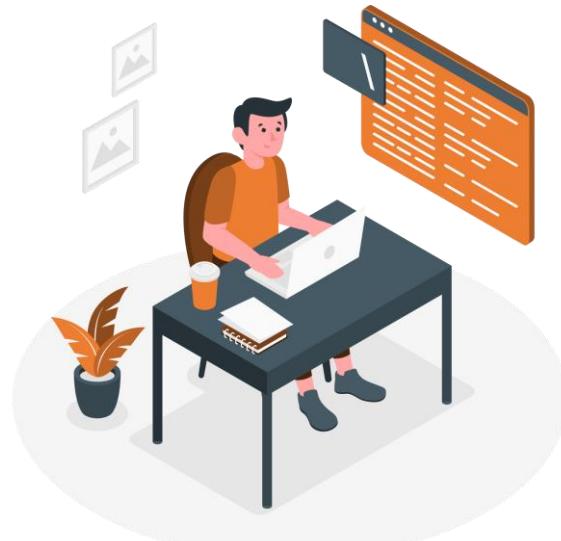


# Dates

- Check the class and type of each by using `class()` and `typeof()`, respectively, as follows:

`class(e)`  
`typeof(e)`

`class(f)`  
`typeof(f)`



# Dates

- Output: The preceding code provides the following output:

```
[1] "Date"  
[1] "double"  
[1] "POSIXct" "POSIXt"  
[1] "double"
```



# Dates

```
#char to numeric, integer
```

```
var <- "5"
```

```
var_num <- as.numeric(var)
```

```
class(var_num)
```

```
typeof(var_num)
```

```
var_int <- as.integer(var)
```

```
class(var_int)
```

```
typeof(var_int).
```



R for Data  
Science

# Dates

- Conversely, we can go the other way and cast the var\_num and var\_int variables back to the character data type using as.character().
- The following code demonstrates this:

```
#numeric, integer to char
```

```
var <- 5
```

```
#numeric to char
```

```
var_char <- as.character(var_num)
```

```
class(var_char)
```

```
typeof(var_char)
```

```
#int to char
```

```
var_char2 <- as.character(var_int)
```

```
class(var_char2)
```

```
typeof(var_char2)
```



# Dates

- A character string can be converted into a Date, but it does need to be in the format Year-Month-Day (Y-M-D) so that you can use the `as.Date()` function, as shown in the following code:

```
#char to date  
date <- "18-03-29"  
Date <- as.Date(date)  
class(Date)  
typeof(Date)
```

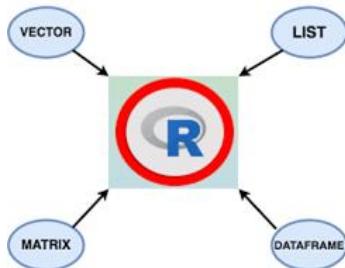


# Dates

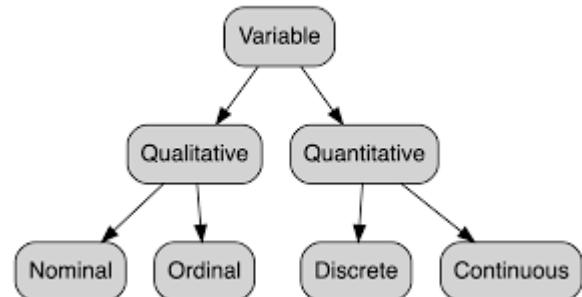
- There are formatting requirements for dates for them to save correctly.
- For example, the following code will not work:  
`date2 <- as.Date("03-29-18")`

- It will throw the following error:

Error in charToDate(x) : character string is not in a standard unambiguous format



# Complete Activity: Identifying Variable Classes and Types



# Data Structures

- There are a few different data structures in R that are crucial to understand, as they directly pertain to the use of data! These include vectors, matrices, and dataframes.
- We'll discuss how to tell the difference between all of these, along with how to create and manipulate them.

# Vectors

- A vector is an object that holds a collection of various data elements in R, though they are limited because everything inside of a vector must all belong to the same variable type.
- You can create a vector using the method `c()`, for example:

```
vector_example <- c(1, 2, 3, 4)
```



# Vectors

- In the following code, we can see an example where the class of the vector is a character because of the B in position 2:

```
vector_example_2 <- c(1, "B", 3)  
class(vector_example_2)
```

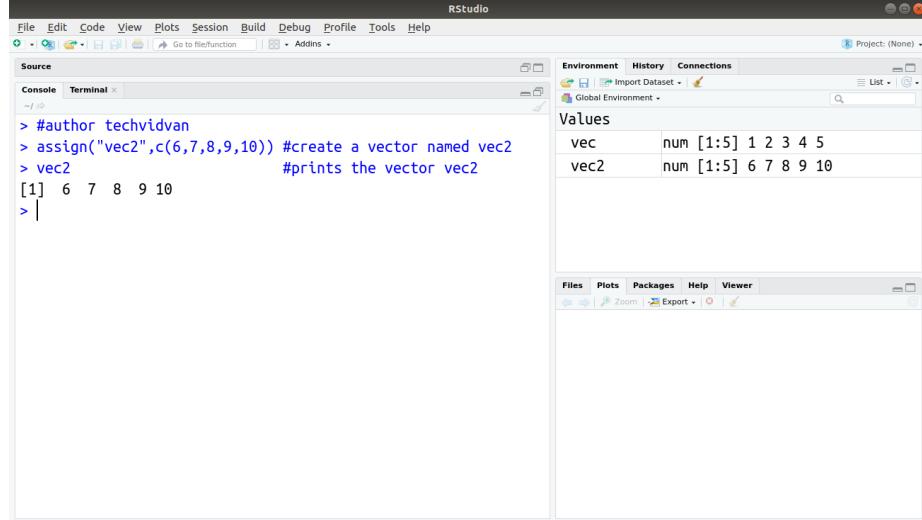
- Output: The preceding code provides the following output:

```
[1] "character"
```

# Vectors

- Create the vectors twenty and alphabet using the following code:

twenty <- c(1:20)  
alphabet <- c(letters)



The screenshot shows the RStudio interface with the following details:

- Console Tab:** Displays R code and its output.

```
> #author techvidvan
> assign("vec2",c(6,7,8,9,10)) #create a vector named vec2
> vec2
[1] 6 7 8 9 10
> |
```
- Environment Tab:** Shows the created objects and their values.

Values
vec num [1:5] 1 2 3 4 5
vec2 num [1:5] 6 7 8 9 10
- Plots Tab:** Empty.
- Packages Tab:** Empty.
- Help Tab:** Empty.
- Viewer Tab:** Empty.

# Vectors

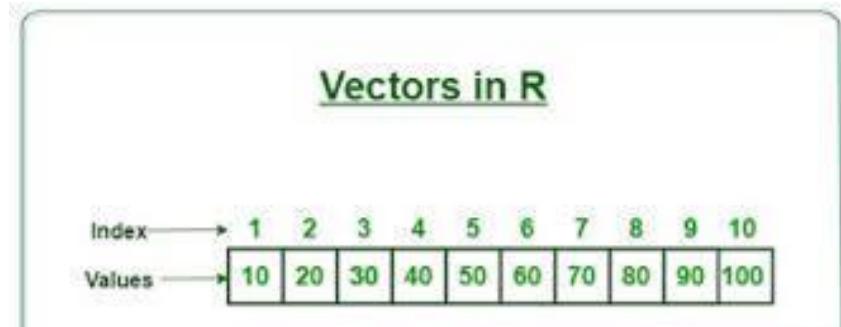
- Check the class and type of twenty and alphabet using class() and typeof(), respectively, as follows:

`class(twenty)`

`typeof(twenty)`

`class(alphabet)`

`typeof(alphabet)`



# Vectors

- Find the numbers at the following positions in twenty using vector indexing:

twenty[5]

twenty[17]

twenty[25]

- Find the letters at the following positions in the alphabet using vector indexing:

alphabet[6]

alphabet[23]

alphabet[33]



R Programming

- Output: The code we write will be as follows:

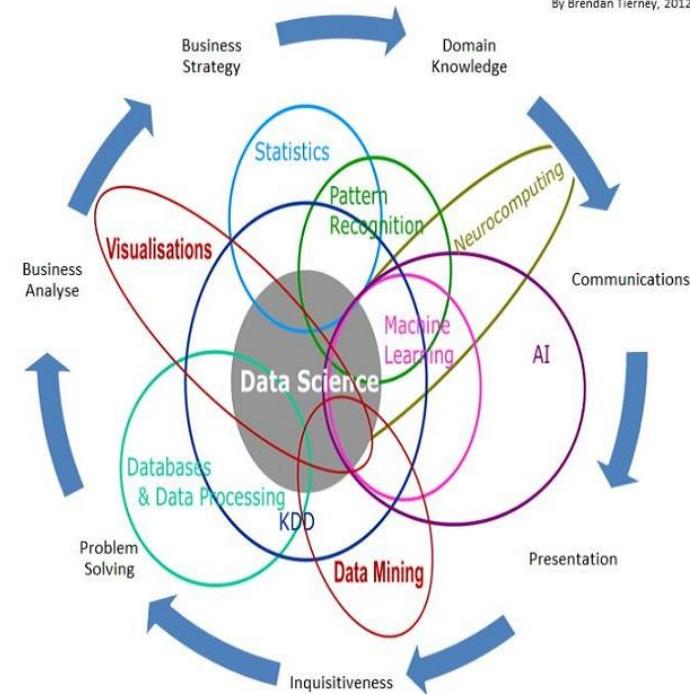
```

twenty <- c(1:20)
alphabet <- c(letters)
class(twenty)
typeof(twenty)
class(alphabet)
typeof(alphabet)
twenty[5]
twenty[17]
twenty[25]
alphabet[6]
alphabet[23]
alphabet[33]

```

## Data Science Is Multidisciplinary

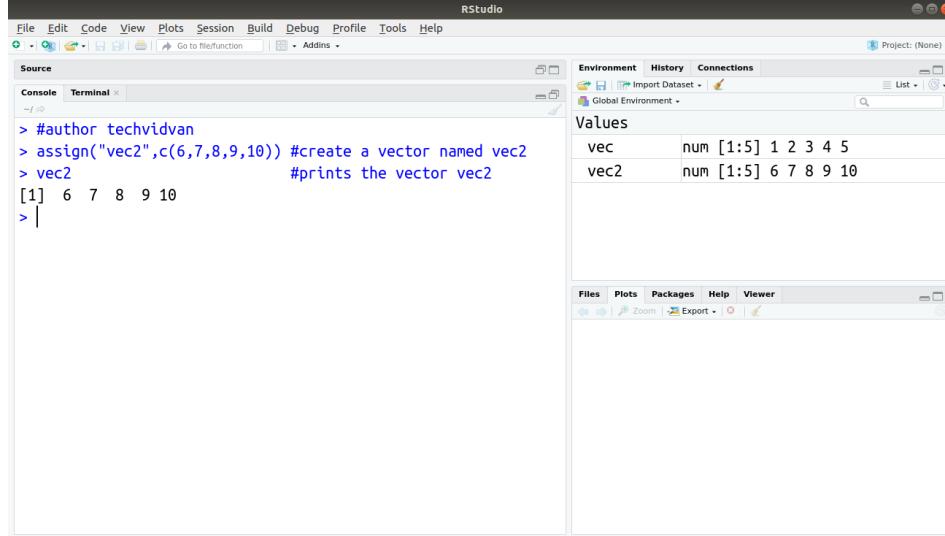
By Brendan Tierney, 2012



# Vectors

- The output we get after executing it is as follows:

```
[1] "integer"  
[1] "integer"  
[1] "character"  
[1] "character"  
[1] 5  
[1] 17  
[1] NA  
[1] "f"  
[1] "w"  
[1] NA
```



The screenshot shows the RStudio interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. The main area has tabs for Source and Terminal, with the Terminal tab active. The terminal window contains the following R session:

```
> #author techvidvan  
> assign("vec2",c(6,7,8,9,10)) #create a vector named vec2  
> vec2                         #prints the vector vec2  
[1] 6 7 8 9 10  
>
```

To the right of the terminal is the Environment pane, which displays two objects:

Values
vec num [1:5] 1 2 3 4 5
vec2 num [1:5] 6 7 8 9 10

Below the Environment pane are tabs for Files, Plots, Packages, Help, and Viewer.

# Lists

- A list is different from a vector because it can hold many different types of R objects inside it, including other lists.
- If you have experience programming in another language, you may be familiar with lists, but if not, don't worry! You can create a list in R using the `list()` function, as shown in the following example:

```
L1 <- list(1, "2", "Hello", "cat", 12, list(1, 2, 3))
```

# Lists

- The following screenshot shows the output of this code:

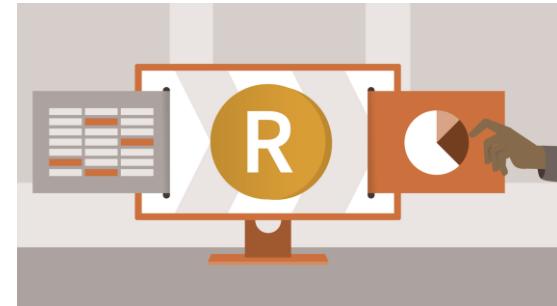
```
> L1 <- list(1, "2", "Hello", "cat", 12, list(1, 2, 3))
>
> L1[[1]]
[1] 1
> L1[[4]]
[1] "cat"
> L1[[6]][1]
[[1]]
[1] 1
```



# Lists

- Lists can also be changed into other data structures. We could turn a list into a dataframe, but this particular list, because it contains a nested list, will not coerce to a vector.
- The following code demonstrates this:

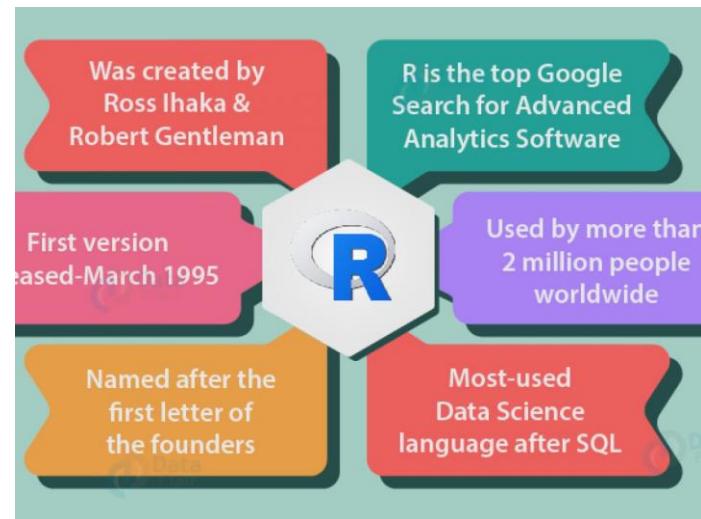
```
L1_df <- as.data.frame(L1)  
class(L1_df)  
L1_vec <- as.vector(L1)  
class(L1_vec)
```



# Lists

- The following screenshot shows the output of this code:

```
> L1_df <- as.data.frame(L1)
> class(L1_df)
[1] "data.frame"
> L1_vec <- as.vector(L1)
> class(L1_vec)
[1] "list"
```



# Matrices

- Use `matrix()` to create `matrix1`, a  $3 \times 3$  matrix containing the numbers 1:12 by column, using the following code:

```
matrix1 <- matrix(c(1:12), nrow = 3, ncol = 3, byrow = FALSE)
```

- Create `matrix2` similarly, also  $3 \times 3$ , and fill it with 1:12 by row, using the following code:

```
matrix2 <- matrix(c(1:12), nrow = 3, ncol = 3, byrow = TRUE)
```

# Matrices

- Set the row and column names of matrix1 with the following:

```
rownames(matrix1) <- c("one", "two", "three")  
colnames(matrix1) <- c("one", "two", "three")
```



# Matrices

- Find the elements at the following positions in matrix1 using matrix indexing:

matrix1[1, 2]

matrix1["one", ]

matrix1[, "one"]

matrix1["one", "one"]

- The output of the code is as follows:

```
> #matrix indexing  
> matrix1[1, 2]  
[1] 4  
> matrix1["one", ]  
one   two three  
     1      4      7
```

# Dataframes

- Dataframes can be created by using `as.data.frame()` on applicable objects or by column- or row-binding vectors using `cbind.data.frame()` or `rbind.data.frame()`.
- Here's an example where we can create a list of nested lists and turn it into a data frame:

```
list_for_df <- list(list(1:3), list(4:6), list(7:9))
```

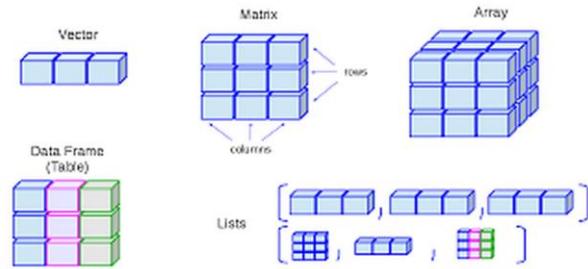
```
example_df <- as.data.frame(list_for_df)
```



# Dataframes

- `example_df` will have three rows and three columns.
- We can set the column names just as we did with the matrix, though it isn't common practice in R to set the row names for most analyses.
- It can be demonstrated by the following code:

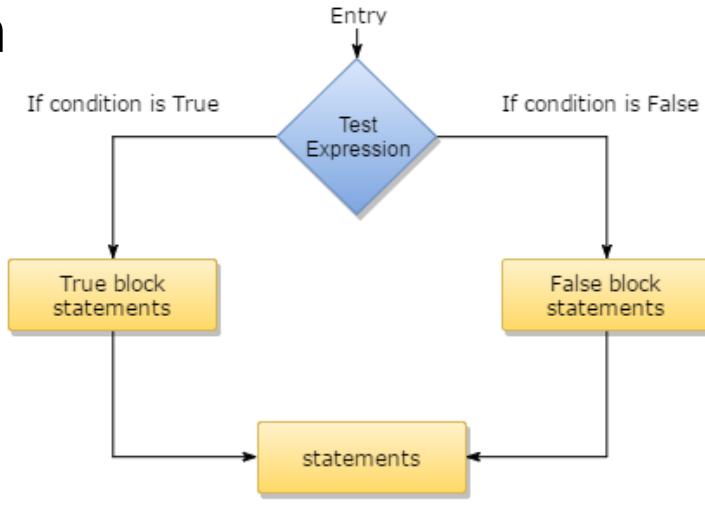
```
colnames(example_df) <- c("one", "two", "three")
```



# Complete Activity: Creating Vectors, Lists, Matrices, and Dataframes

# Basic Flow Control

- Flow Control includes different kinds of loops that you can use in R, such as the if/else, for, and while loops.
- While many of the concepts are very similar to how flow control and loops are used in other programming languages, they may be written differently in R.



Flow Chart of Flow Control Statements

# If/else

- The if loop will only run a block of code if a certain condition is TRUE. It can be paired with else statements to create an if/else loop.
- This will work similarly to an if/else loop in other programming languages, though the syntax may be different.
- The usual syntax for using if is as follows:

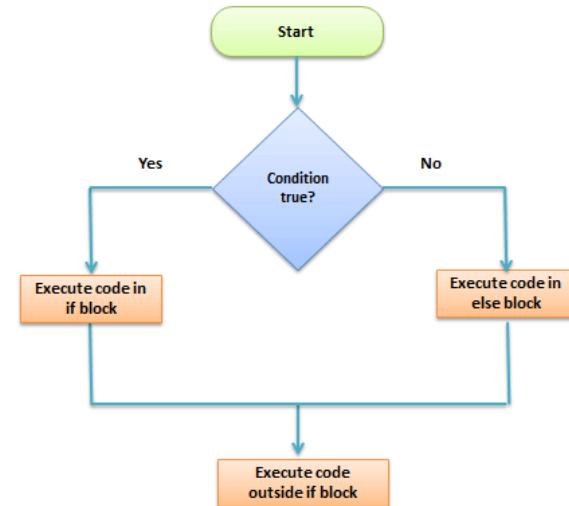
```
if(test_condition){  
    some_action  
}
```



# If/else

- If there's something you want to happen, even if the test condition isn't true, you would use an if/else, where the syntax usually looks like this:

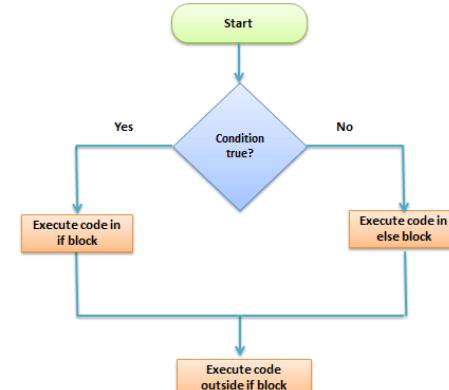
```
if(test_expression){  
    some_action  
}else{  
    some_other_action  
}
```



# If/else

- Even if the test\_expression isn't true, some\_other\_action will still happen.
- Finally, you can evaluate multiple test conditions with if/else if/else, as shown in the following syntax:

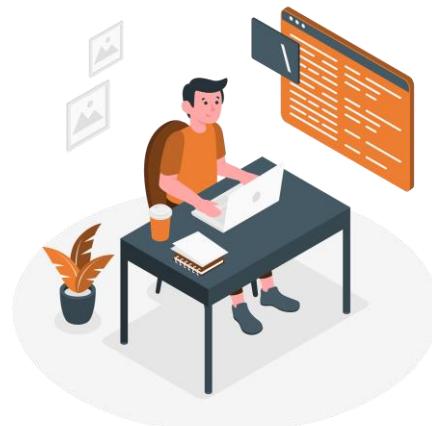
```
if(test_expression){  
    some_action  
}else if(another_test_expression){  
    some_other_action  
}else{  
    yet_another_action  
}
```



# If/else

- Let's do some actual examples to help illustrate these points.
- Take a look at the following code:

```
var <- "Hello"  
if(class(var) == "character"){  
  print("Your variable is a character string.")  
}
```



# If/else

- With the following code, when var is 5, something will print:

```
var <- 5
if(class(var) == "character"){
  print("Your variable is a character string.")
}else{
  print("Your variable is not a character")
}
```



# If/else

- Because we specified else, we will see the output "Your variable is not a character." This isn't very informative, however, so let's expand and use an else if:

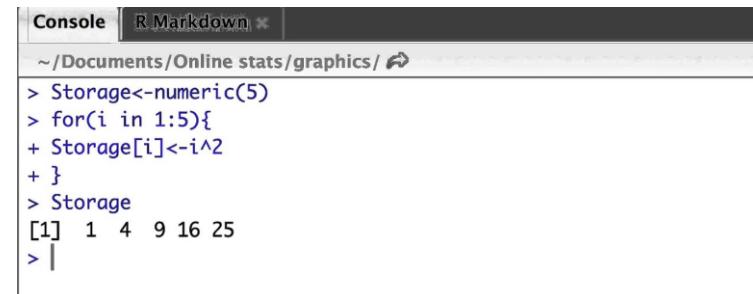
```
if(class(var) == "character"){
  print("Your variable is a character string.")
}else if (class(var) == "numeric"){
  print("Your variable is numeric")
}else{
  print("Your variable is something besides character or numeric.")
}
```



# For loop

- For loops are often used to go through every column or row of a dataframe in R.
- If we simply want to print the means to the console, we could use a for loop as follows:

```
for(i in seq_along(iris)){
  print(mean(iris[[i]]))
}
```



The screenshot shows the RStudio interface with the 'Console' tab selected. The code `for(i in seq\_along(iris)){ print(mean(iris[[i]]))}` is entered and executed. The output shows the mean of each column: [1] 1 4 9 16 25.

```
Console | R Markdown * | ~ /Documents/Online stats/graphics/ ↗
> Storage<-numeric(5)
> for(i in 1:5){
+   Storage[i]<-i^2
+ }
> Storage
[1] 1 4 9 16 25
> |
```

# For loop

- The output will be as follows:

```
> for(i in seq_along(iris)){
+   print(mean(iris[[i]]))
+ }
[1] 5.843333
[1] 3.057333
[1] 3.758
[1] 1.199333
[1] NA
```

Warning message:

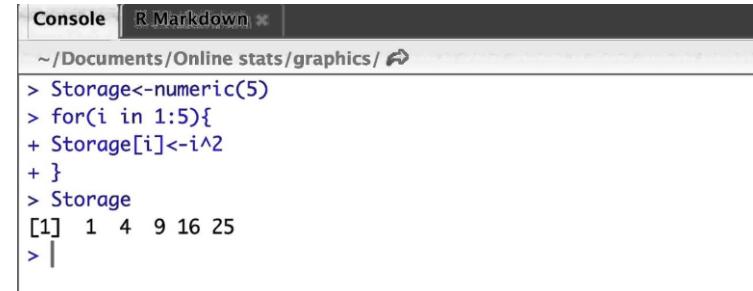
In mean.default(iris[[i]]) :  
argument is not numeric or logical: returning NA



# For loop

- We'll come back to the output, especially that warning message, in a second—first, let's break down the components of the for loop.
- The syntax will always be as follows:

```
for(i in a range of numbers){  
  some_action  
}
```



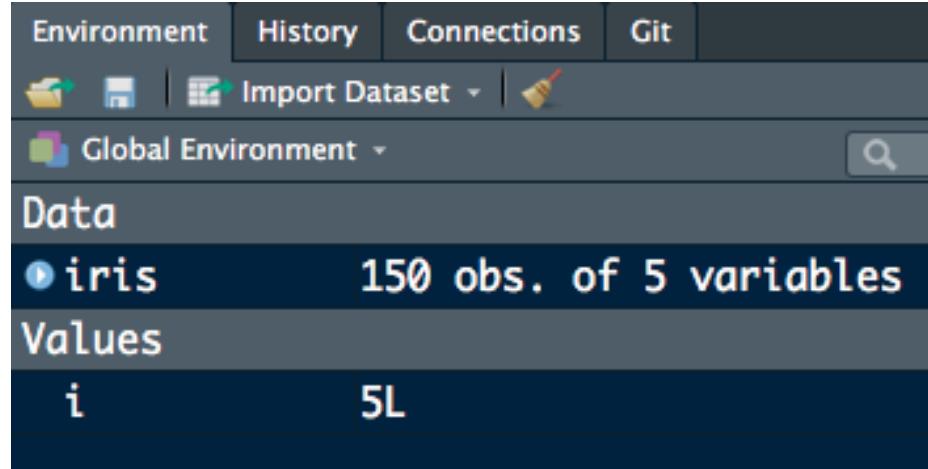
A screenshot of the RStudio interface showing a console window. The console tab is selected, and the R Markdown tab is visible. The code in the console is:

```
~/Documents/Online stats/graphics/  
> Storage<-numeric(5)  
> for(i in 1:5){  
+   Storage[i]<-i^2  
+ }  
> Storage  
[1] 1 4 9 16 25  
> |
```

The output shows a numeric vector [1] 1 4 9 16 25.

# For loop

- It is displayed on the screen, as shown in the following screenshot:



# For loop

- This is actually a great example of where we can combine for loops with an if statement.
- Take a look at the following code:

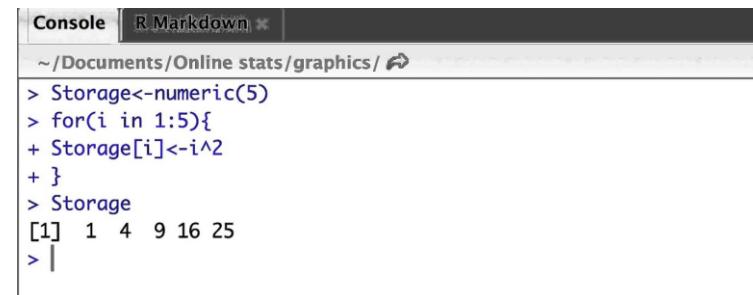
```
for(i in seq_along(iris)){
  if(class(iris[[i]]) == "numeric"){
    print(mean(iris[[i]]))
  }
}
```



# For loop

- The if statement here will only print the mean of an iris column if the class of that column is numeric (which makes sense, since only numeric columns should have means!) The output is now only as follows:

```
[1] 5.843333  
[1] 3.057333  
[1] 3.758  
[1] 1.199333
```



A screenshot of the RStudio interface showing the Console tab. The console window displays R code and its output. The code creates a numeric vector 'Storage' of length 5, initializes it to zero, and then uses a for loop to square each element. The resulting vector 'Storage' is printed, showing values 1, 4, 9, 16, and 25.

```
Console | R Markdown * |  
~/Documents/Online stats/graphics/  
> Storage<-numeric(5)  
> for(i in 1:5){  
+ Storage[i]<-i^2  
+ }  
> Storage  
[1] 1 4 9 16 25  
> |
```

# For loop

- If we're really feeling fancy, we could have even added an else statement with a different message for when the class of a column isn't numeric, such as in this loop:

```
for(i in seq_along(iris)){
  if(class(iris[[i]]) == "numeric"){
    print(mean(iris[[i]]))
  }else{
    print(paste("Variable", i, "isn't numeric"))
  }
}
```



# For loop

- The output is as follows:

```
[1] 5.843333  
[1] 3.057333  
[1] 3.758  
[1] 1.199333  
[1] "Variable 5 isn't numeric"
```



# For loop

- The following code will print every row of the Species column in iris:

```
for(i in 1:nrow(iris)){  
  print(iris[i, "Sepal.Width"])  
}
```



# For loop

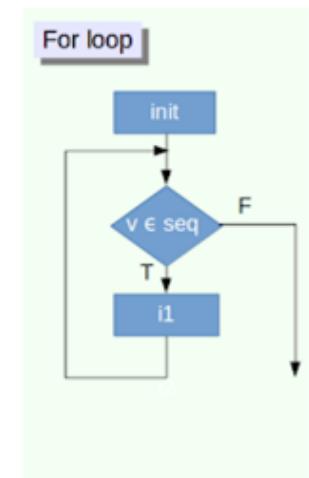
- nrow() simply returns the number of rows of iris versus the entire sequence of the number of columns that seq\_along() returns as shown below:

nrow(iris)

[1] 150

seq\_along(iris)

[1] 1 2 3 4 5



# While loop

- It requires you to add a line of code inside the body of the loop that increments or decrements your iterator, usually i.
- Generally, the syntax for a while loop is as follows:

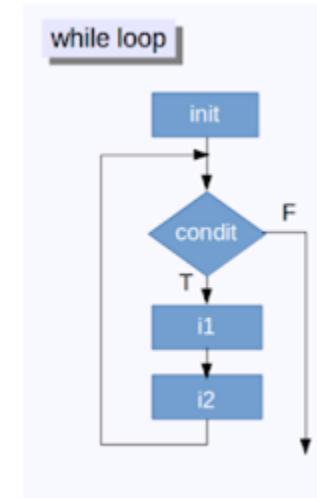
```
while(test_expression){  
    some_action  
}
```



# While loop

- A classic example of a while loop is one that prints out numbers, such as the following:

```
i = 0
while(i <= 5){
    print(paste("loop", i))
    i = i + 1
}
```



# While loop

- The output of the preceding code is as follows:

```
[1] "loop 0"  
[1] "loop 1"  
[1] "loop 2"  
[1] "loop 3"  
[1] "loop 4"  
[1] "loop 5"
```



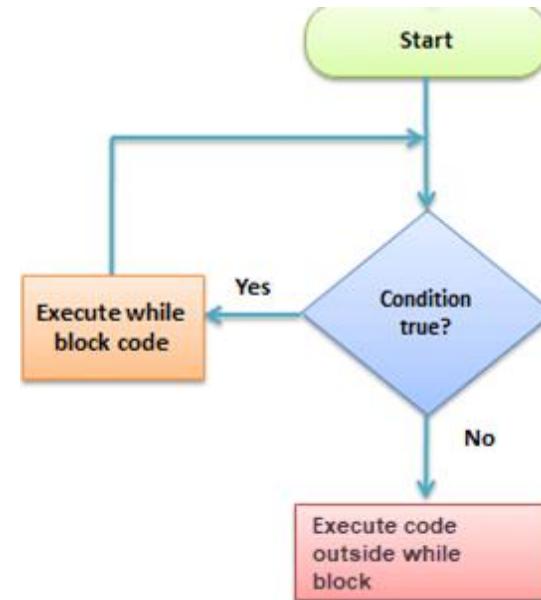
# While loop

- If the while loop test expression is never FALSE, the loop will never stop, as shown in the following code:

```
while(TRUE){  
    print("yes!")  
}
```

- The output will be as follows:

```
[1] "yes!"  
[1] "yes!"  
[1] "yes!"  
[1] "yes!"  
.....
```



# While loop

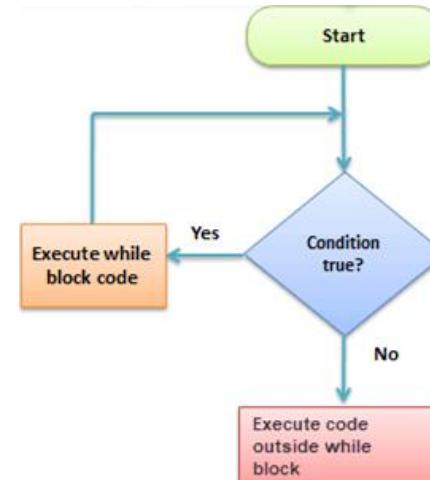
- If you're concerned about them, R does have a break statement, which will jump out of the while loop, but you'll see the following error:

Error: no loop for break/next, jumping to top level

# While loop

- For example, if we forgot that i is in our global environment, and that it equals 5, the following loop will never run:

```
while(i < 5){  
  print(paste(i, "is this number"))  
  i = i + 1  
}
```



# While loop

- Examine the following code snippet.
- Try to predict what the output will be:

```
vec <- seq(1:10)
for(num in seq_along(vec)){
  if(num %% 2 == 0){
    print(paste(num, "is even"))
  } else{
    print(paste(num, "is odd"))
  }
}
```



# While loop

- Examine the following code snippet.
- Try to predict what the output will be:

```
example <- data.frame(color = c("red", "blue", "green"),  
number = c(1, 2, 3))  
for(i in seq(nrow(example))){  
print(example[i,1])  
}
```



# While loop

- Examine the following code snippet.
- Try to predict what the output will be:

```
var <- 5
while(var > 0){
  print(var)
  var = var - 1
}
```



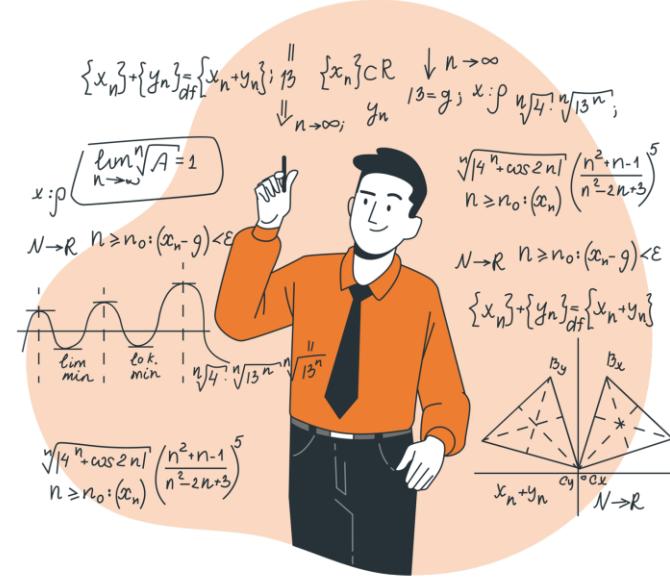
```
[1] "1 is odd"
[1] "2 is even"
[1] "3 is odd"
[1] "4 is even"
[1] "5 is odd"
[1] "6 is even"
[1] "7 is odd"
[1] "8 is even"
[1] "9 is odd"
[1] "10 is even"
```

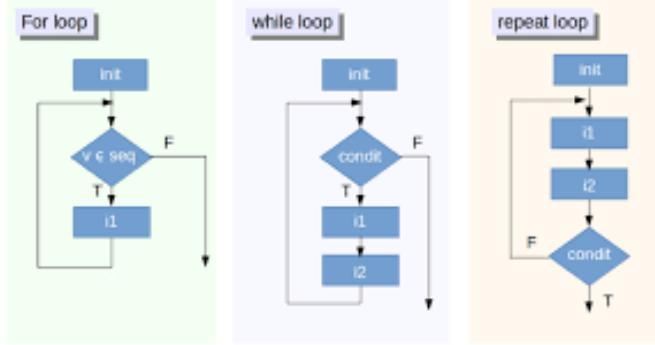
The output for the second step will be as follows:

```
[1] red
Levels: blue green red
[1] blue
Levels: blue green red
[1] green
Levels: blue green red
```

The output for the third step will be as follows:

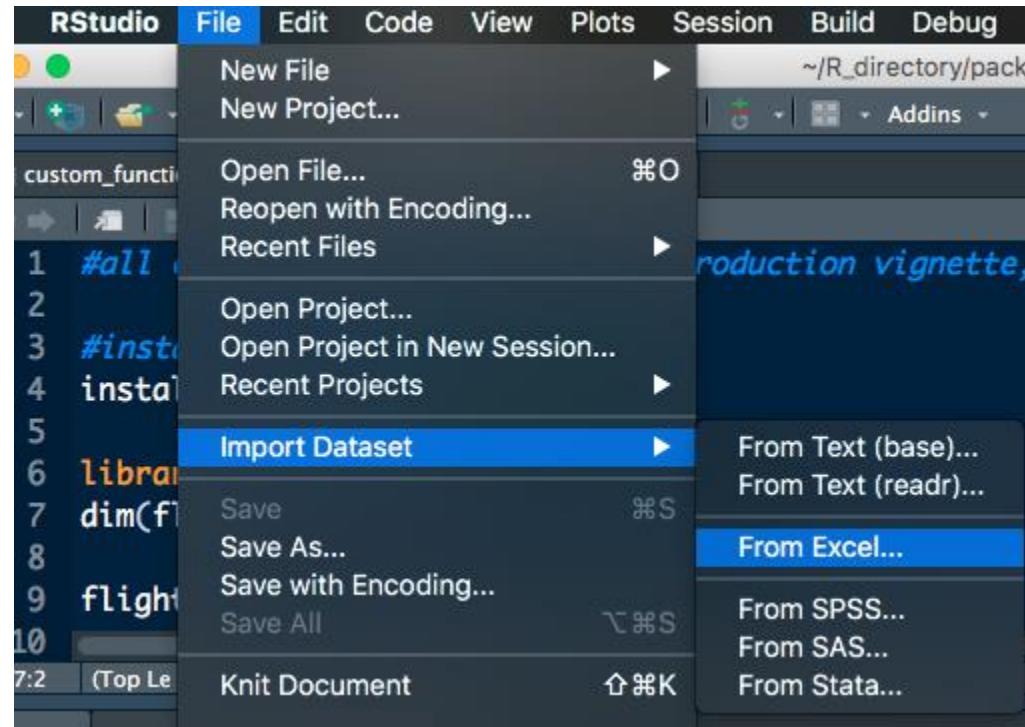
```
[1] 5
[1] 4
[1] 3
[1] 2
[1] 1
```





## Complete Activity: Building Basic Loops

# Data Import and Export



# Data Import and Export

RStudio also has point and click methods for importing data. If you navigate to File | Import Dataset, you'll see that you have six options:

- From text (base)
- From text (readr)
- From Excel
- From SPSS
- From SAS
- From Stata



## Import Text Data

File/Url:

~/R\_directory/d4dchi\_lobbyists/Lobbyist\_Data\_-\_Gifts.csv

Browse...

Data Preview:

GIFT_ID (integer)	PERIOD_START (character)	PERIOD_END (character)	GIFT (character)	RECIPIENT_FIRST_NAME (character)	RECIPIENT_LAST_NAME (character)	RI
1101	10/01/2012	12/31/2012	candy	Carl	Erickson	
288	01/01/2012	06/30/2012	lunch	Elizabeth	Horton-Newkirk	
88	01/01/2012	06/30/2012	Breakfast	Mary	O'Connor	
583	07/01/2012	09/30/2012	meal / snack	Deborah	Graham	
601	07/01/2012	09/30/2012	None	None	None	
933	10/01/2012	12/31/2012	Gift Basket	Tamra	Collins	
1047	10/01/2012	12/31/2012	Lunch	Michael	Zalewski	
1044	10/01/2012	12/31/2012	Lunch	Timothy	Cullerton	
939	10/01/2012	12/31/2012	Gift Basket	Lawrence	Grisham	

Previewing first 50 entries.

Import Options:

Name: <input type="text" value="Lobbyist_Data_Gifts"/>	<input checked="" type="checkbox"/> First Row as Names	Delimiter: <input type="button" value="Comma"/>	Escape: <input type="button" value="None"/>
Skip: <input type="text" value="0"/>	<input checked="" type="checkbox"/> Trim Spaces	Quotes: <input type="button" value="Default"/>	Comment: <input type="button" value="Default"/>
<input checked="" type="checkbox"/> Open Data Viewer		Locale: <input type="button" value="Configure..."/>	NA: <input type="button" value="Default"/>

Code Preview:

```
library(readr)
Lobbyist_Data_Gifts <- read_csv("~/R_directory/d4dchi_lobbyists/Lobbyist_Data_-_Gifts.csv")
View(Lobbyist_Data_Gifts)
```

(?) Reading rectangular data using readr

Import

Cancel

# Data Import and Export

- We can import data directly from GitHub by using the `read.table()` function.
- If we input the URL where the dataset is stored, the function will download it for you, as shown in the following example:

```
students_text <-  
read.table("https://github.com/fenago/students.txt")
```

# Data Import and Export

- While this code will read in the table, if we open and examine it, we will notice that the variable names are currently the first row of the dataset.
- To find the options for `read.table()`, we can use the following command:

?`read.table`



# Data Import and Export

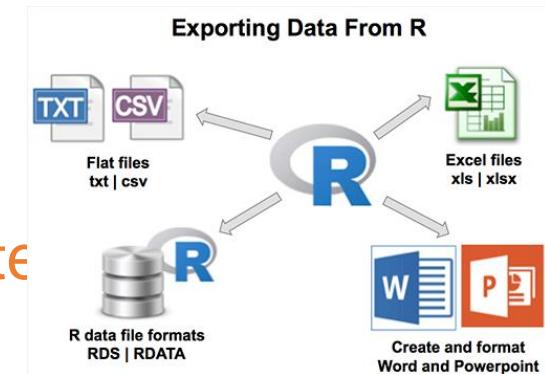
- Reading the documentation, it says that the default value for the header argument is FALSE.
- If we set it to TRUE, `read.table()` will know that our table contains—as the first row—the variable names, and they should be read in as names.
- Here is the example:

```
students_text <-  
read.table("https://github.com/fenago/students.txt",  
header = TRUE)
```

# Data Import and Export

- We may want to convert the Height\_inches variable to centimeters and the Weight\_lbs variable to kilograms.
- We can do so with the following code:

```
students_text$Height_cm <-  
(students_text$Height_inches * 2.54)  
students_text$Weight_kg <- (students_te  
/ 0.453592)
```



# Data Import and Export

- Since we've added these two variables, it may now be necessary to export the table out of R, perhaps to email it to a colleague for their use or to re-upload it on GitHub.
- The opposite of `read.table()` is `write.table()`. Take a look at the following example:

```
write.table(students_text, "students_text_out.txt")
```

# Data Import and Export

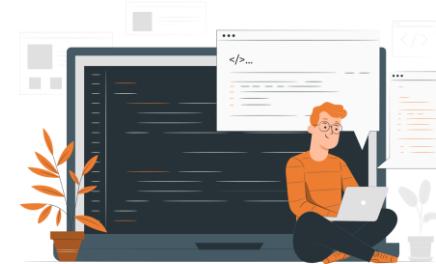
- Import the students.csv file from GitHub using `read.table()`.
- Save it as a dataset called `students_csv1`, using the following code:

```
students_csv1 <-  
read.table("https://github.com/fenago/students.cvc",  
header = TRUE, sep = ",")
```

# Data Import and Export

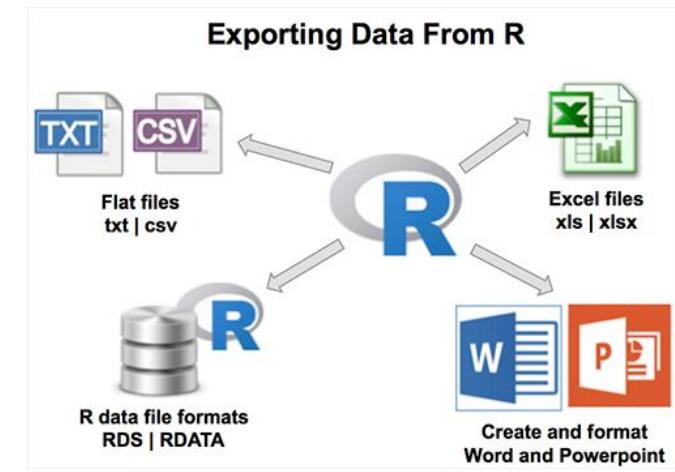
- Import students.csv using read.csv(), which works very similar to read.table():

```
students_csv2 <-  
read.csv("https://github.com/fenago/students.csv")
```



# Data Import and Export

- Download the `readr` package:  
`install.packages("readr")`
- Load the `readr` package:  
`library(readr)`



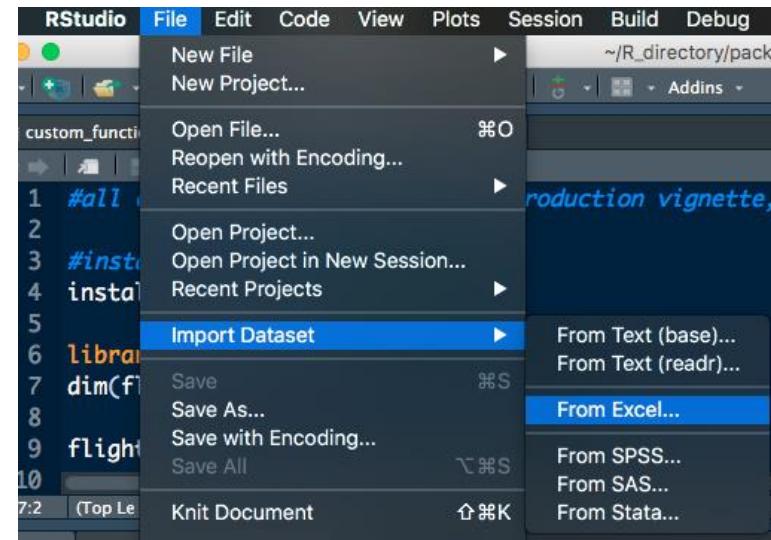
Import `students.csv` using `read_csv()`:

```
students_csv3 <-  
read_csv("https://github.com/fenago/students.csv")
```

# Data Import and Export

- Examine `students_csv2` and `students_csv3` with `str()`:

```
str(students_csv2)  
str(students_csv3)
```



# Data Import and Export

- The output for the str(students\_csv2) function is as follows:  
'data.frame': 20 obs. of 5 variables:  
  \$ Height\_inches: int 65 55 60 61 62 66 69 54 57 58 ...  
  \$ Weight\_lbs: int 120 135 166 154 189 200 250 122 101 178  
  ...  
  \$ EyeColor: Factor w/ 4 levels "Blue","Brown",...: 1 2 4 2 3 3 1  
    1 1 2 ...  
  \$ HairColor: Factor w/ 4 levels "Black","Blond",...: 3 2 1 3 2 4  
    4 3 3 1 ...  
  \$ USMensShoeSize: int 9 5 6 7 8 9 10 5 6 4 ...

# Data Import and Export

- The output for the str(students\_csv3) function is as follows:

Classes 'tbl\_df', 'tbl' and 'data.frame': 20 obs. of 5 variables:

```
$ Height_inches : int 65 55 60 61 62 66 69 54 57 58 ...
$ Weight_lbs : int 120 135 166 154 189 200 250 122
101 178 ...
$ EyeColor : chr "Blue" "Brown" "Hazel" "Brown" ...
$ HairColor : chr "Brown" "Blond" "Black" "Brown" ...
$ USMensShoeSize: int 9 5 6 7 8 9 10 5 6 4 ...
```

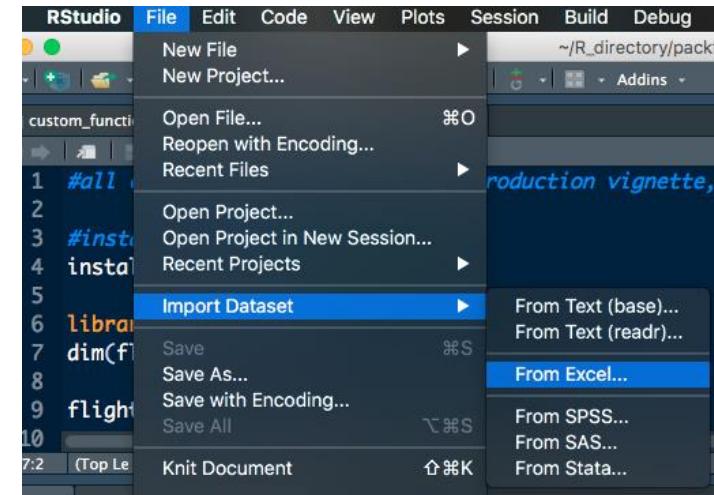
# Excel Spreadsheets

- Install and load the xlsx package by using the following code:

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

- Import students.csv using read.xlsx():

```
students_xlsx <-  
read.xlsx("students.xlsx")
```



# Excel Spreadsheets

- Create a new variable in `students_xlsx`, called `id`, with the following code:

```
students_xlsx$id <- seq(1:nrow(students_xlsx))
```

- Export `students_xlsx` to your working directory:  
`write.xlsx(students_xlsx, "students_xlsx_out.xlsx")`

File Type	Delimiter/Origin of dataset	function(s) to import data (package name)	function(s) to export data (package name)
<code>.csv</code> , CSV	comma	<code>read.csv</code> (base), <code>read_csv</code> (readr)	<code>write.csv</code> (base), <code>write_csv</code> (readr)
depends ( <code>.txt</code> , <code>.csv</code> )	tab	<code>read.table</code> , with sep = "\t"	<code>write.table</code> , with sep = "/t"
<code>.xlsx</code>	Excel sheet	<code>read.xlsx</code> (xlsx)	<code>write.xlsx</code> (xlsx)
<code>.sav</code>	SPSS	<code>spss.get</code> (Hmisc), <code>read_sav</code> (haven)	not advisable
<code>.sas7bdat</code>	SAS	<code>sasxport.get</code> (Hmisc), <code>read_sas</code> (haven)	not advisable
<code>.dta</code>	STATA	<code>read.dta</code> (foreign), <code>read_dta</code> (haven)	not advisable

Import Text Data

File/Url:  
~/R\_directory/d4dchi\_lobbyists/Lobbyist\_Data\_-\_Gifts.csv

Data Preview:

GIFT_ID (integer)	PERIOD_START (character)	PERIOD_END (character)	GIFT (character)	RECIPIENT_FIRST_NAME (character)	RECIPIENT_LAST_NAME (character)
1101	10/01/2012	12/31/2012	candy	Carl	Erickson
288	01/01/2012	06/30/2012	lunch	Elizabeth	Horton-Newkirk
88	01/01/2012	06/30/2012	Breakfast	Mary	O'Connor
583	07/01/2012	09/30/2012	meal / snack	Deborah	Graham
601	07/01/2012	09/30/2012	None	None	None
933	10/01/2012	12/31/2012	Gift Basket	Tamra	Collins
1047	10/01/2012	12/31/2012	lunch	Michael	Zalewski
1044	10/01/2012	12/31/2012	lunch	Timothy	Cullen
939	10/01/2012	12/31/2012	Gift Basket	Lawrence	Grisham

Previewing first 50 entries.

Import Options:

Name: <input type="text" value="Lobbyist_Data_Gifts"/>	<input checked="" type="checkbox"/> First Row as Names	Delimiter: <input type="button" value="Comma"/>	Escape: <input type="button" value="None"/>
Skip: <input type="text" value="0"/>	<input checked="" type="checkbox"/> Trim Spaces	Quotes: <input type="button" value="Default"/>	Comment: <input type="button" value="Default"/>
<input checked="" type="checkbox"/> Open Data Viewer		Locale: <input type="button" value="Configure..."/>	NA: <input type="button" value="Default"/>

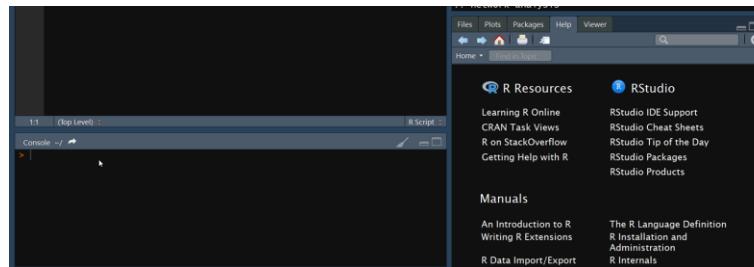
Code Preview:

```
library(readr)
Lobbyist_Data.Gifts <- read_csv("~/R_directory/d4dchi_lobbyists/Lobbyist_Data - Gifts.csv")
View(Lobbyist_Data.Gifts)
```

# Complete Activity: Exporting and Importing the mtcars Dataset

# Getting Help with R

- One advantage of using R is that it is a very well-documented programming language. This is often because there is a certain amount of documentation that is required by CRAN before it will publish a package on the website.
- It is considered a best practice to document your packages or functions well, no matter where you are publishing them.



# Package Documentation and Vignettes

- For example, say you (this one happens a lot) can't remember off the top of your head the inputs to the `glm()` function.
- The following code will bring up the documentation for the `glm()` function, as shown in the following screenshot:

```
Source on Save  
1 #help and ?  
2 help("glm")  
3 ?glm
```





## Search Results



### Code demonstrations:

[gbm::bernoulli](#) example of boosting for logistic regression, aka LogitBoost [\(Run demo in console\)](#)

[SQUAREM::mmlogistic](#) Examples of MM acceleration for Logistics Regression  
Maximum Likelihood Estimation.

### Help pages:

<a href="#">agricolae::reg.homog</a>	Homologation of regressions
<a href="#">base::&lt;</a>	Relational Operators
<a href="#">base::Control</a>	Control Flow
<a href="#">base::::</a>	Logical Operators
<a href="#">base::NA</a>	'Not Available' / Missing Values
<a href="#">base::all</a>	Are All Values True?
<a href="#">base::all.equal</a>	Test if Two Objects are (Nearly) Equal
<a href="#">base::any</a>	Are Some Values True?
<a href="#">base::as.data.frame</a>	Coerce to a Data Frame
<a href="#">base::bitwNot</a>	Bitwise Logical Operations
<a href="#">base::duplicated</a>	Determine Duplicate Elements
<a href="#">base::identical</a>	Test Objects for Exact Equality
<a href="#">base::ifelse</a>	Conditional Element Selection
<a href="#">base::logical</a>	Logical Vectors
<a href="#">base::match</a>	Value Matching
<a href="#">base::unique</a>	Extract Unique Elements
<a href="#">base::which</a>	Which indices are TRUE?
<a href="#">boot::inv.logit</a>	Inverse Logit Function
<a href="#">boot::logit</a>	Logit of Proportions
<a href="#">boot::neuro</a>	Neurophysiological Point Process Data
<a href="#">broom::multinom_tidiers</a>	Tidying methods for multinomial logistic regression models

# Tibbles

Tibbles are a modern take on data frames. They keep the features that have stood the test of time, and drop the features that used to be convenient but are now frustrating (i.e. converting character vectors to factors).

## Creating

`tibble()` is a nice way to create data frames. It encapsulates best practices for data frames:

- It never changes an input's type (i.e., no more `stringsAsFactors = FALSE!`).

```
tibble(x = letters)
#> # A tibble: 26 x 1
#>   x
#>   <chr>
#> 1 a
#> 2 b
#> 3 c
#> 4 d
#> # ... with 22 more rows
```

This makes it easier to use with list-columns:

```
tibble(x = 1:3, y = list(1:5, 1:10, 1:20))
#> # A tibble: 3 x 2
#>   x     y
#>   <int> <list>
#> 1     1 <int [5]>
#> 2     2 <int [10]>
#> 3     3 <int [20]>
```

List-columns are most commonly created by `do()`, but they can be useful to create by hand.

- It never adjusts the names of variables:

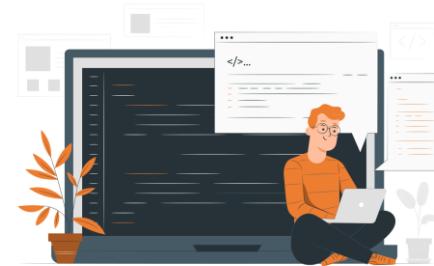
```
names(data.frame(`crazy name` = 1))
#> [1] "crazy.name"
names(tibble(`crazy name` = 1))
```

# Complete Activity: Exploring the Introduction to dplyr Vignette

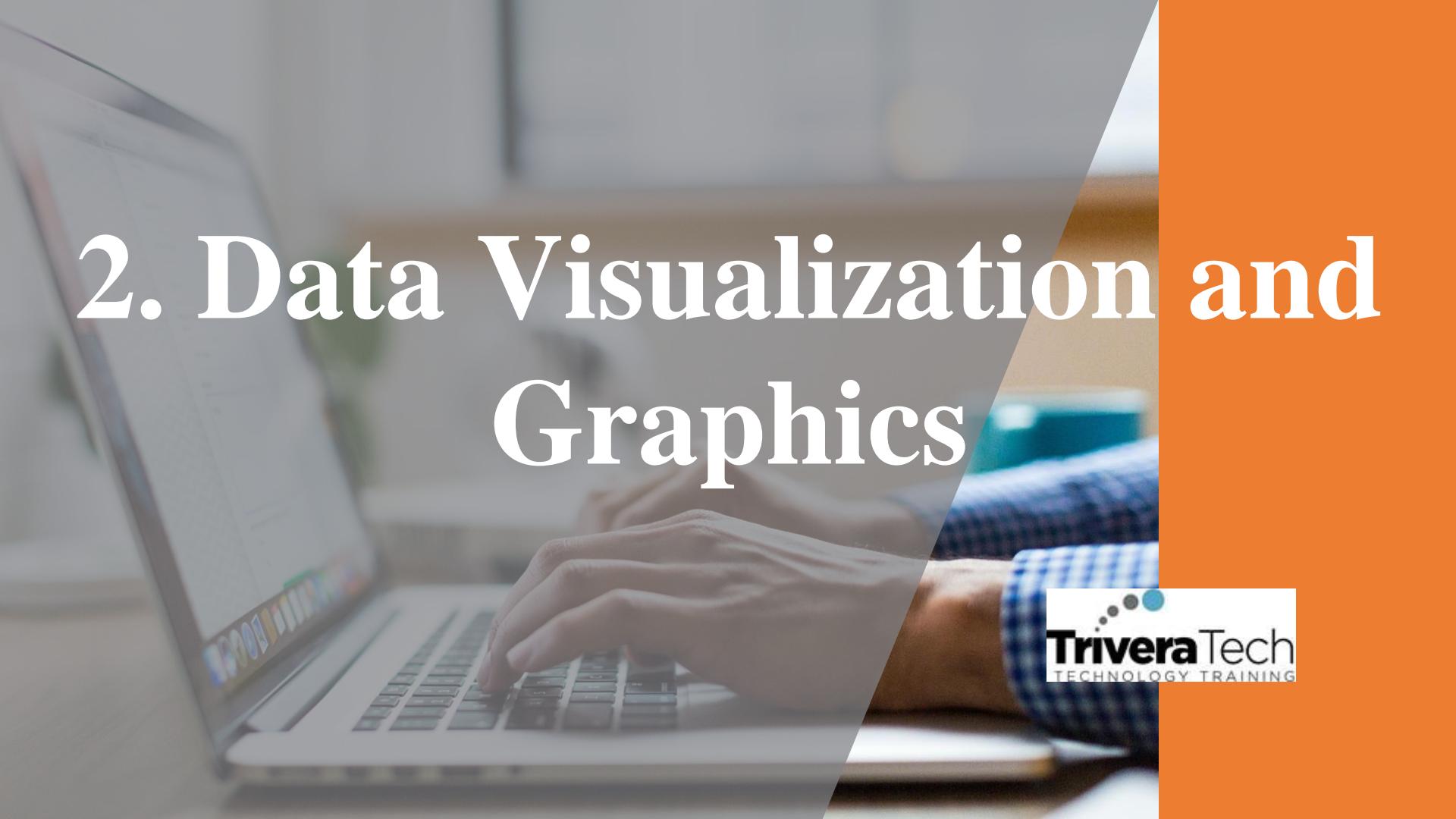


# Summary

- We've covered a lot in this introductory lesson and have plenty more to do, but don't fret!
- We'll continue to use plenty of examples and activities throughout to help you remember what we're learning.



# 2. Data Visualization and Graphics

A blurred background image of a person's hands typing on a laptop keyboard, suggesting a technical or data-related environment.

# Data Visualization and Graphics

By the end of this lesson, you will be able to:

- Use Base R for plotting, and identify when to do so
- Create a variety of different data visualizations using the ggplot2 package
- Explain different tools for interactive plotting in R

# Creating Base Plots

## The `plot()` Function

`main`

a overall title for the plot: see [title](#).

`sub`

a sub title for the plot: see [title](#).

`xlab`

a title for the x axis: see [title](#).

`ylab`

a title for the y axis: see [title](#).

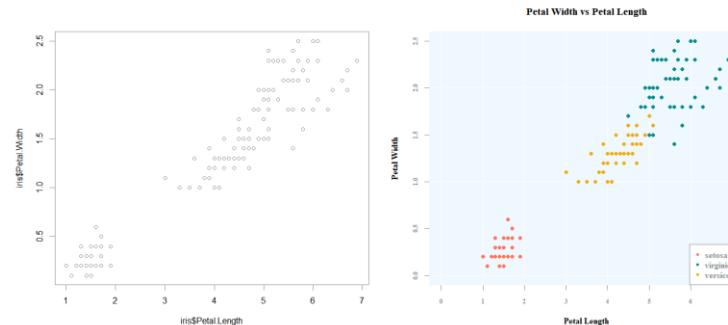
`asp`

the y/x aspect ratio, see [plot.window](#).

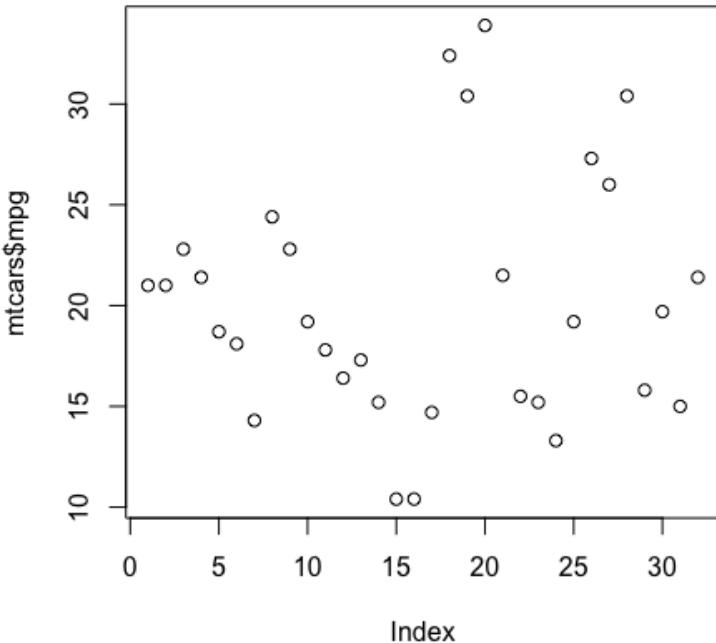
# Creating Base Plots

When you start out to write plots in base R, you may be interested to know that there are many other inputs besides just the data you want to plot. You can access the R help documentation for the `plot()` function in the following ways:

- `?plot`
- `help("plot")`
- `help(plot)`



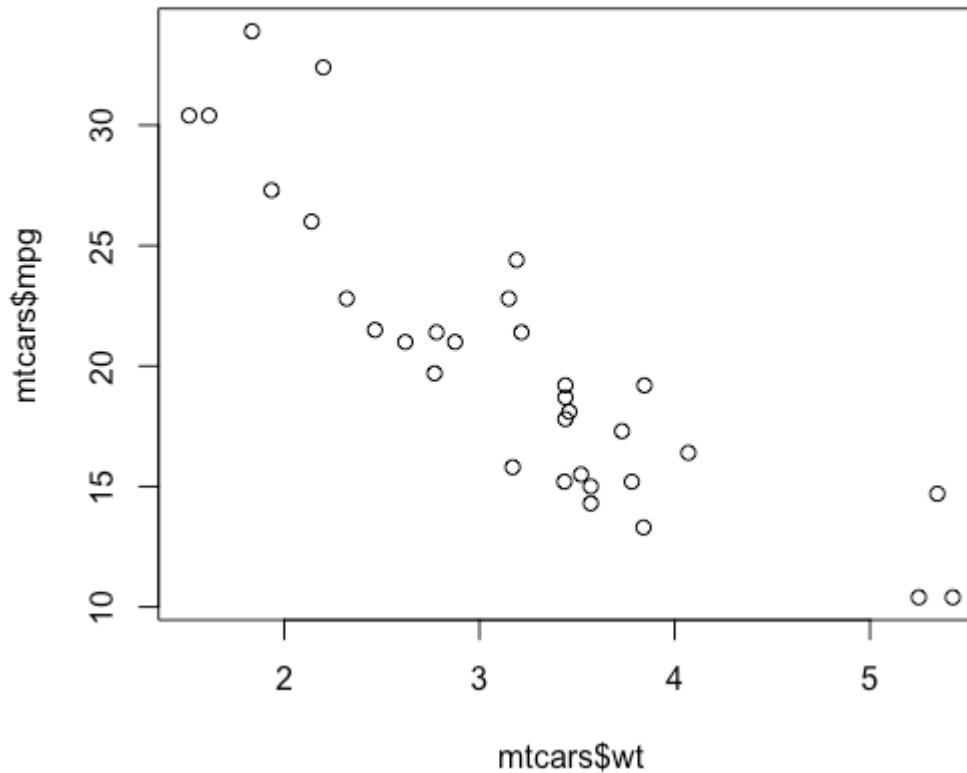
# Creating Base Plots



- This plot isn't very informative, but it is powerful in terms of seeing how well R can plot even when it is not installed on a particular machine.
- Let's add in a second variable and plot mpg versus wt:

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

### wt vs. mpg, mtcars

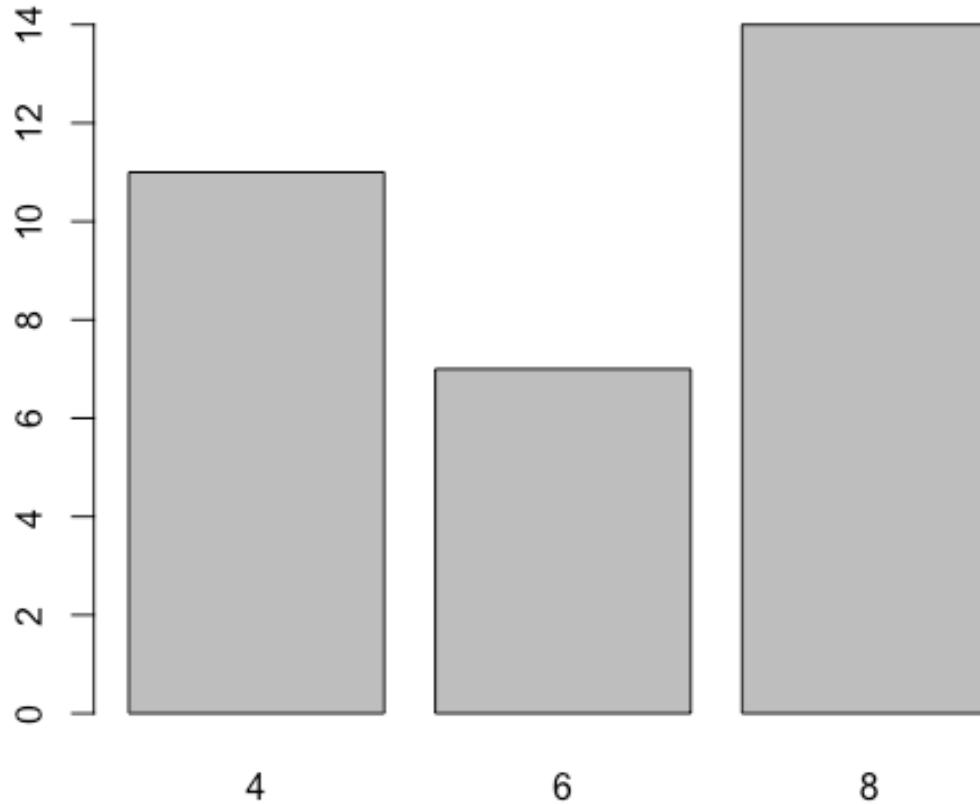


# Factor Variables

- For example, the cyl variable in mtcars gives the number of cylinders each car has.
- If we input it as a factor variable into plot, we get a bar chart (histogram) by default, where each bar gives a count of how many cars have each number of cylinders:

```
plot(as.factor(mtcars$cyl))
```

### histogram of cyl from mtcars



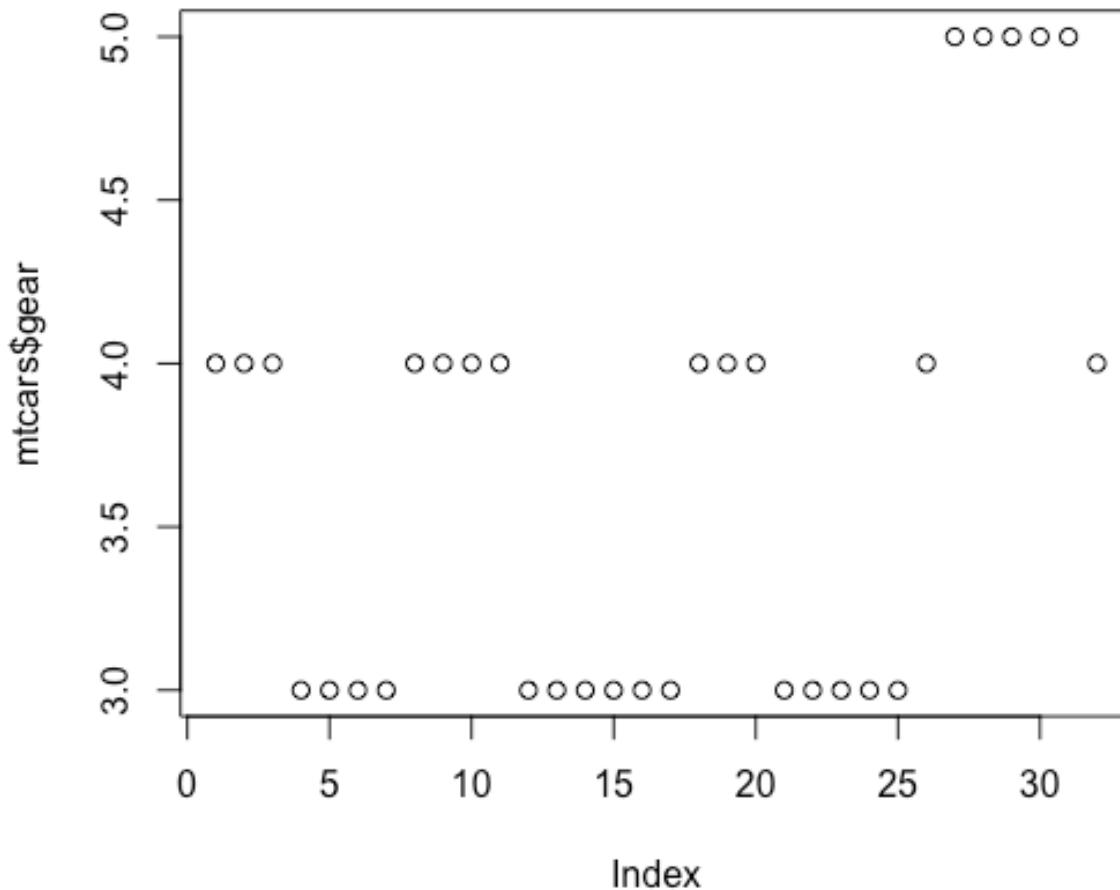
# Factor Variables

- Now, plot the gear variable of mtcars as a factor variable, as follows:

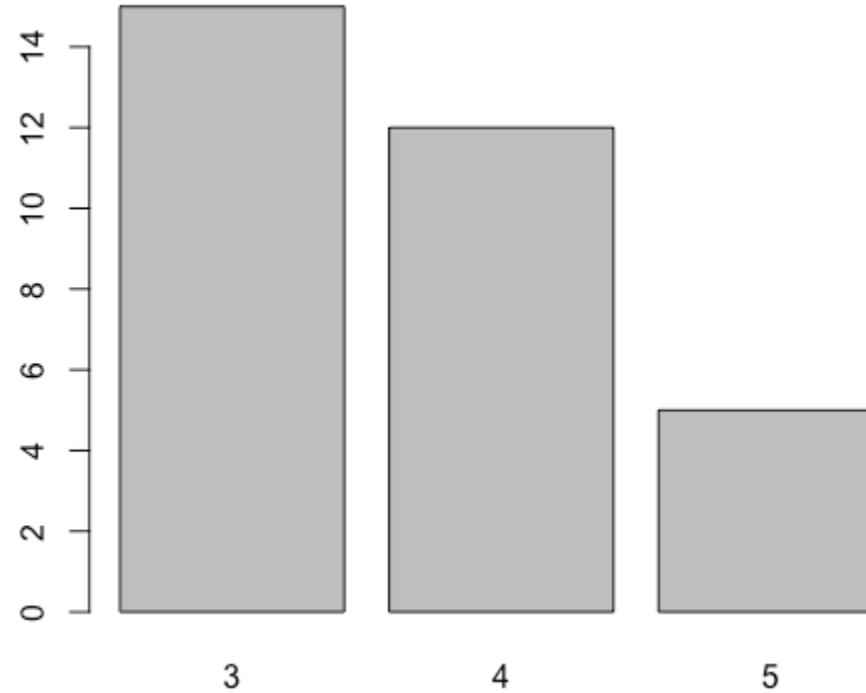
```
plot(as.factor(mtcars$gear))
```

What kind of plot is generated?





# Factor Variables



# Model Objects

- If you were to input a linear model object, `plot()` automatically returns four helpful model diagnostic plots, including the Residuals versus Fitted and Normal Q-Q plots, which help you determine whether your model fits well.
- The following code demonstrates this:

```
mtcars_lm <- lm(mpg ~ wt, data = mtcars)  
plot(mtcars_lm)
```

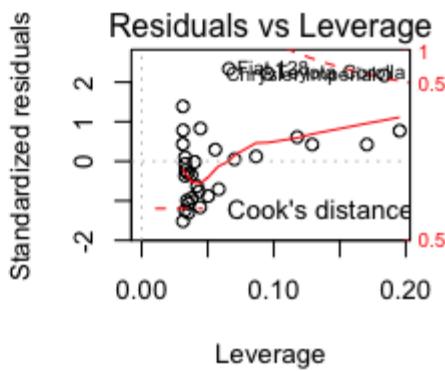
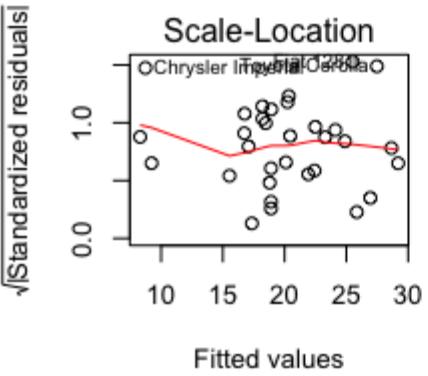
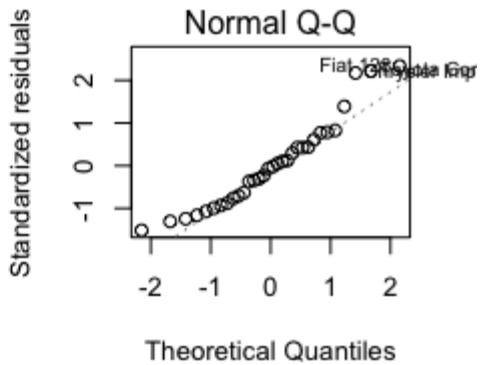
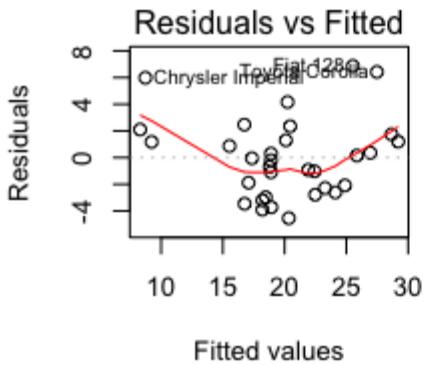
# Plotting More Than One Plot at a Time

- If we return to the mtcars\_lm() example we just covered, we can plot all four model diagnostic plots in the same window by first running the following line of code:

```
par(mfrow = c(2,2))
```

- Next, you need to execute the following code:

```
plot(mtcars_lm)
```



# Creating and Plotting a Linear Model Object

- Build your own version of mtcars\_lm, which looks at how the displacement and weight variables affect mpg using the following code:

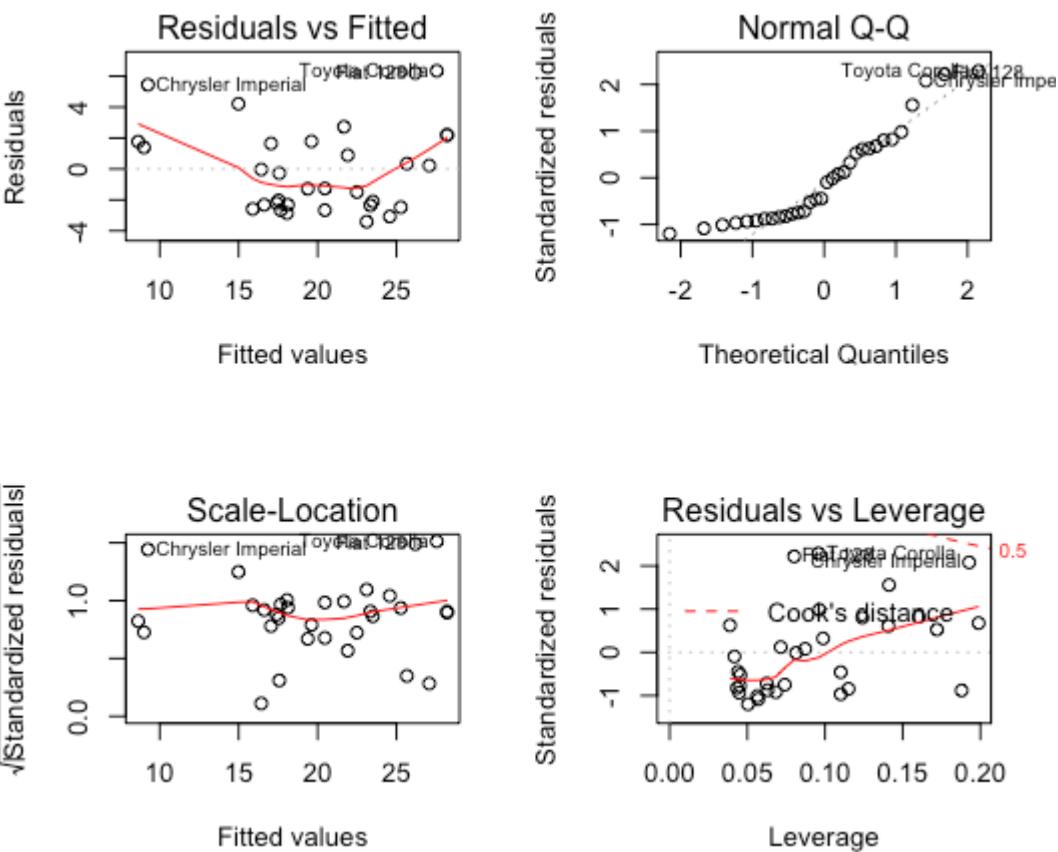
```
mtcars_lm <- lm(mpg ~ disp + wt, data = mtcars)
```

- Run the following code to enable plotting a  $2 \times 2$  grid of plots so that looking at model diagnostic plots is easier with the following method:

```
par(mfrow = c(2, 2))
```

- Turn the  $2 \times 2$  grid off using `dev.off()`.

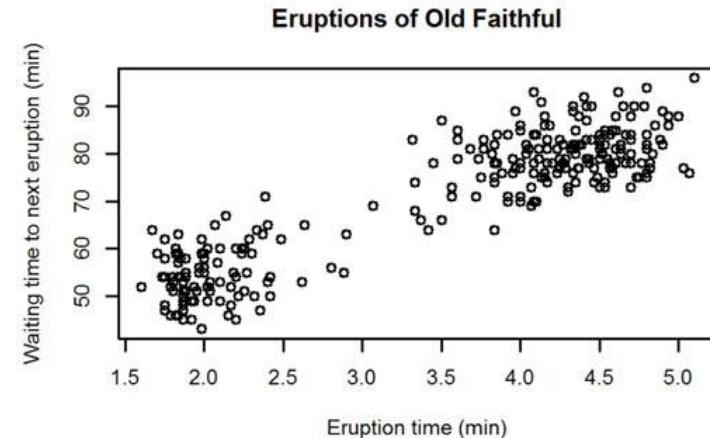
Output: The following is the output we get when we execute the `plot()` function as mentioned in Step 3.



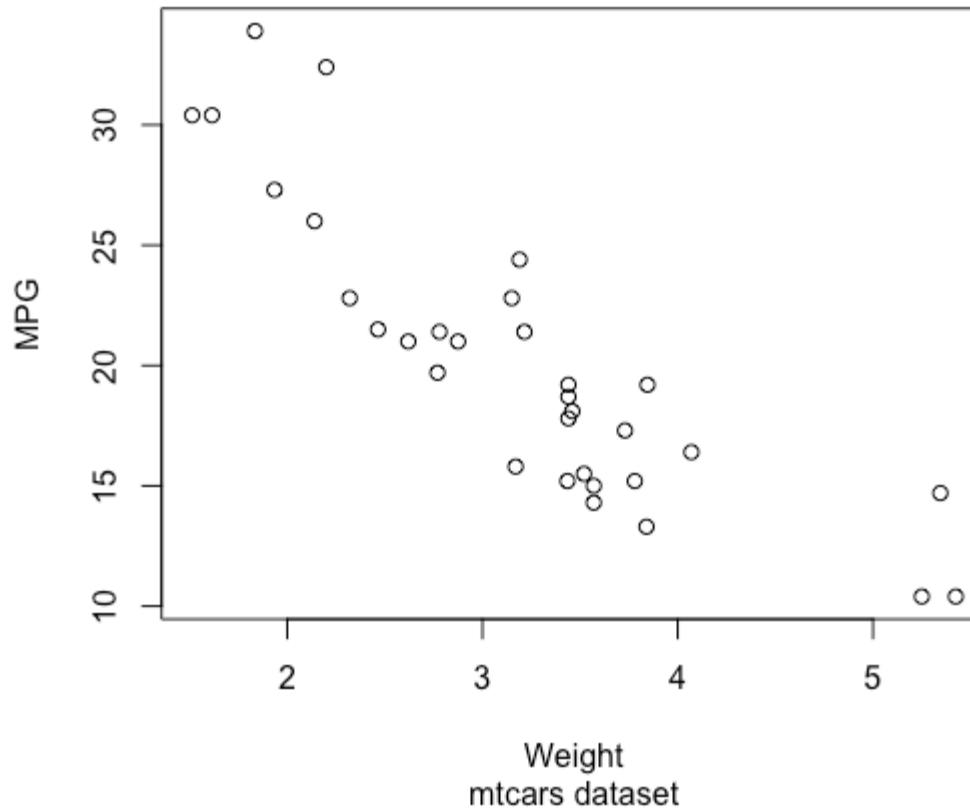
# Titles and Axis Labels

- Let's return to our mtcars scatterplot, add a title and subtitle, and also change the axis labels with the following code:

```
plot(mtcars$wt, mtcars$mpg,  
     main = "MPG vs. Weight",  
     sub = "mtcars dataset",  
     xlab = "Weight",  
     ylab = "MPG")
```



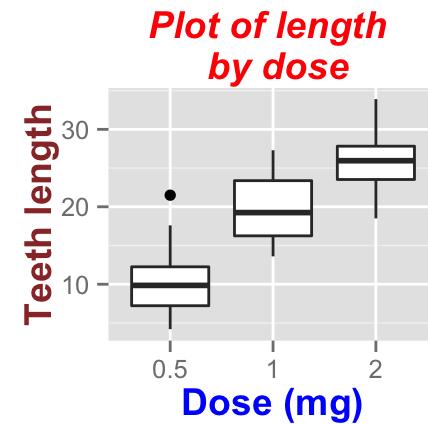
## MPG vs. Weight



# Titles and Axis Labels

- Plot petal length and width from the iris dataset to see what the plot looks like, and take note of the default axis labels as follows:

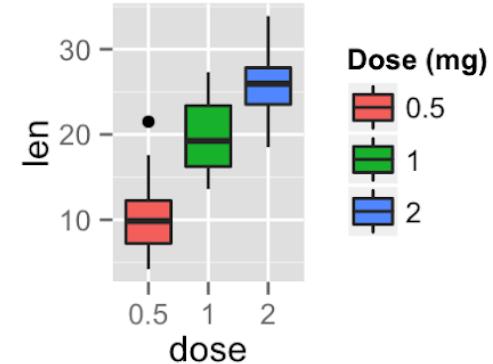
```
plot(iris$Petal.Length, iris$Petal.Width)
```

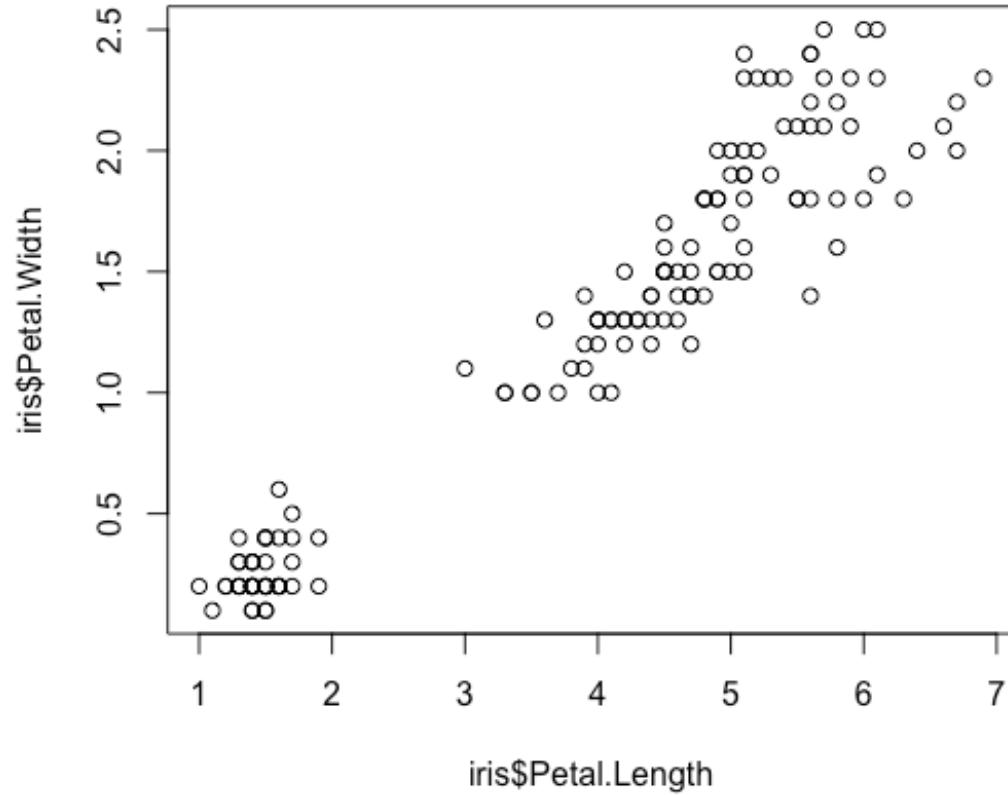


# Titles and Axis Labels

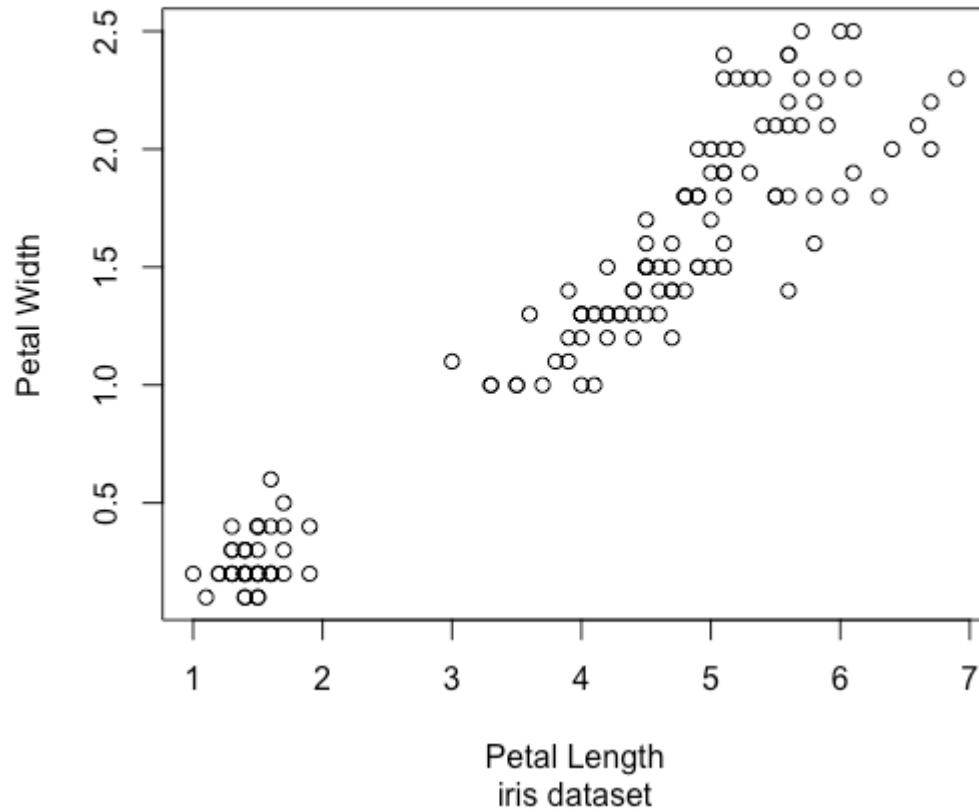
- Now, add a title, subtitle, and custom axis labels to the same plot using the following code:

```
plot(iris$Petal.Length, iris$Petal.Width,  
     main = "Petal Width vs. Length",  
     sub = "iris dataset",  
     xlab = "Petal Length",  
     ylab = "Petal Width")
```





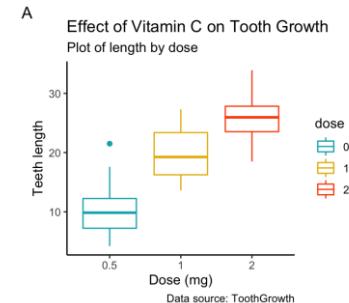
## Petal Width vs. Length



# Titles and Axis Labels

- R supports the names of many different colors along with hexadecimal color codes.
- The code to change the previous plot to red would be as follows:

```
plot(mtcars$wt, mtcars$mpg,  
     main = "mpg vs. wt, mtcars data",  
     xlab = "weight",  
     ylab = "mpg",  
     col = "red")
```



# Changing the Color of Base Plots

- Use the col option to turn the plot from the last exercise blue as follows:

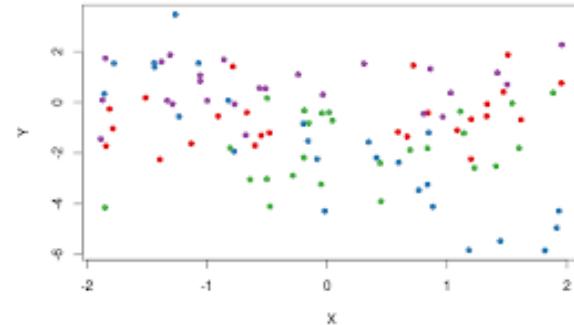
```
plot(iris$Petal.Length, iris$Petal.Width,  
     main = "Petal Width vs. Length",  
     sub = "iris dataset",  
     xlab = "Petal Length",  
     ylab = "Petal Width",  
     col = "blue")
```



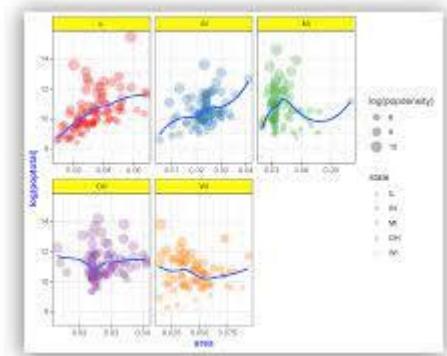
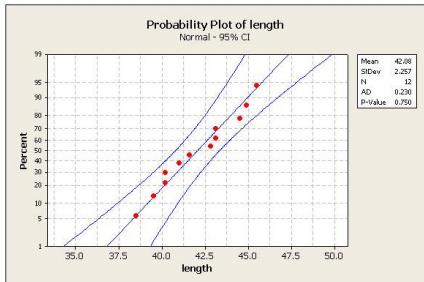
# Changing the Color of Base Plots

- Use the col option to turn the plot from the last exercise yellow using the hexadecimal color code 111111:

```
plot(iris$Petal.Length, iris$Petal.Width,  
     main = "Petal Width vs. Length",  
     sub = "iris dataset",  
     xlab = "Petal Length",  
     ylab = "Petal Width",  
     col = "111111")
```

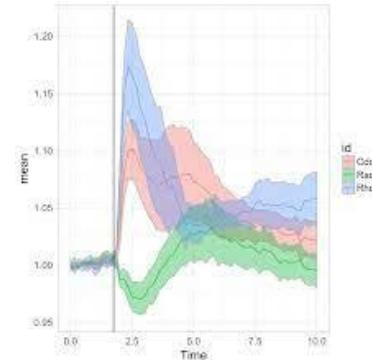


# Complete Activity: Recreating Plots with Base Plot Methods



# ggplot2

- ggplot2 is an incredibly popular graphics package in R.
- It can be installed on its own or comes as part of the Tidyverse set of packages.

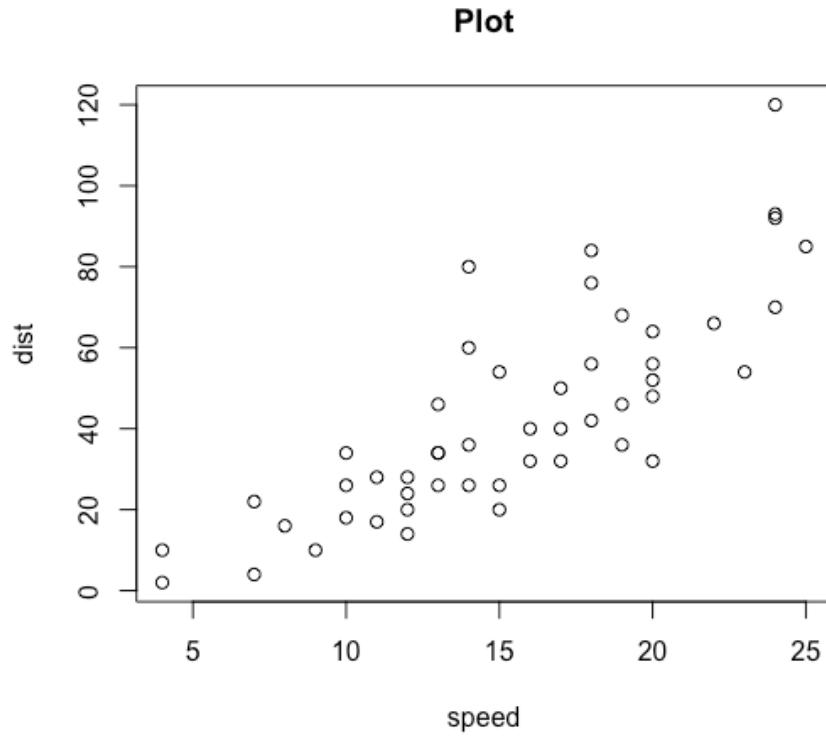


# ggplot2

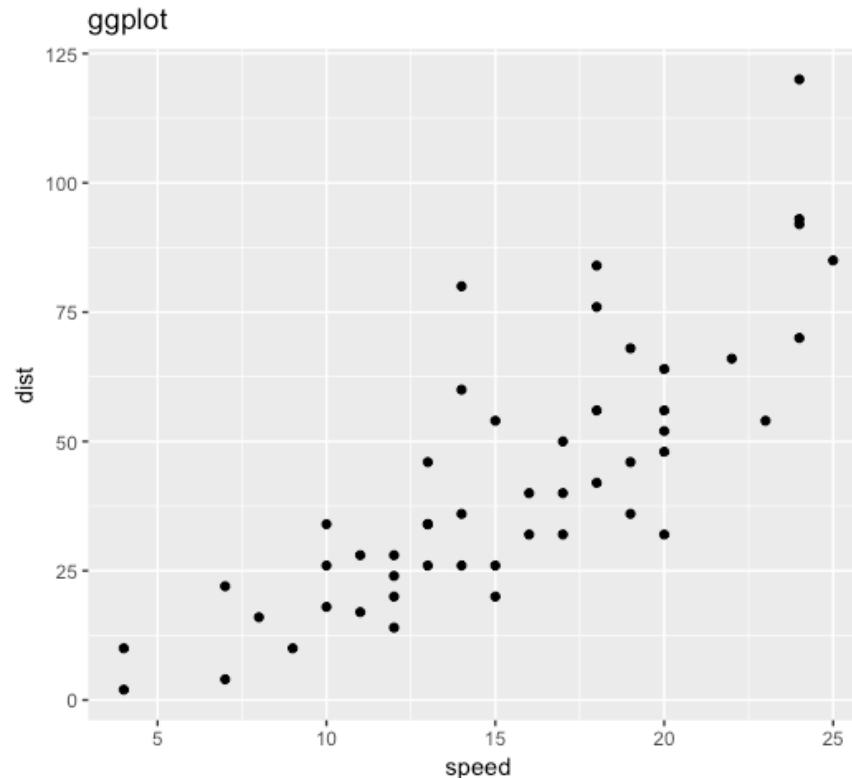
- Recall that we learned about Stack Overflow in lesson 1, Introduction to R, as a valuable resource for seeking assistance with R:

```
> library(ggplot2)
Stackoverflow is a great place to get help:
http://stackoverflow.com/tags/ggplot2.
```

# ggplot2 Basics



# ggplot2 Basics

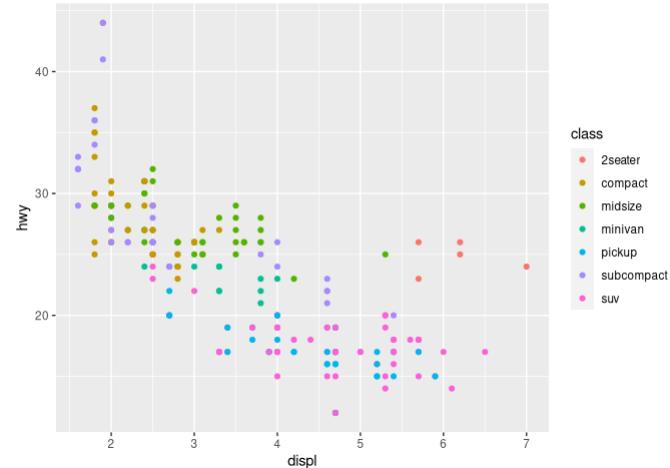


# ggplot2 Basics

- Plot 1:  
`plot(cars)`

- Plot 2:

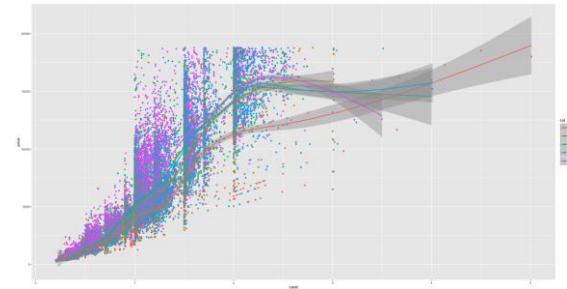
```
library(ggplot2)  
ggplot(cars, aes(speed, dist)) + geom_point()
```



# ggplot2 Basics

- You can layer on additional aesthetics, such as plot titles, axis labels, colors, different point types, and more.

```
ggplot(data = <DATA>) +  
<GEOM_FUNCTION>(mapping =  
aes(<MAPPINGS>))
```



# ggplot2 Basics

- Mappings are the variables you want to graph plus other aesthetics (aes() is short for aesthetics)

```
ggplot(data = <DATA>, aes(<GLOBAL  
MAPPINGS>)) + <GEOM_FUNCTION>(mapping  
= aes(<LOCAL MAPPINGS>))
```

# ggplot2 Basics

- There are a few things you should know about creating ggplots that will help you along the way.
- Firstly, you can save a ggplot call and use it for multiple graphs, for example:

```
#save the ggplot data and mappings as 'mtcars_ggplot':  
mtcars_ggplot <- ggplot(mtcars, aes(wt, mpg))  
#create 2 additional plots:  
mtcars_ggplot + geom_point()  
mtcars_ggplot + geom_point(aes(col = factor(cyl)))
```

Name	Type	Value
mtcars_ggplot	list [9] (S3: gg, ggplot)	List of length 9
data	list [32 x 11] (S3: data.frame)	A data.frame with 32 rows and 11 columns
layers	list [0]	List of length 0
scales	environment [1] (S3: ScalesList)	<environment: 0x108eda358>
mapping	list [2] (S3: uneval)	List of length 2
x	symbol	'wt'
y	symbol	'mpg'
theme	list [0]	List of length 0
coordinates	environment [3] (S3: CoordCa)	<environment: 0x108ed8ae8>
facet	environment [3] (S3: FacetNull)	<environment: 0x108ed76d8>
plot_env	environment [4]	<environment: R_GlobalEnv>
labels	list [2]	List of length 2
x	character [1]	'wt'
y	character [1]	'mpg'

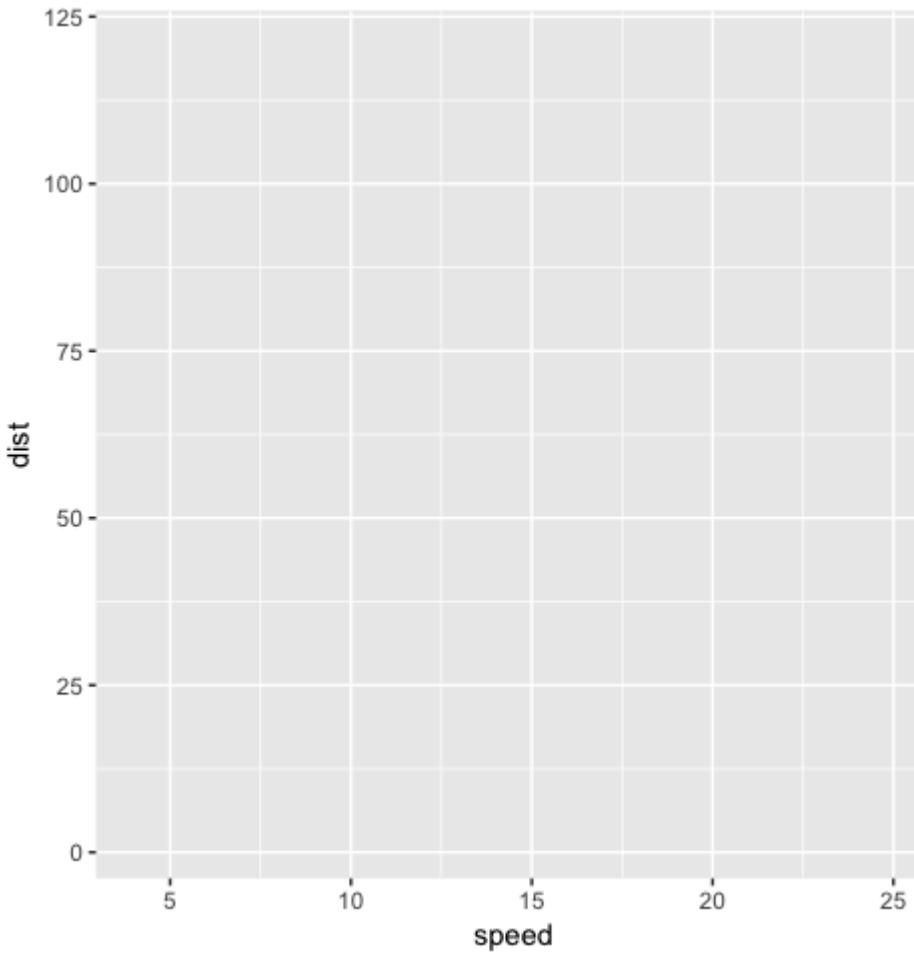
# ggplot2 Basics

- Secondly, the plus signs you'll need to add layers to a `ggplot()` object must always come at the end of a line.
- The following code will run successfully to create the plot we saw at the beginning of this subtopic:

```
ggplot(cars, aes(speed, dist)) + geom_point()
```

- The following code will not run, because the plus sign has been moved down to the second line, in front of `geom_point()`:

```
ggplot(cars, aes(speed, dist)) + geom_point()
```



# ggplot2 Basics

- If you attempt to run code with the plus sign at the beginning of a line, preceding geom\_point(), as in the previous example, a blank plot, as shown in the preceding screenshot, will generate in your Plots window in RStudio and you will get the following error in your console:

Error in +geom\_point() : invalid argument to unary operator

# Histogram

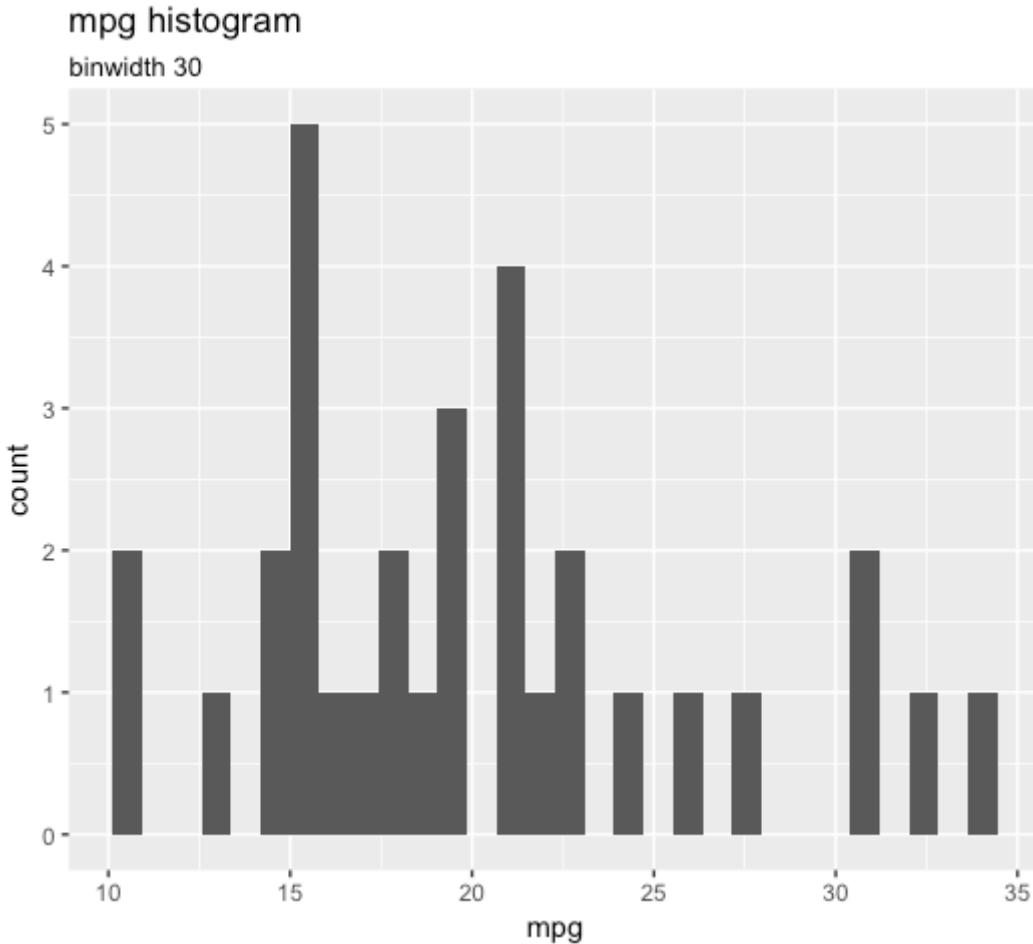
- When you have one continuous variable, it's a good idea to use a histogram to get an idea of its distribution.
- The height of the bar of the histogram corresponds to the number of observations that have that value.
- We can create a histogram of the mpg variable in mtcars using the following code:

```
ggplot(mtcars, aes(mpg)) + geom_histogram()
```

# Histogram

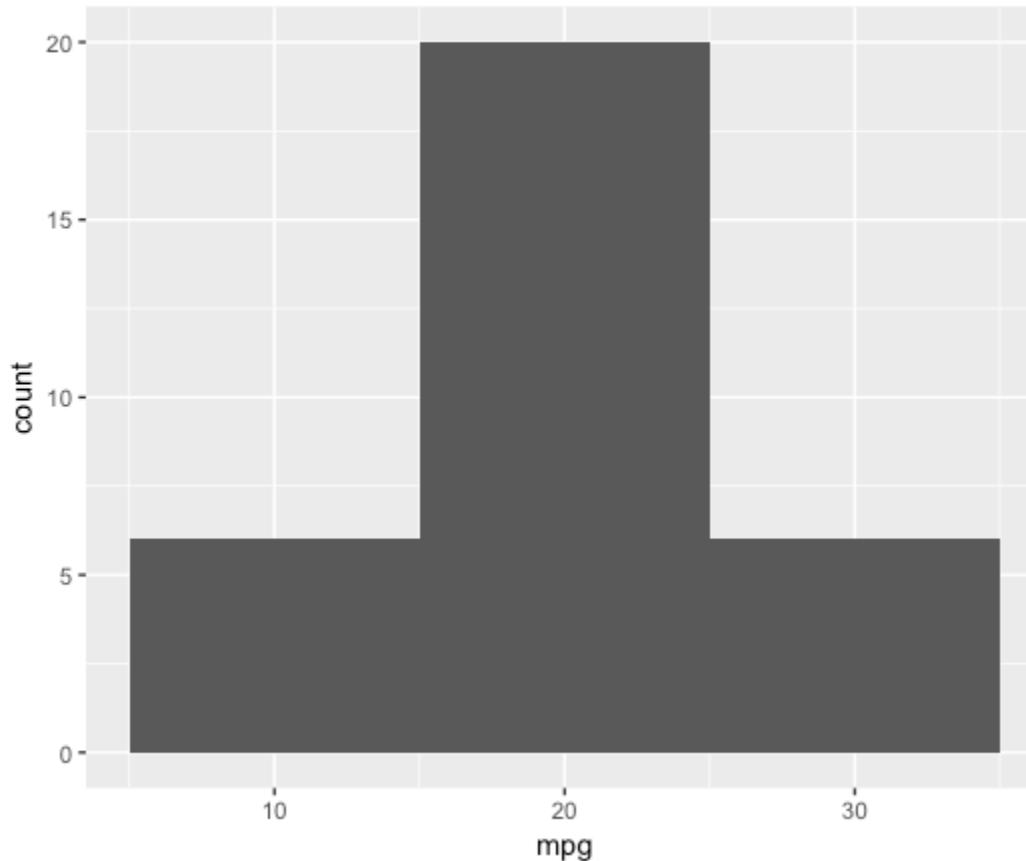
- Previous code will throw a warning:

'stat\_bin()' using 'bins = 30'. Pick better value with 'binwidth'.



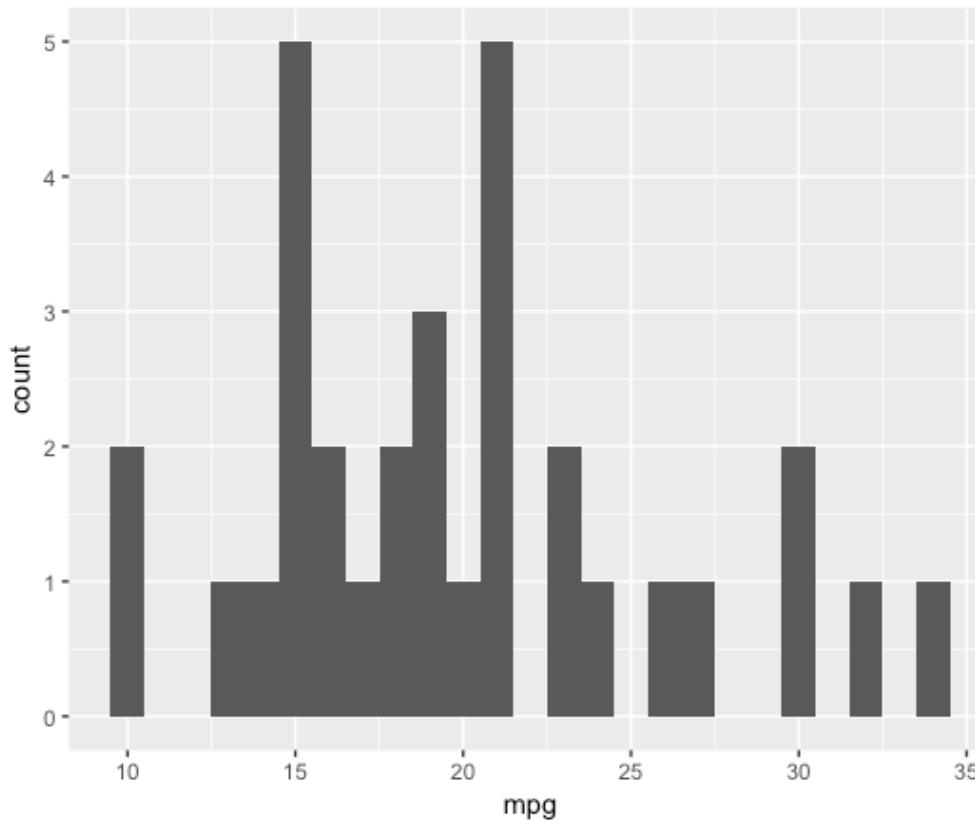
## mpg histogram

binwidth 10



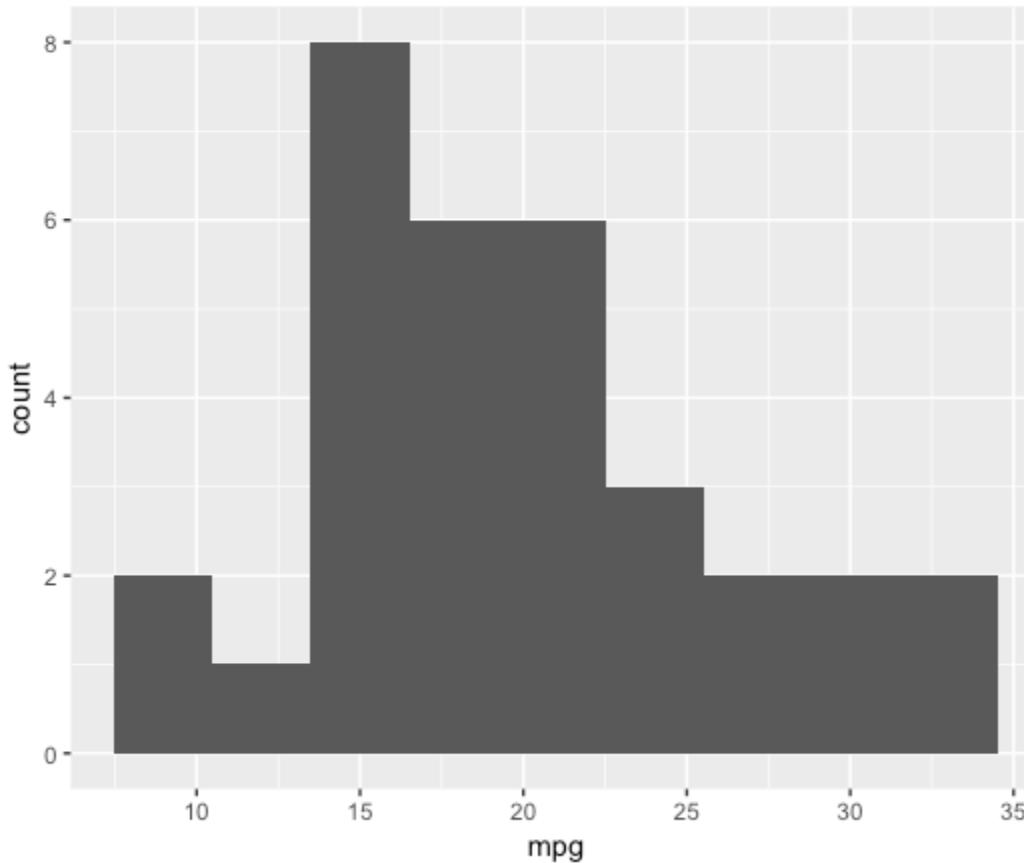
## mpg histogram

binwidth 1



## mpg histogram

binwidth 3



# Creating Histograms using ggplot2

- Install the ggplot2 library and then load it:

```
install.packages("ggplot2")  
library(ggplot2)
```

- Load the msleep dataset, a built-in dataset that comes installed with ggplot2, using data("msleep")

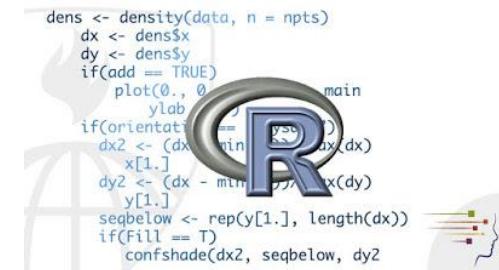
# Creating Histograms using ggplot2

- Create a histogram of the sleep\_total variable from msleep. Do you get the binwidth error?

```
ggplot(msleep, aes(sleep_total)) + geom_histogram()
```

- Try the same histogram, but with binwidth = 10. Does the histogram improve?

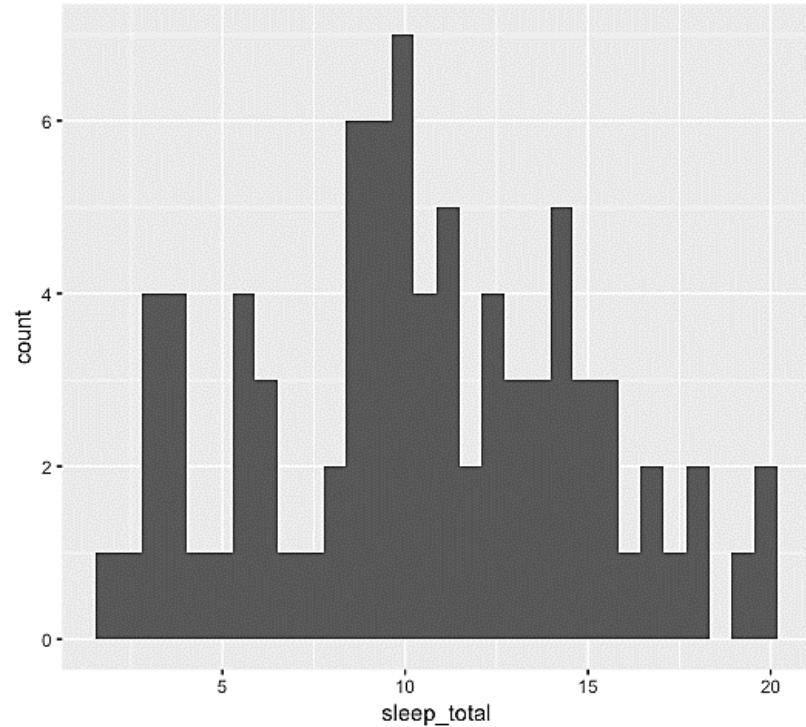
```
ggplot(msleep, aes(sleep_total)) +  
  geom_histogram(binwidth = 10)
```

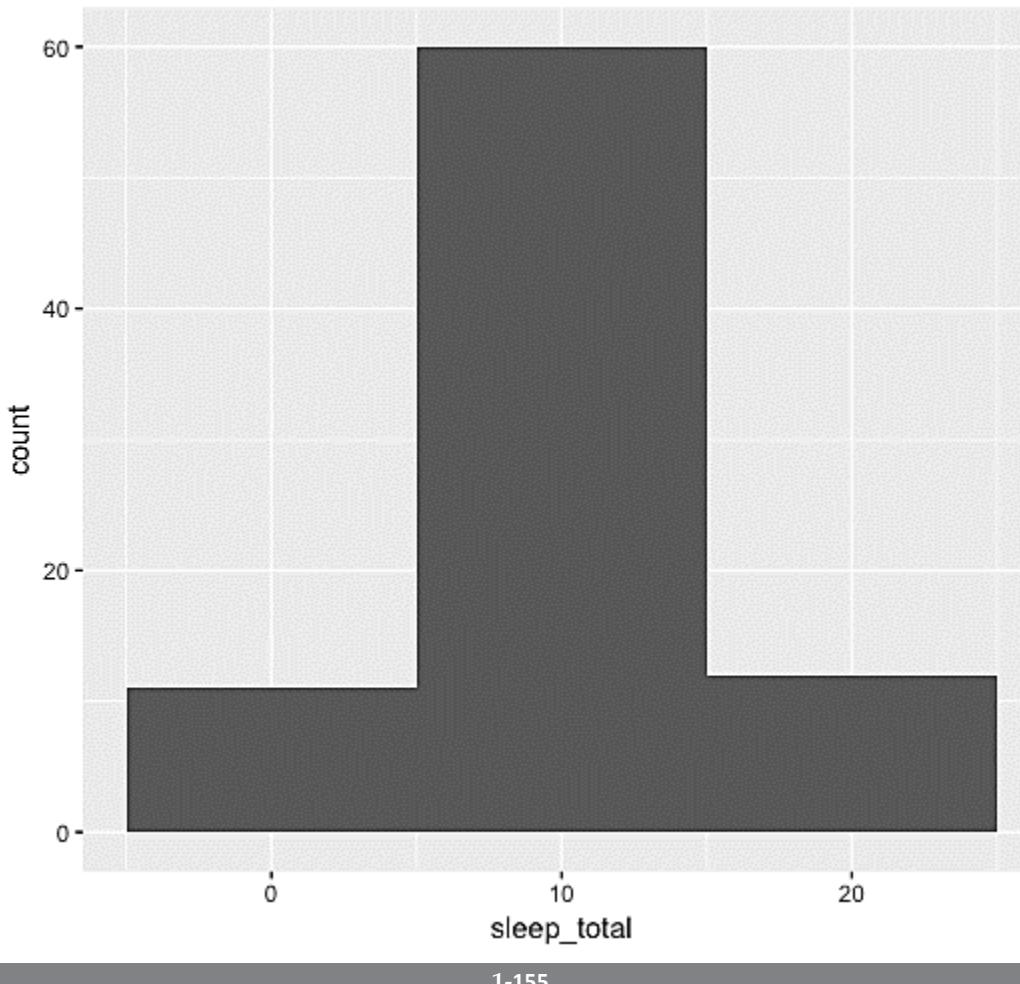


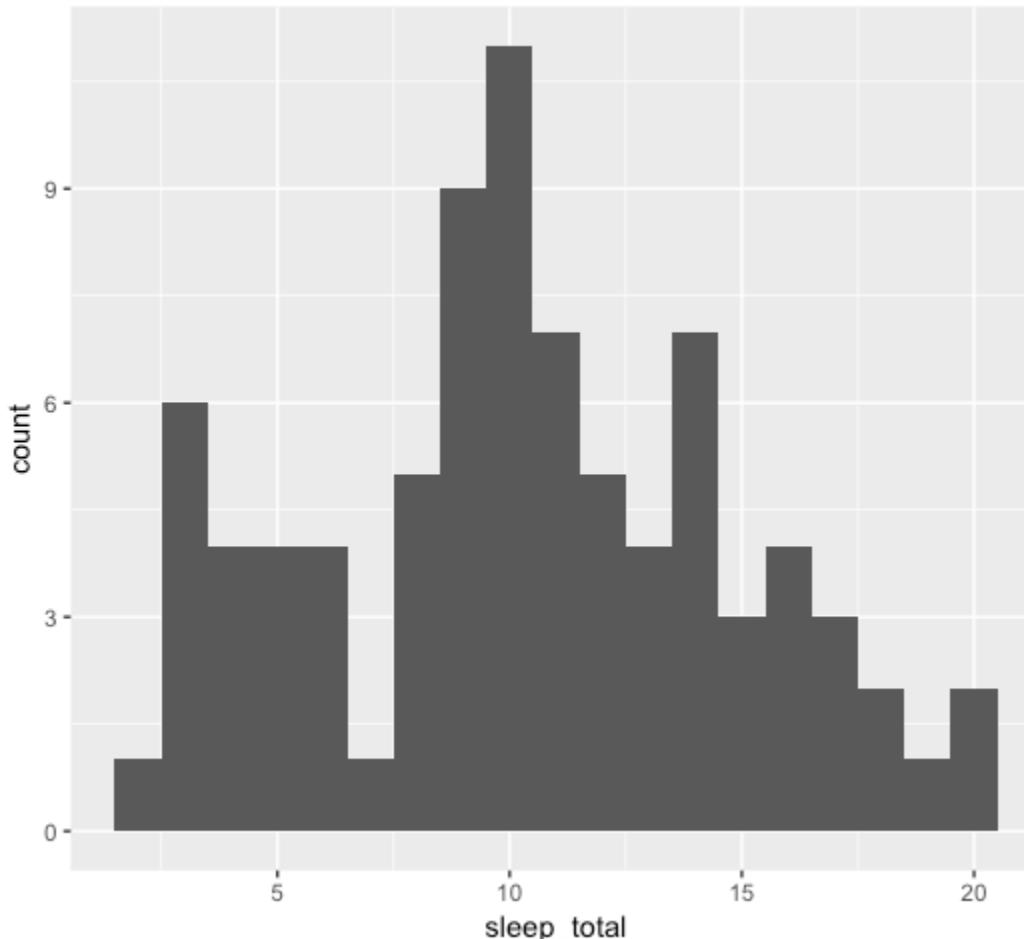
# Creating Histograms using ggplot2

- Try the histogram one more time, now with binwidth = 1:

```
ggplot(msleep,  
aes(sleep_total)) +  
geom_histogram(binwidth = 1)
```



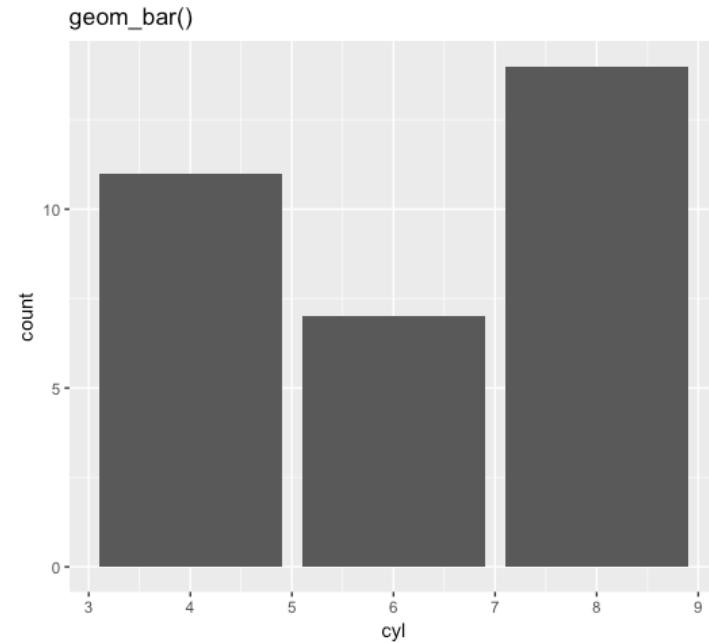




# Bar Chart

- For one categorical or factor variable, you can create a bar chart.
- We can create a bar chart of the cyl variable of mtcars using the following code:

```
#using geom_bar()  
ggplot(mtcars, aes(cyl)) + geom_bar()
```

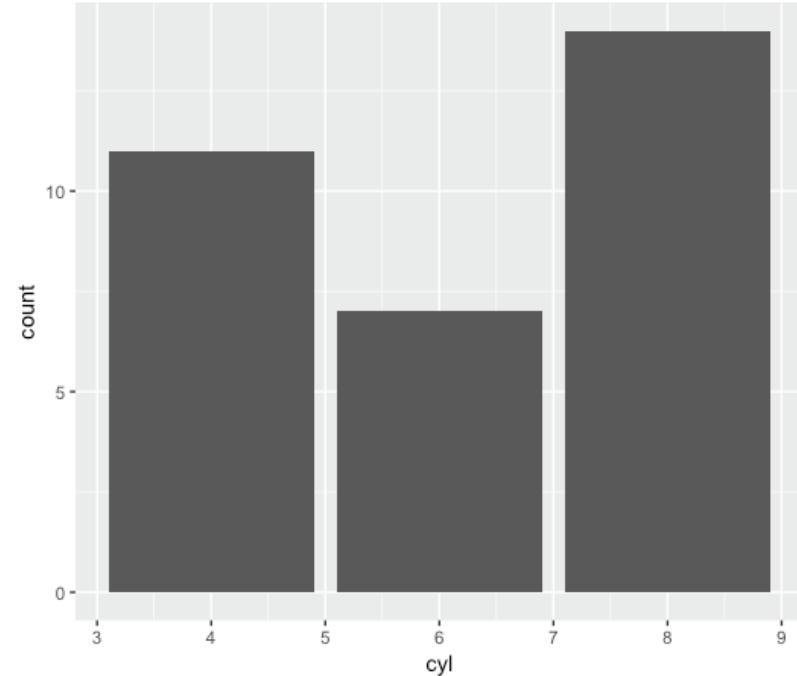


# Bar Chart

- One fun fact is that we can actually create bar charts with the `geom_histogram()` call as well, by including `stat = "count"`, as follows:

```
#using geom_histogram() and stat
ggplot(mtcars, aes(cyl)) +
geom_histogram(stat = "count")
```

`geom_hist(stat = "count")`



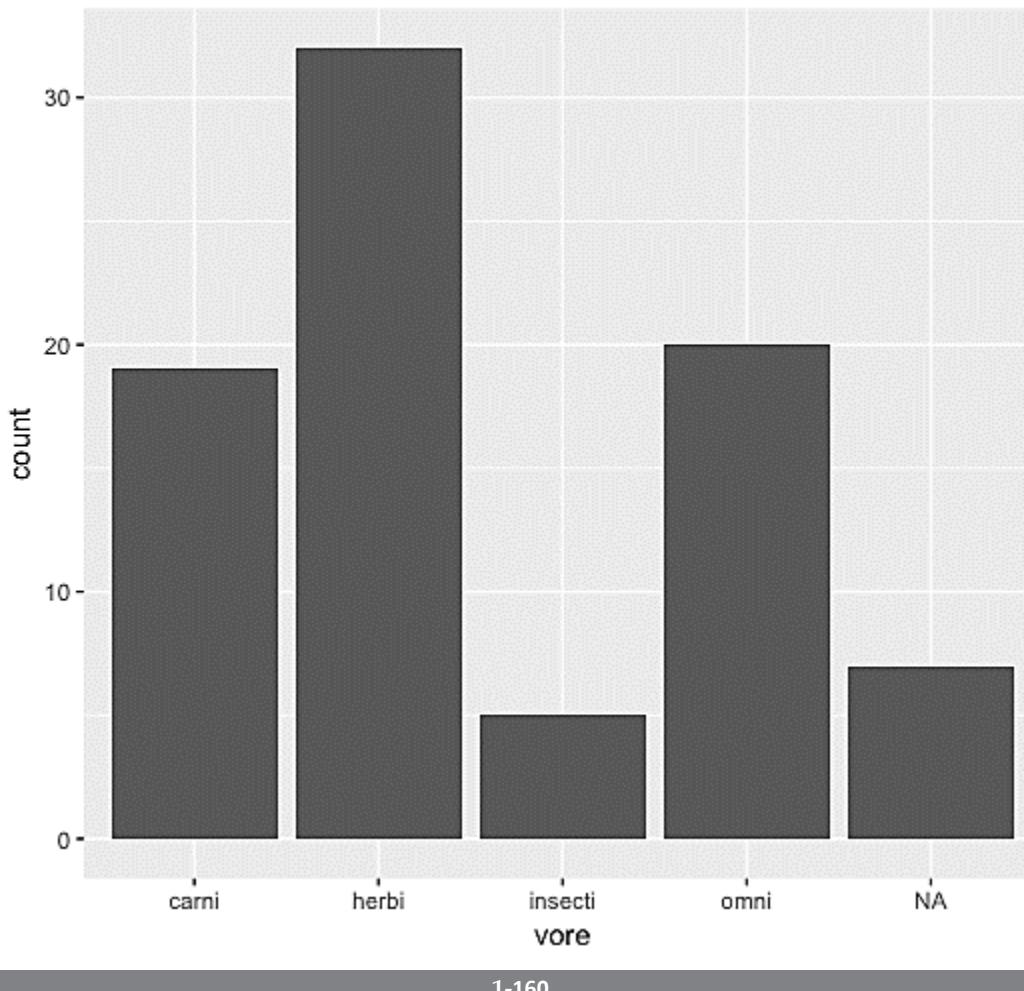
# Creating a Bar Chart with ggplot2 using 2 Different Methods

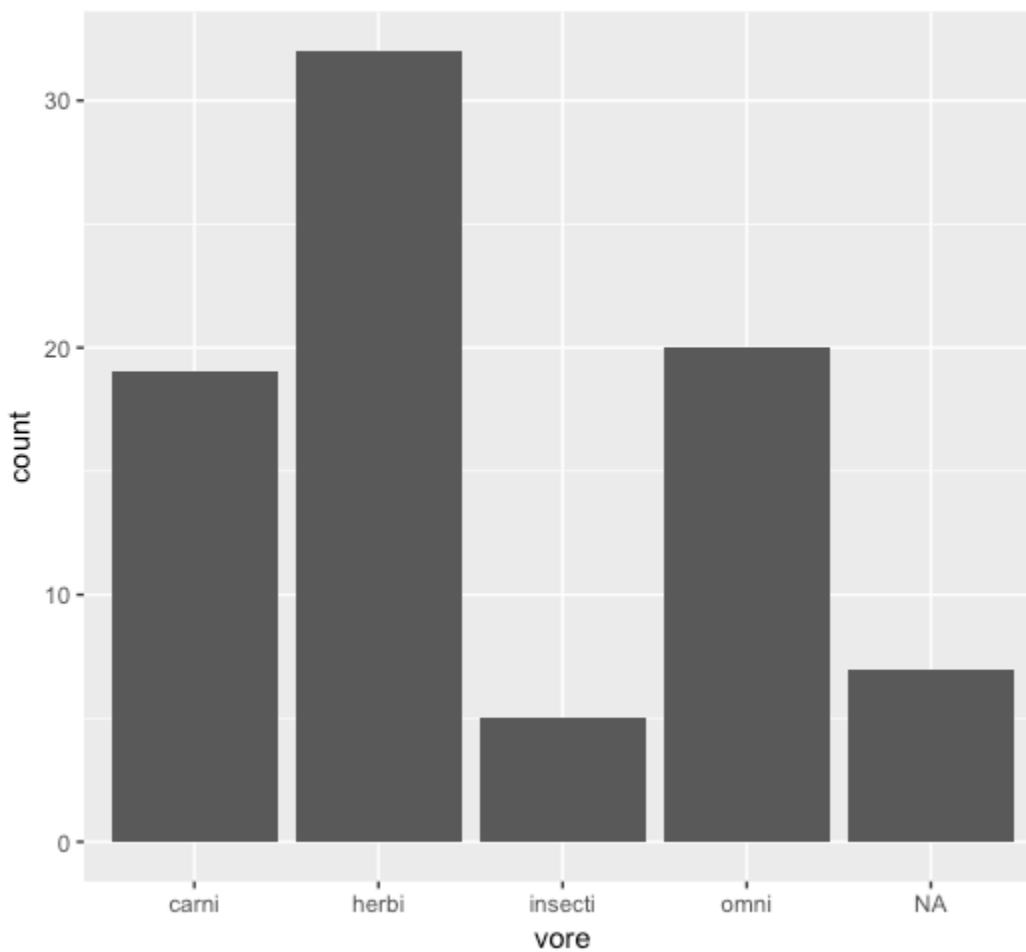
- Create a bar chart of the vore variable from msleep using geom\_bar(), as follows:

```
ggplot(msleep, aes(vore)) + geom_bar()
```

- Create the same bar chart of the vore variable from msleep using geom\_histogram(stat = "count"), as follows:

```
ggplot(msleep, aes(vore)) + geom_histogram(stat =  
"count")
```

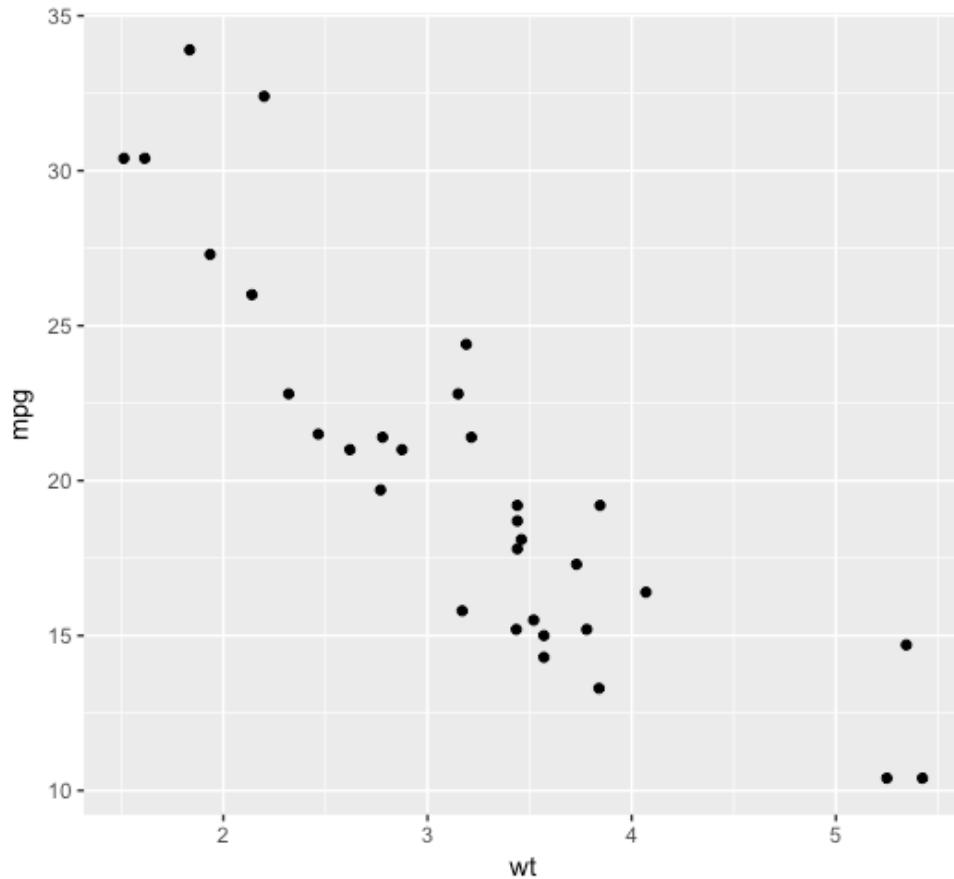




# Scatterplot

- We can create a scatterplot with the wt and mpg variables from mtcars using the following code:

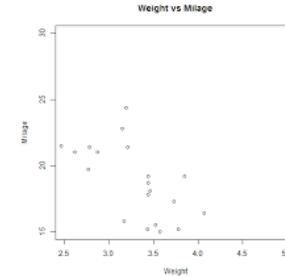
```
ggplot(mtcars, aes(wt,  
mpg)) + geom_point()
```

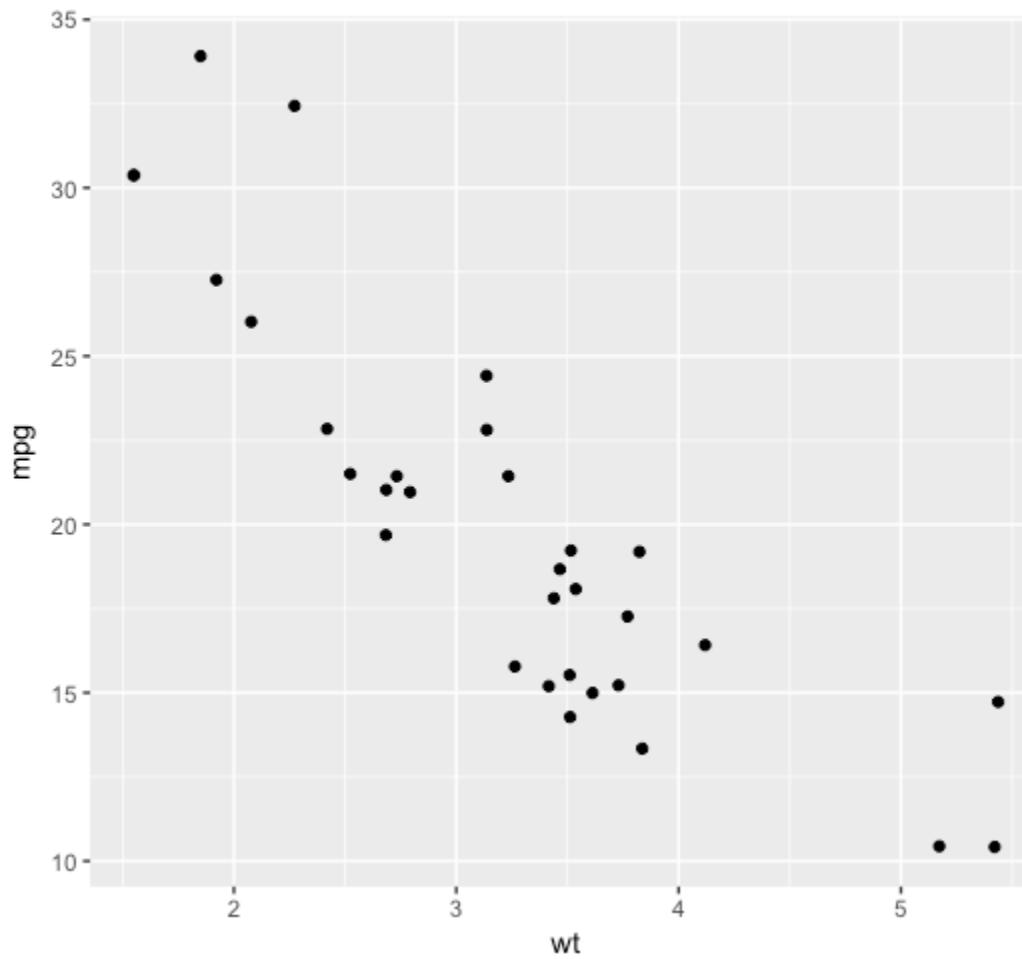


# Scatterplot

- We won't really be able to see much of an effect with this dataset, you can create a scatterplot with a bit of jitter introduced using the following code:

```
ggplot(mtcars, aes(wt, mpg)) + geom_jitter(width = 0.1)
```





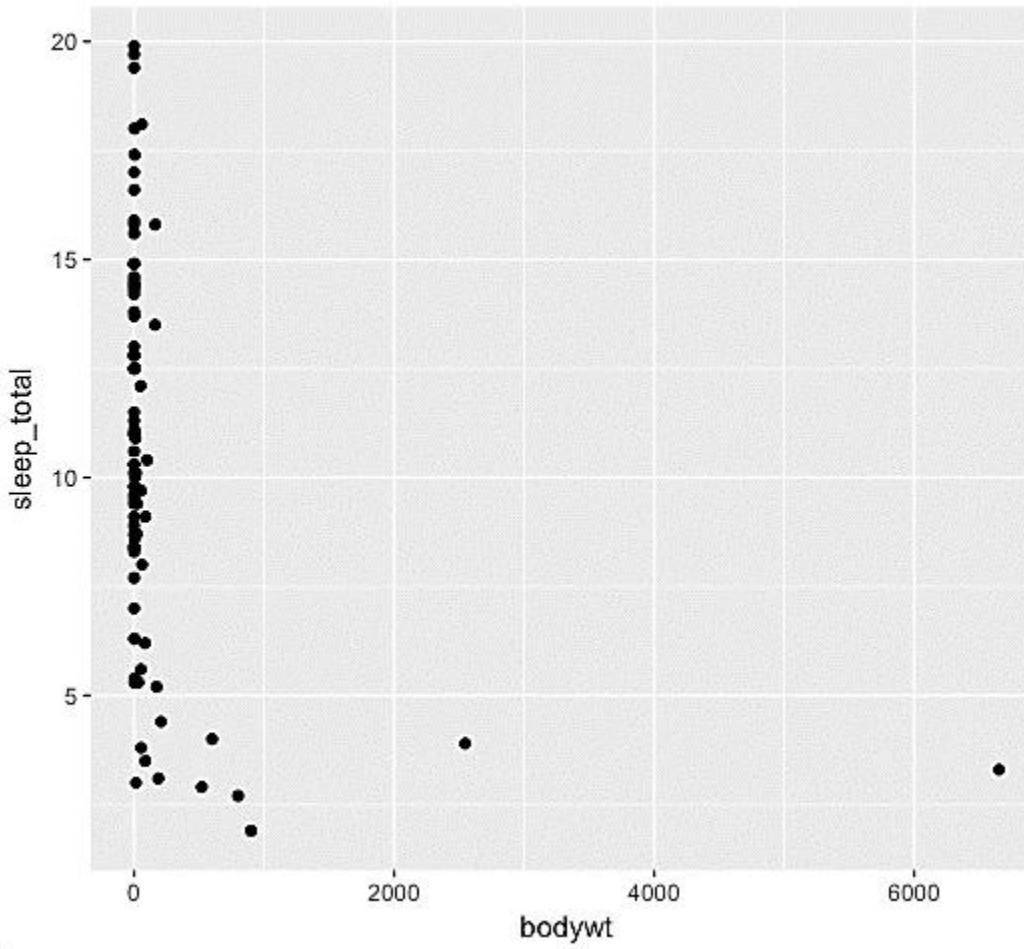
# Creating a Scatterplot of Two Continuous Variables

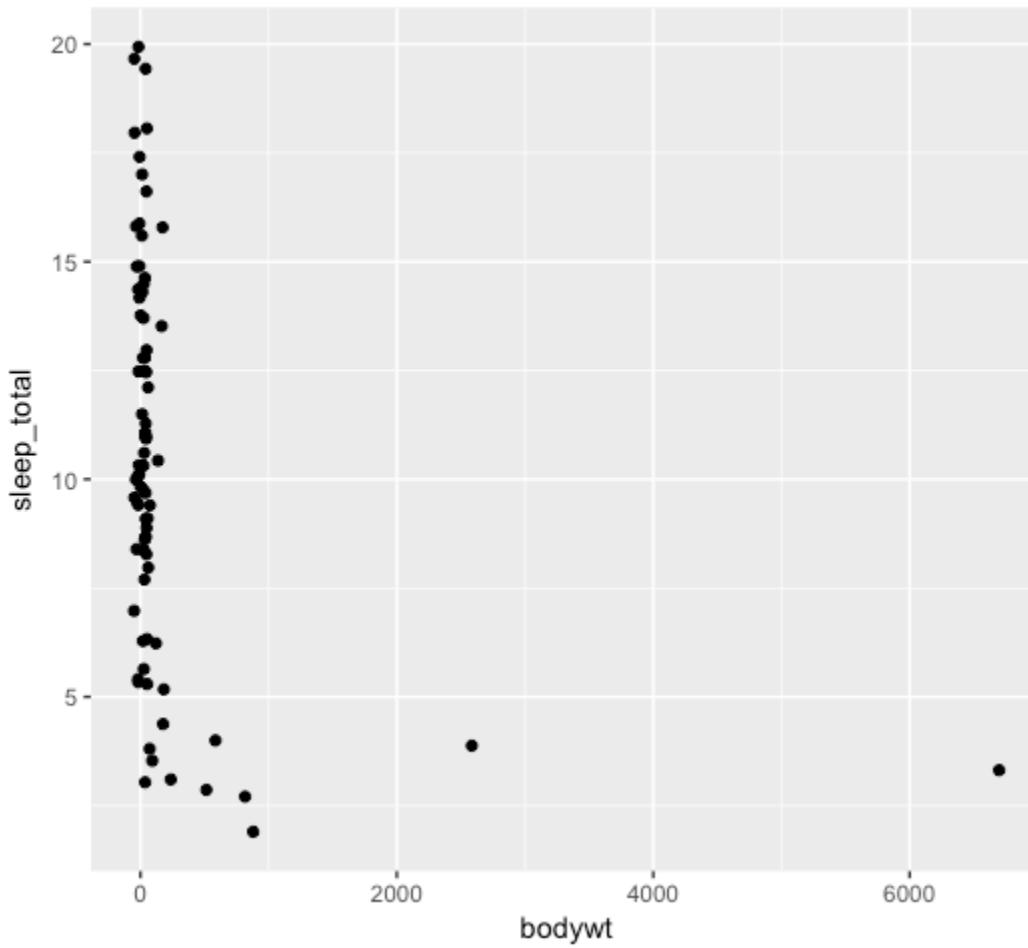
- Create a scatterplot of the bodywt and sleep\_total variables from msleep:

```
ggplot(msleep, aes(bodywt, sleep_total)) + geom_point()
```

- This scatterplot is a great candidate for using geom\_jitter(), as many of the sleep\_total observations cluster around the zero bodyweight.
- We'll use a fairly large width for jitter to really separate these points, because the scale of bodywt is in the thousands:

```
ggplot(msleep, aes(bodywt, sleep_total)) + geom_jitter(width = 50)
```





# Boxplot

- Boxplots are most appropriate when you want to check the distribution of a continuous y variable with some categorical (factor) x variable.
- We cannot create a boxplot of the mpg variable with the cyl variable in mtcars using the following code:

```
ggplot(mtcars, aes(cyl, mpg)) + geom_boxplot()
```

- We will get a warning as follows:

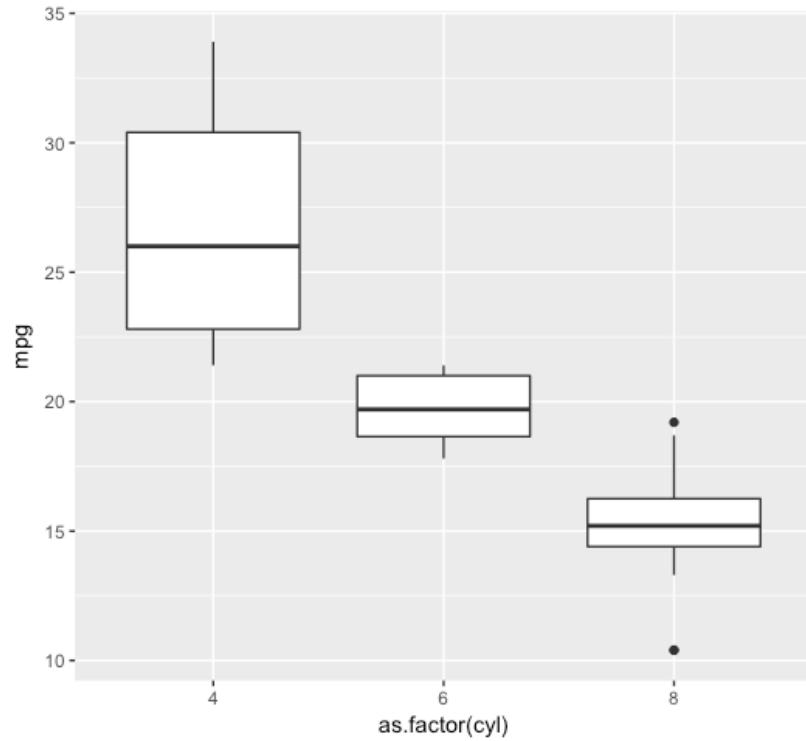
Warning message: Continuous x aesthetic -- did you forget aes(group=...)?

# Boxplot

- The following code, which transforms cyl into a factor variable using `as.factor()`, will fix it and plot the boxplot correctly:

```
ggplot(mtcars, aes(as.factor(cyl),  
mpg)) + geom_boxplot()
```

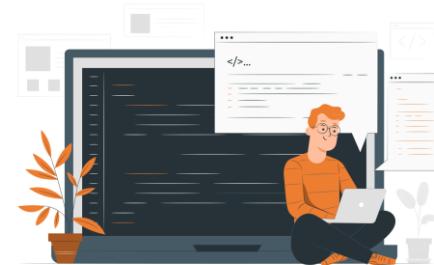
- Thus, we get the following graph as an output:

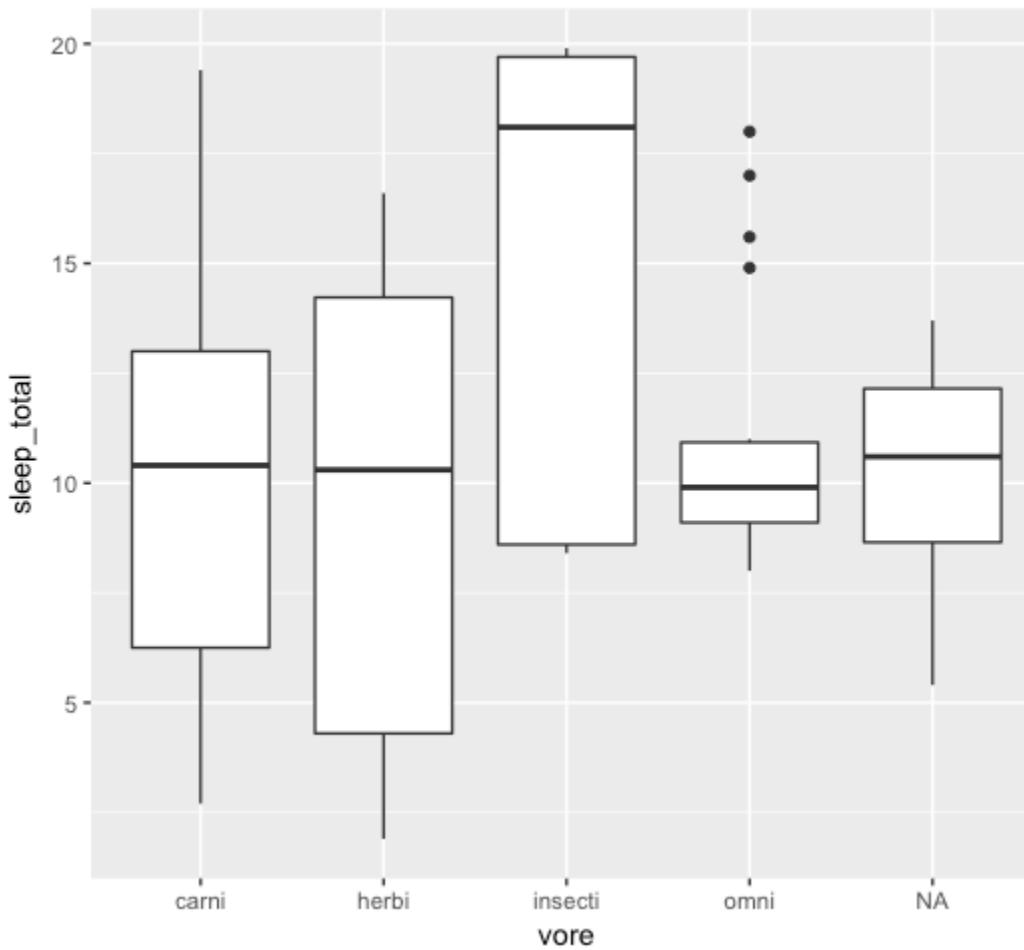


# Creating Boxplots using ggplot2

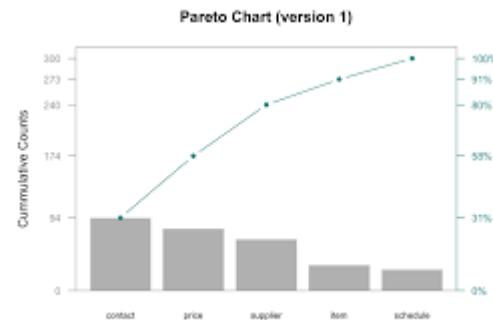
- Create a boxplot of sleep\_total with vore, both variables from the msleep dataset using the following code:

```
ggplot(msleep, aes(vore, sleep_total)) + geom_boxplot()
```





# Complete Activity: Recreating Plots Using ggplot2



# Digging into aes()

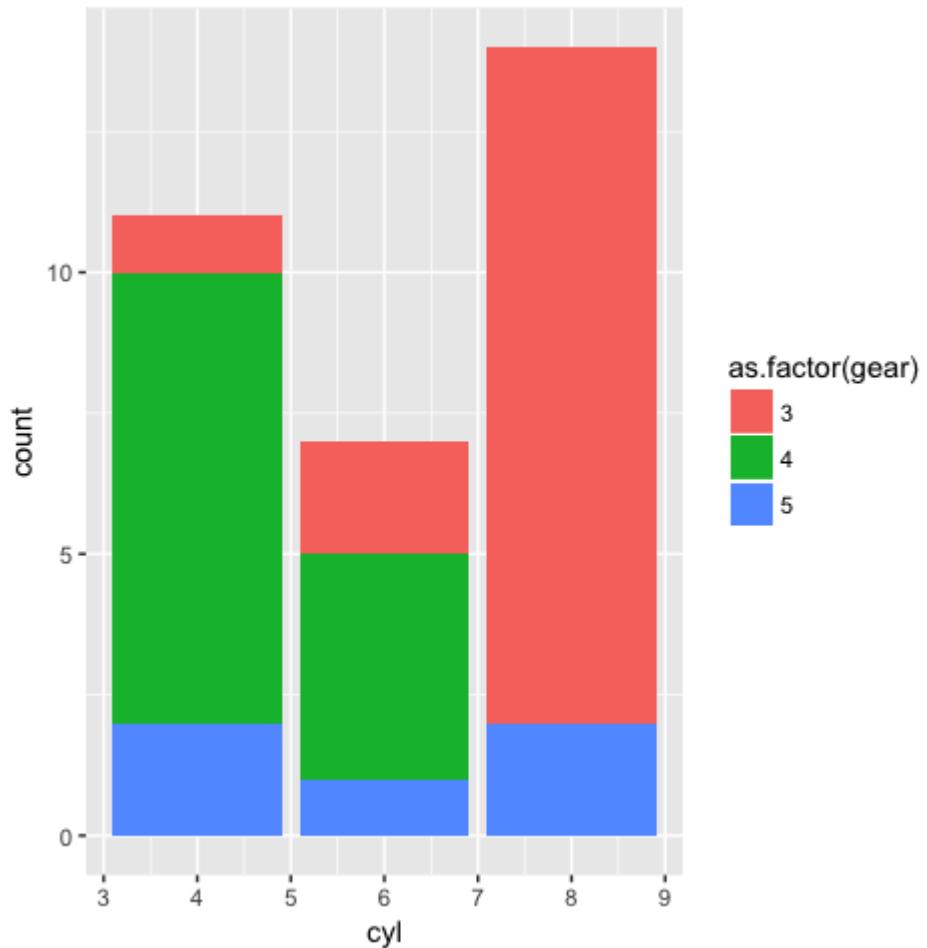
- The following code is a bar chart of how many cars have each number of cylinders, where fill is the number of gears the car has, all from mtcars:

```
ggplot(mtcars, aes(cyl, fill = as.factor(gear))) +  
  geom_bar()
```



# Digging into aes()

- A legend appears to let us know which color corresponds to which number of gears, as shown in the following graph:



# Digging into aes()

- If, instead, you were looking to make all of the bars light blue, seeing the preceding code, you might be tempted to run the following code:

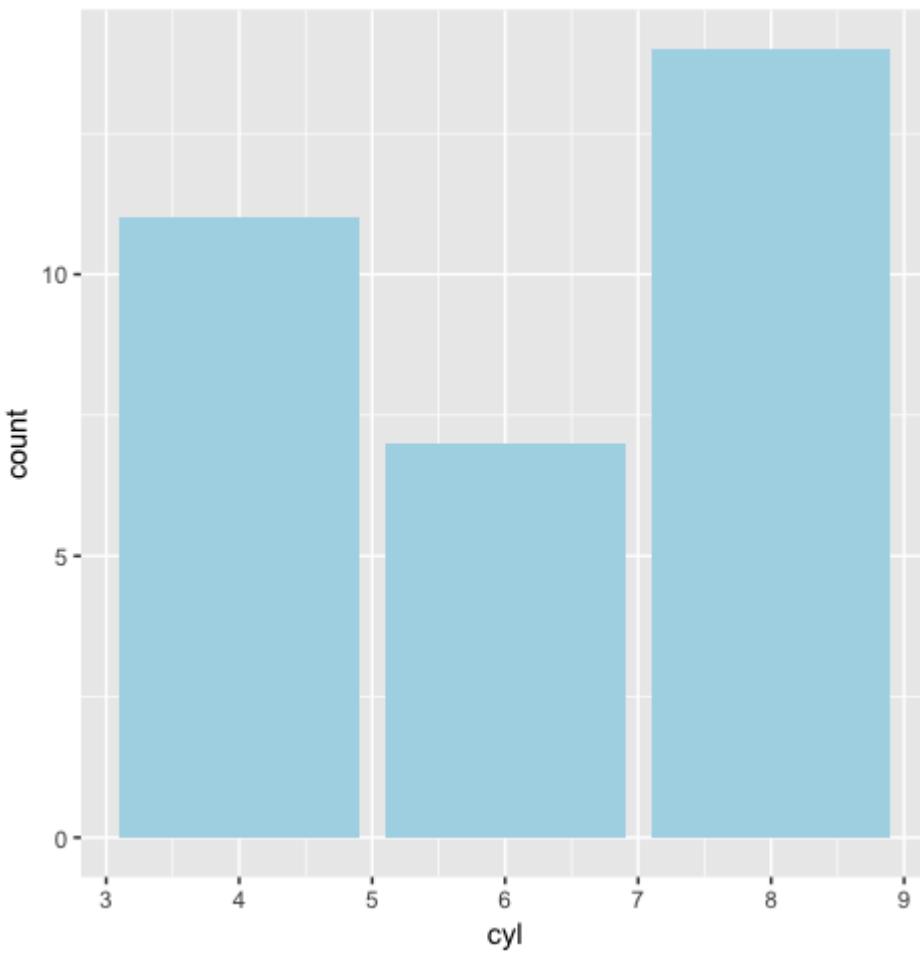
```
ggplot(mtcars, aes(cyl, fill = "lightblue")) + geom_bar()
```

- Or even the following code:

```
ggplot(mtcars, aes(cyl, fill = lightblue)) + geom_bar()
```

- To actually fill the bars light blue, you should use:

```
ggplot(mtcars, aes(cyl)) + geom_bar(fill = "lightblue")
```



# Bar Chart

- To make these charts better, we're going to convert the cyl and gear variables in mtcars to factor variables using the following code:

```
mtcars$cylfactor <- as.factor(mtcars$cyl)  
mtcars$gearfactor <- as.factor(mtcars$gear)
```

# Bar Chart

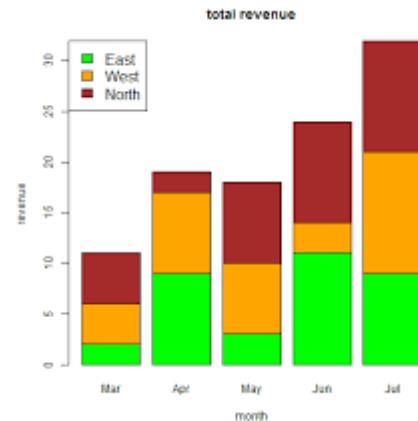
- We previously saw how to both automatically change the color of a bar chart (when we made them light blue) and also how to fill a bar chart with another variable.
- We did this using the following code:

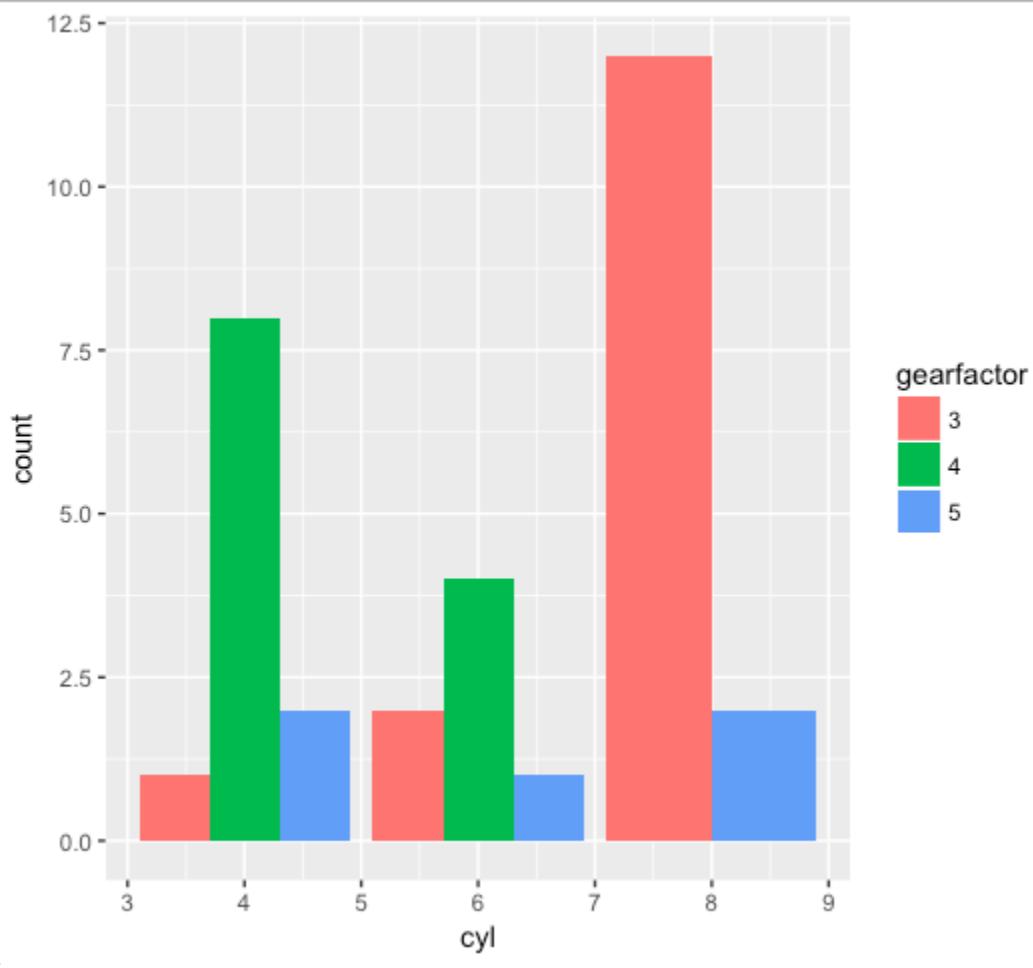
```
ggplot(mtcars, aes(cyl, fill = gearfactor)) + geom_bar()
```

# Bar Chart

- If we want the bars to be next to each other instead, we can add position = "dodge" inside geom\_bar(), with the following code:

```
ggplot(mtcars, aes(cyl, fill = gearfactor)) +  
  geom_bar(position = "dodge")
```



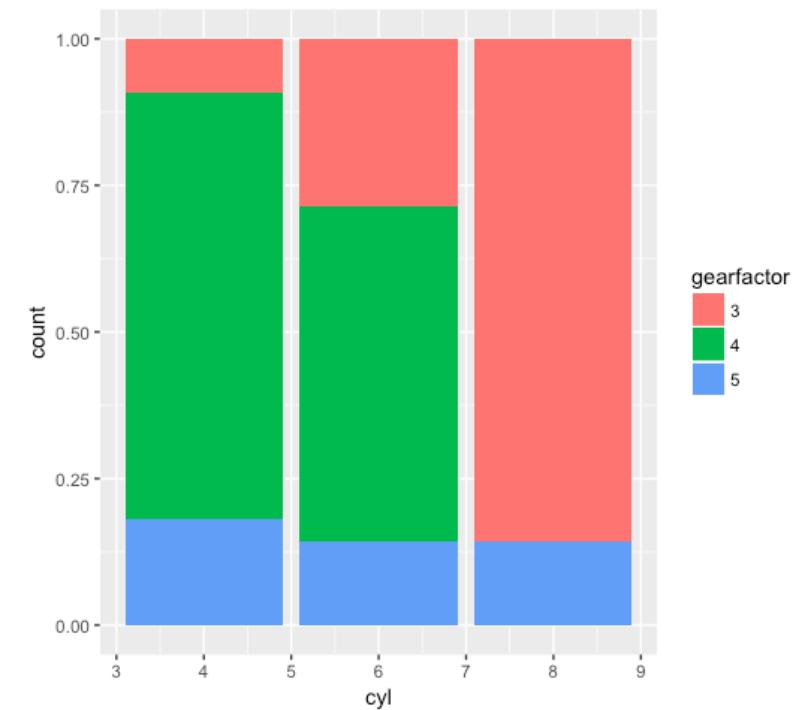


# Bar Chart

- If we want the bars to reflect percentages instead of representing the count of cars with a certain number of gears and cylinders, we can add position = "fill" inside geom\_bar():

```
ggplot(mtcars, aes(cyl, fill = gearfactor))  
+ geom_bar(position = "fill")
```

- The output we get is as follows:



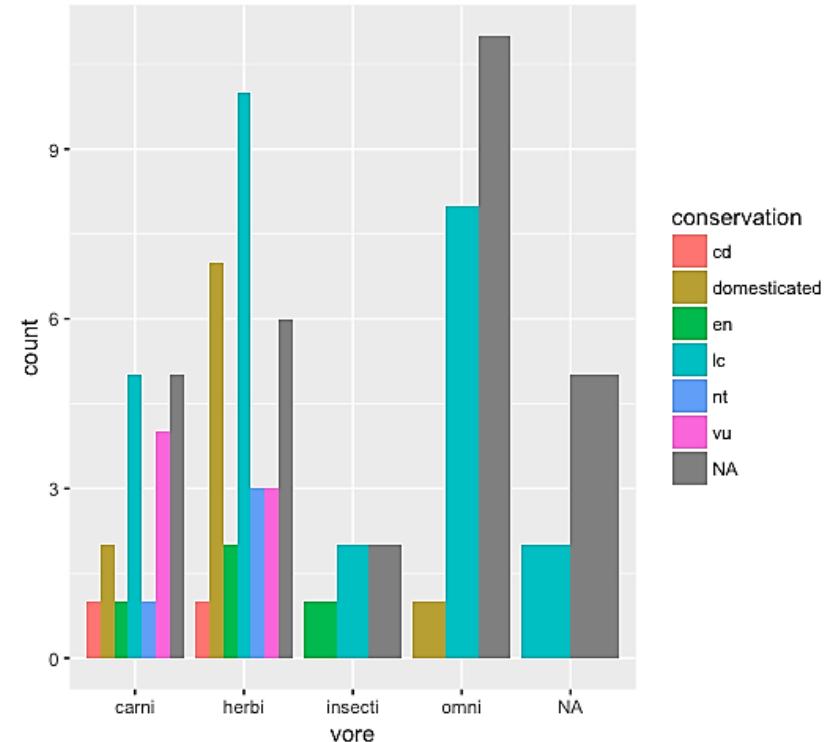
# Using Different Bar Chart Aesthetic Options

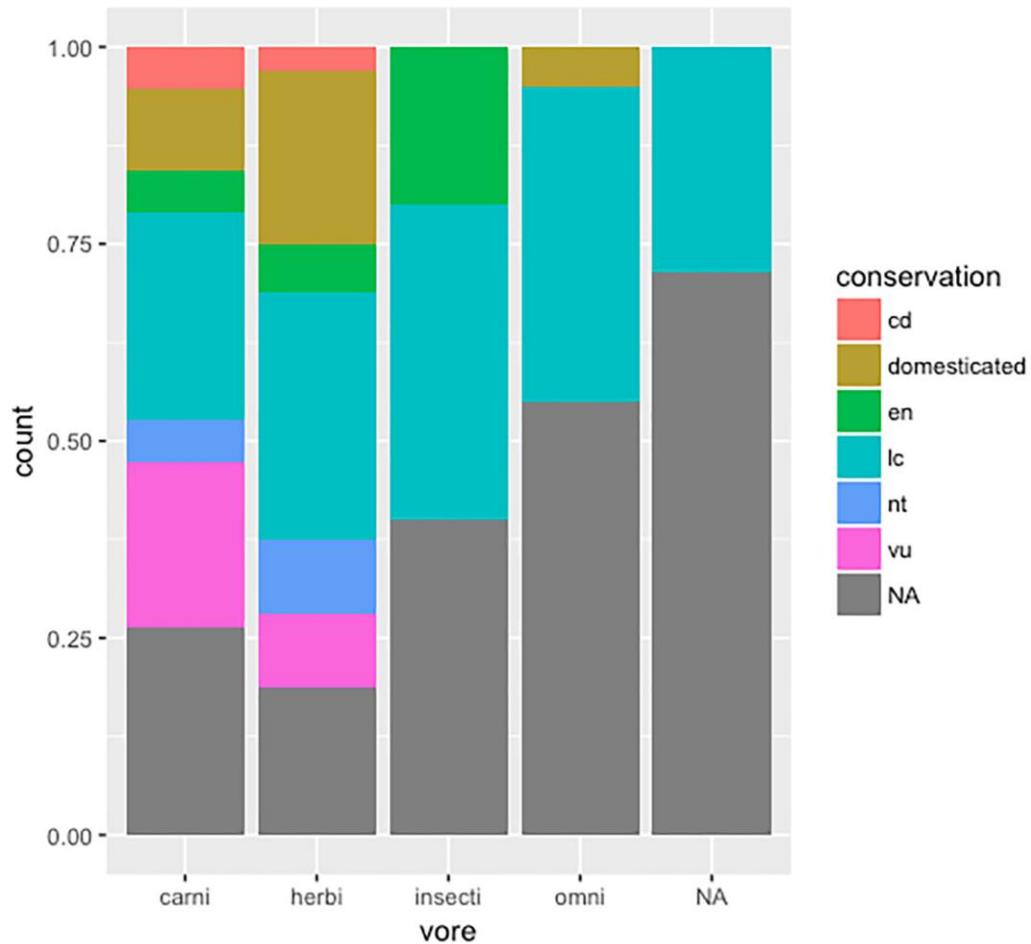
- Create a bar chart using the dodge position aesthetic of vore, filled with the conservation variable. (These variables are already declared as factor variables when you load msleep.)
- The code for this is as follows:

```
ggplot(msleep, aes(vore, fill = conservation)) +  
  geom_bar(position = "dodge")
```

# Using Different Bar Chart Aesthetic Options

- Create a bar chart with the same variables, this time using the fill position aesthetic:  
`ggplot(msleep, aes(vore, fill = conservation)) + geom_bar(position = "fill")`
- Output: The following is the output we get when we execute the code mentioned in Step 1:





# Facet Wrapping and Gridding

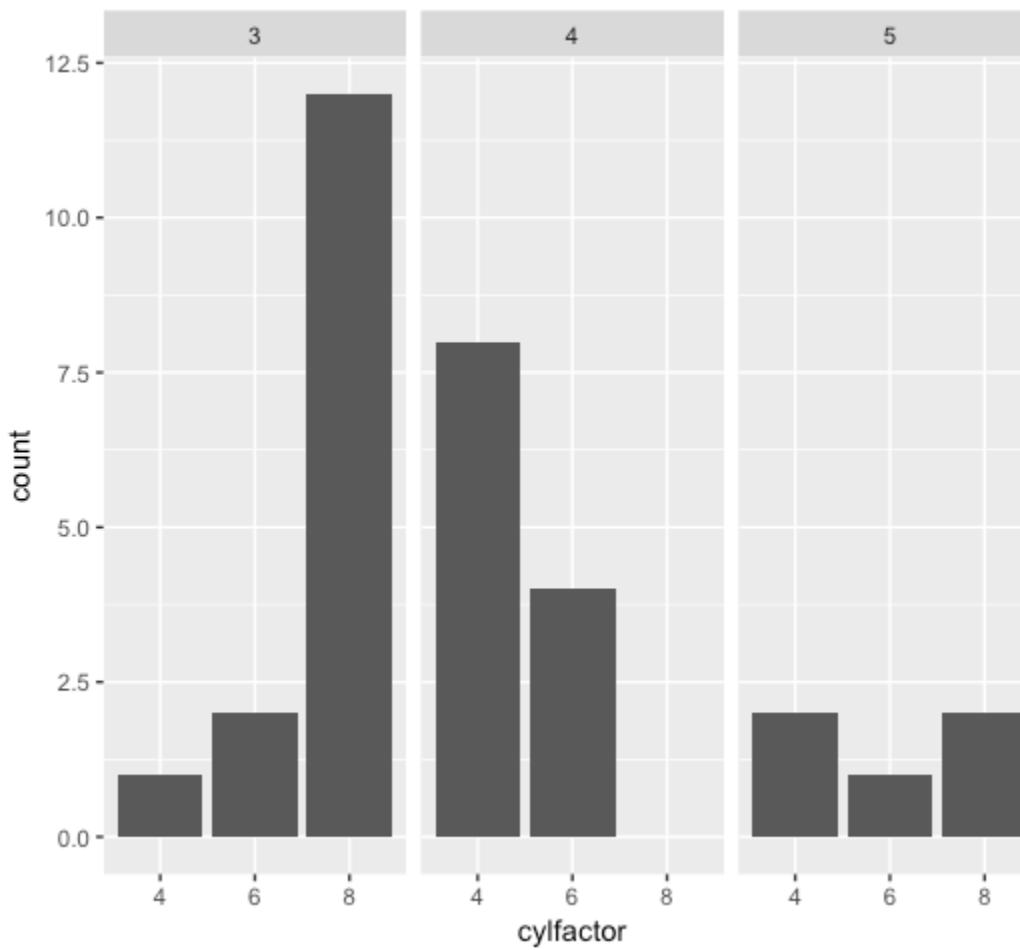
- Facet wrapping and gridding can be applied to any ggplot, not just bar charts.
- Facet wrapping will split the base ggplot (which, here, is the count of cars with each number of cylinders) by a second variable, which, here, will be the number of gears, generating three plots.
- The code for this is as follows:

```
ggplot(mtcars, aes(cylfactor)) + geom_bar() +  
facet_wrap(~gear)
```

# Facet Wrapping and Gridding

- Facet gridding is closely related to facet wrapping but allows for gridding by (row ~ column).
- The following code will generate the same as the preceding facet wrapping code, as gear is in the column place:

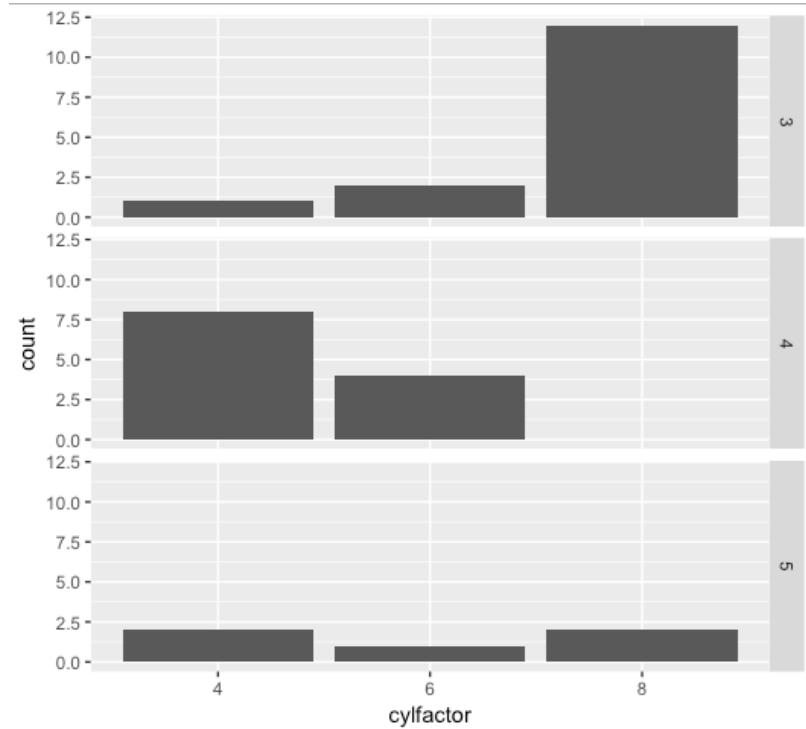
```
ggplot(mtcars, aes(cylfactor)) + geom_bar() +  
facet_wrap(~gear)
```



# Facet Wrapping and Gridding

- However, if you move gear to the row place, it will grid the plots by row instead of column, as shown in the following code:

```
ggplot(mtcars, aes(cylfactor)) +  
  geom_bar() + facet_grid(gear~)
```



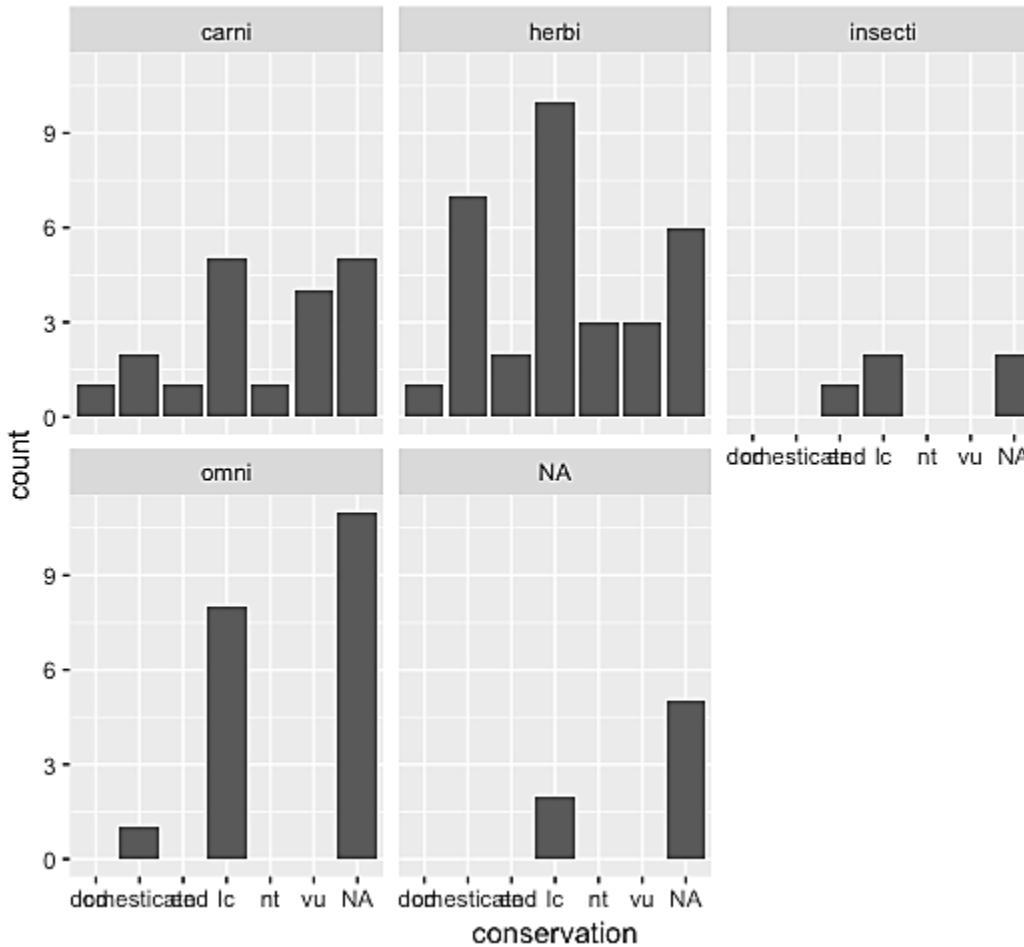
# Utilizing Facet Wrapping and Gridding to Visualize Data Effectively

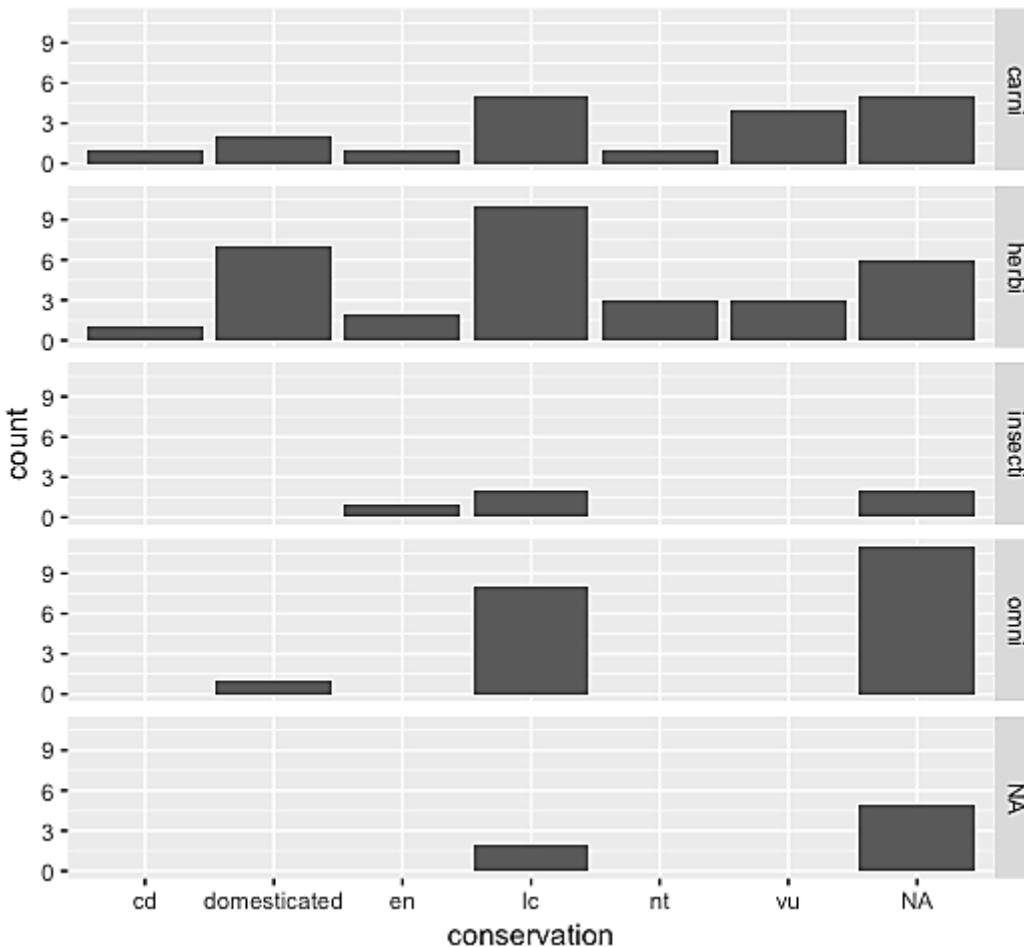
- Create a bar chart of conservation, facet wrapped by vore, both variables from the msleep dataset, as shown in the following code:

```
ggplot(msleep, aes(conservation)) + geom_bar() +  
facet_wrap(~vore)
```

- Create the same bar chart, but use facet\_grid() to grid the charts by row instead of column, as shown in the following code:

```
ggplot(msleep, aes(conservation)) + geom_bar() +  
facet_grid(vore~)
```





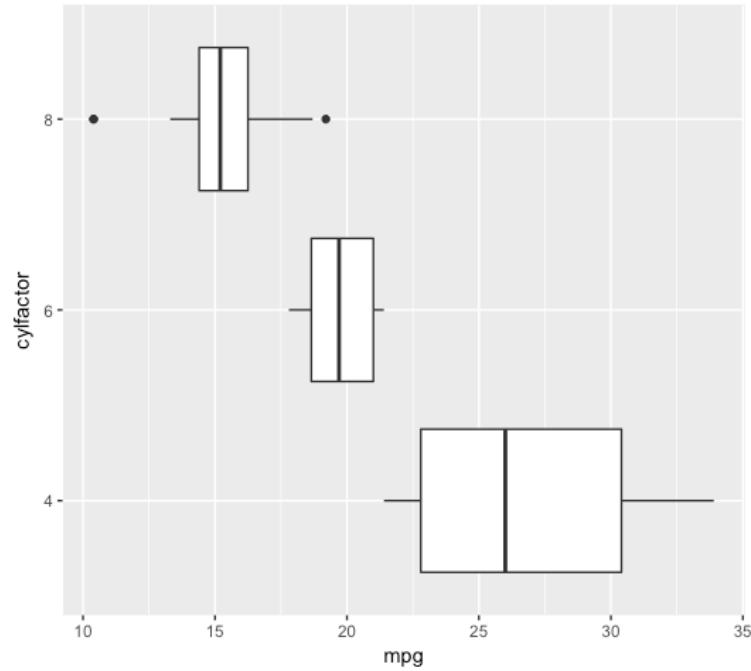
# Boxplot + coord\_flip()

- Let's return to our boxplot example from mtcars, which shows the distribution of mpg by the number of cylinders.
- We modify the code and add coord\_flip() as follows:

```
ggplot(mtcars, aes(cylfactor, mpg)) + geom_boxplot() +  
coord_flip()
```

# Boxplot + coord\_flip()

- The output we get will be as shown in the following screenshot:



# Scatterplot

- We'll return to our mtcars example of plotting mpg versus wt.
- If you recall the code used to build boxplots, you might be tempted to color the scatterplots using fill = cylfactor and code that looks like this:

```
ggplot(mtcars, aes(wt, mpg, fill = cylfactor)) +  
  geom_point()
```



# Scatterplot

- If you thought to yourself We need to use col = cylfactor in that aes() call, then you're absolutely right.
- The code should be:

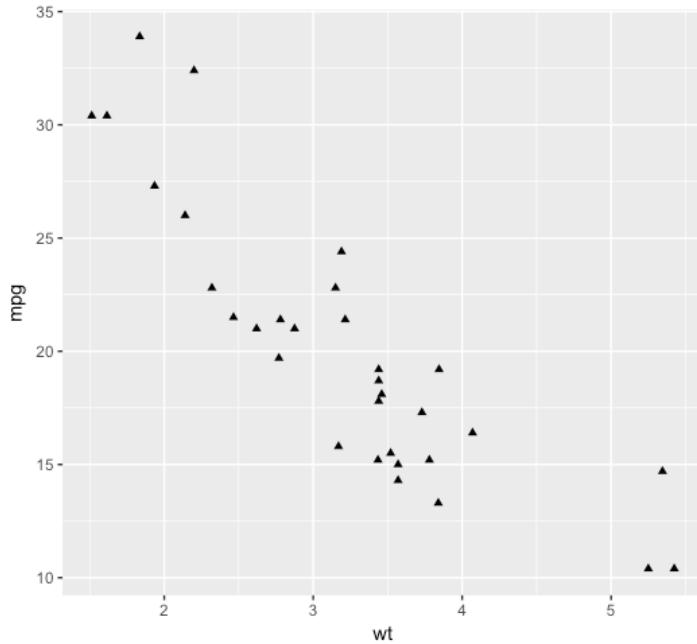
```
ggplot(mtcars, aes(wt, mpg, col = cylfactor)) + geom_point()
```

- The code will also run if you spell out col as color. We can change the shape of the points inside a scatterplot with the shape aesthetic. shape = 17 makes all the points into little triangles.

- The code is as follows:

```
ggplot(mtcars, aes(wt, mpg)) + geom_point(shape = 17)
```

# Scatterplot



- However, these are very tiny.
- Let's make them bigger with the size options, which we'll also specify inside of geom\_point() itself.
- Here is the code for it:

```
ggplot(mtcars, aes(wt, mpg)) +  
  geom_point(shape = 17, size = 3)
```

# Scatterplot

- Let's start with alpha = 0.6 and then adjust as needed.
- Here is the code for that:

```
ggplot(mtcars, aes(wt, mpg)) + geom_point(shape = 17,  
size = 3, alpha = 0.6)
```



# Scatterplot

## Utilizing Different Aesthetics for Scatterplots, Including Shapes, Colors, and Transparencies

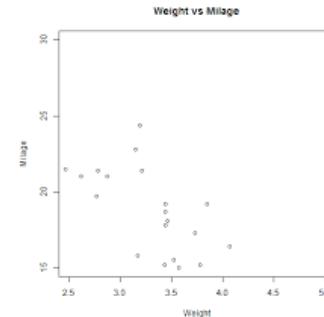
- To make these scatterplots more visually appealing, load dplyr and remove the two observations with a bodywt greater than 2000, creating the msleep2 dataset, as follows:

```
library(dplyr) msleep2 <- msleep %>% filter(bodywt < 2000)
```

# Scatterplot

- Now, create a scatterplot of bodywt versus brainwt, using triangles for the points.
- You will see an error in your console window saying that it removed rows with missing values.
- Don't worry about this for now; missing data isn't the focus of this exercise.

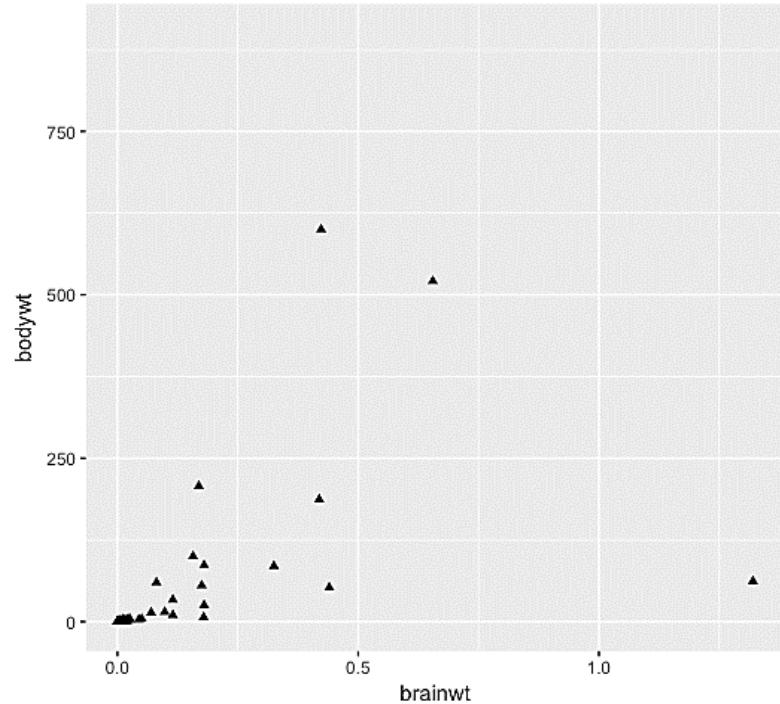
```
ggplot(msleep2, aes(brainwt, bodywt)) +  
  geom_point(shape = 17)
```

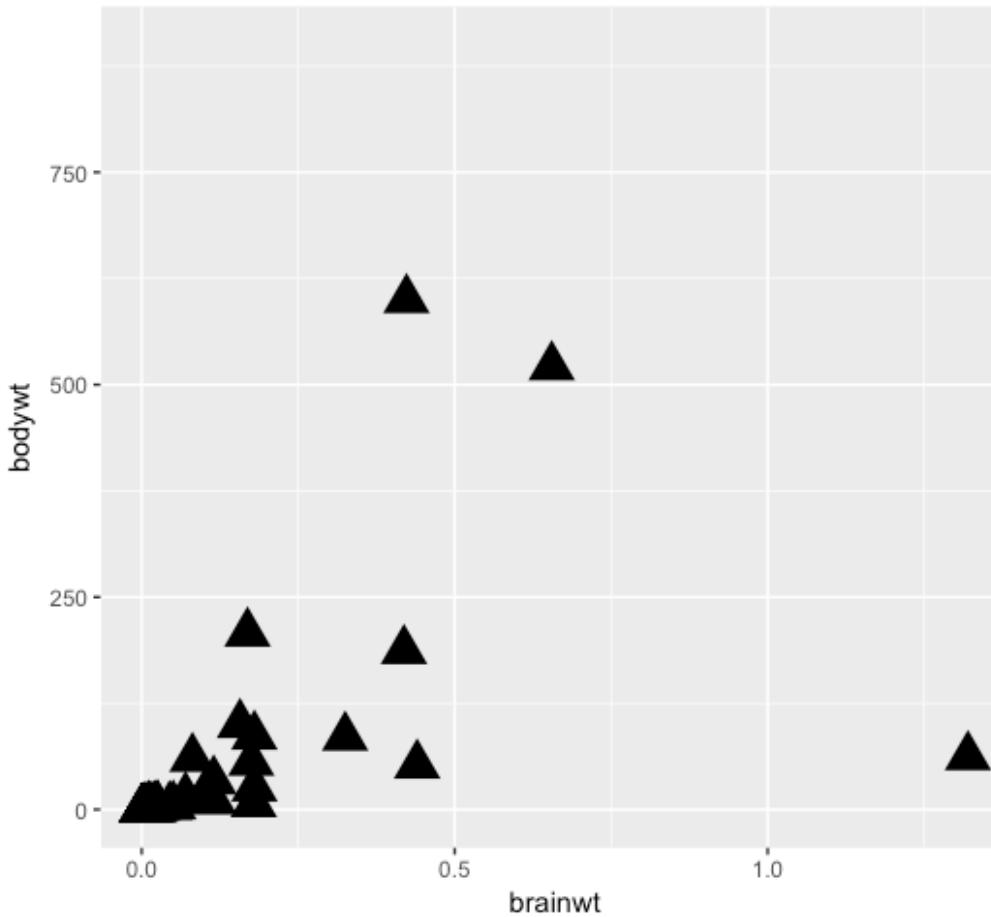


# Scatterplot

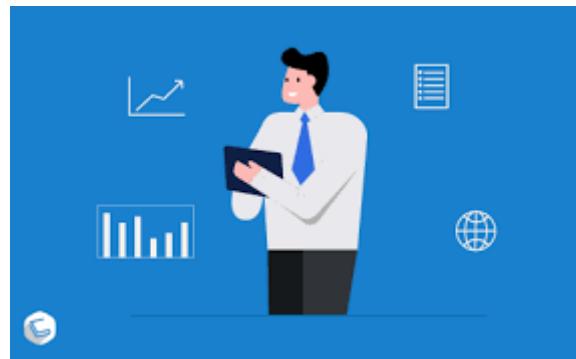
- Create the same scatterplot but make the triangles much bigger.

```
ggplot(msleep2, aes(brainwt,  
bodywt)) + geom_point(shape  
= 17, size = 6)
```





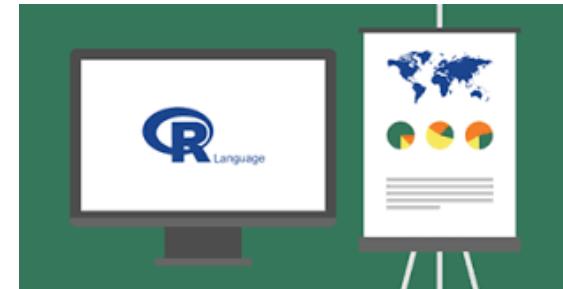
# Complete Activity: Utilizing ggplot2 Aesthetics



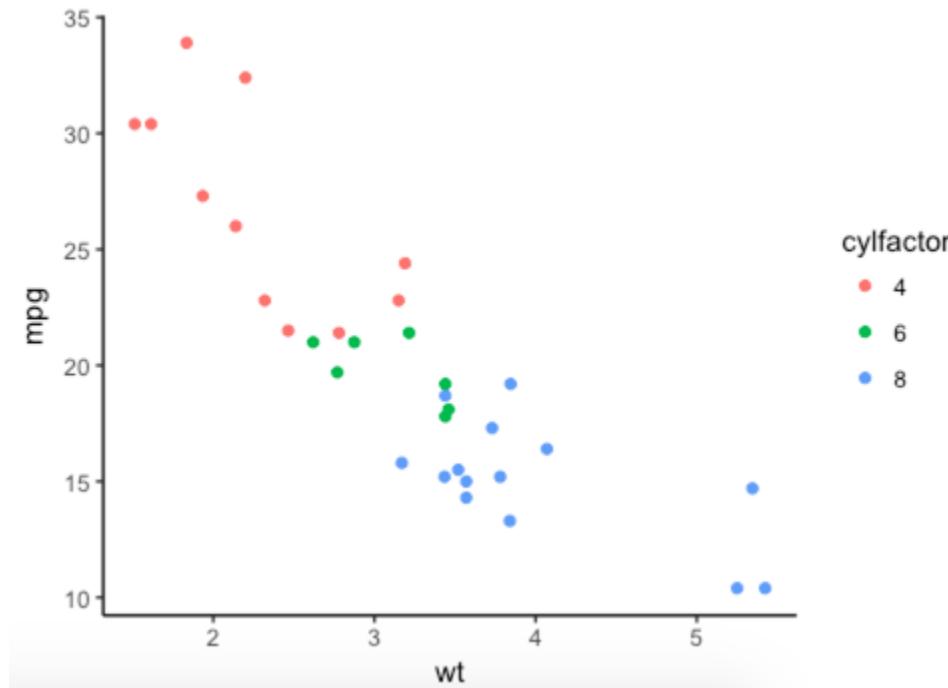
# Extending the Plots with Titles, Axis Labels, & Themes

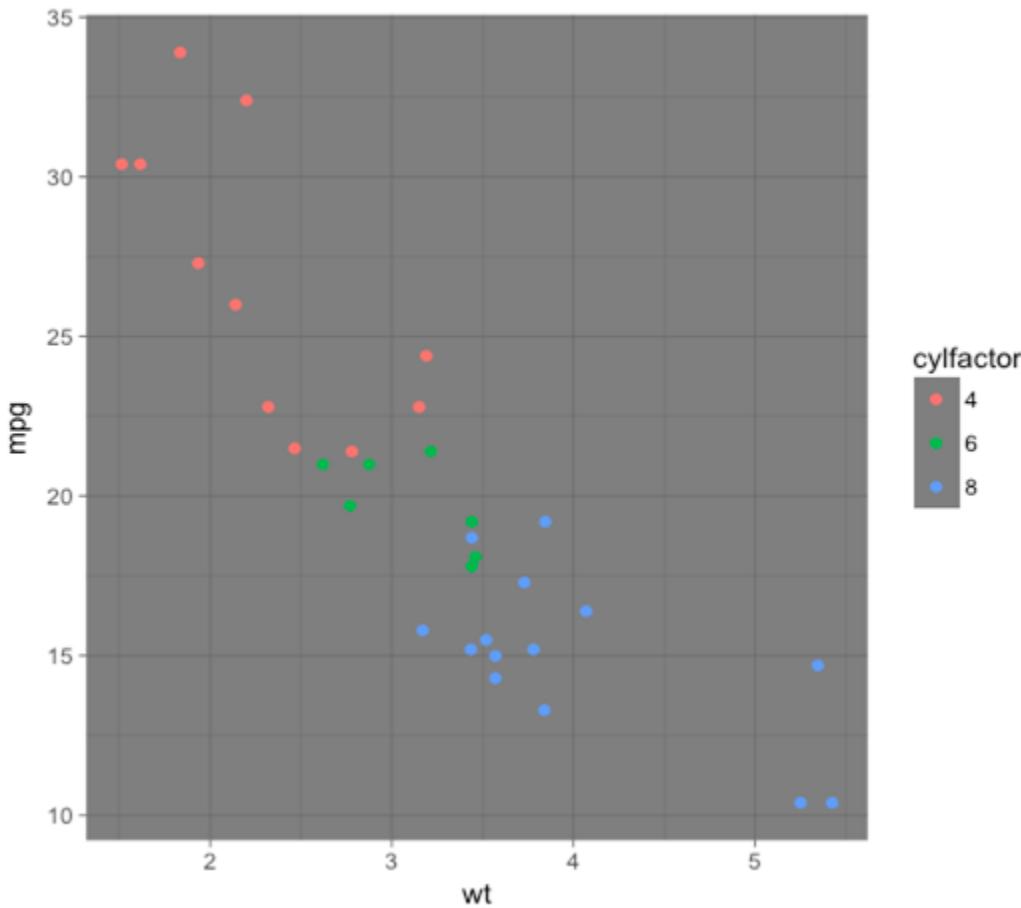
- `labs()` takes as an argument anything you'd like to change the label of, such as title, subtitle, x, y, caption, or even the label of the legend.
- It is often used as follows:

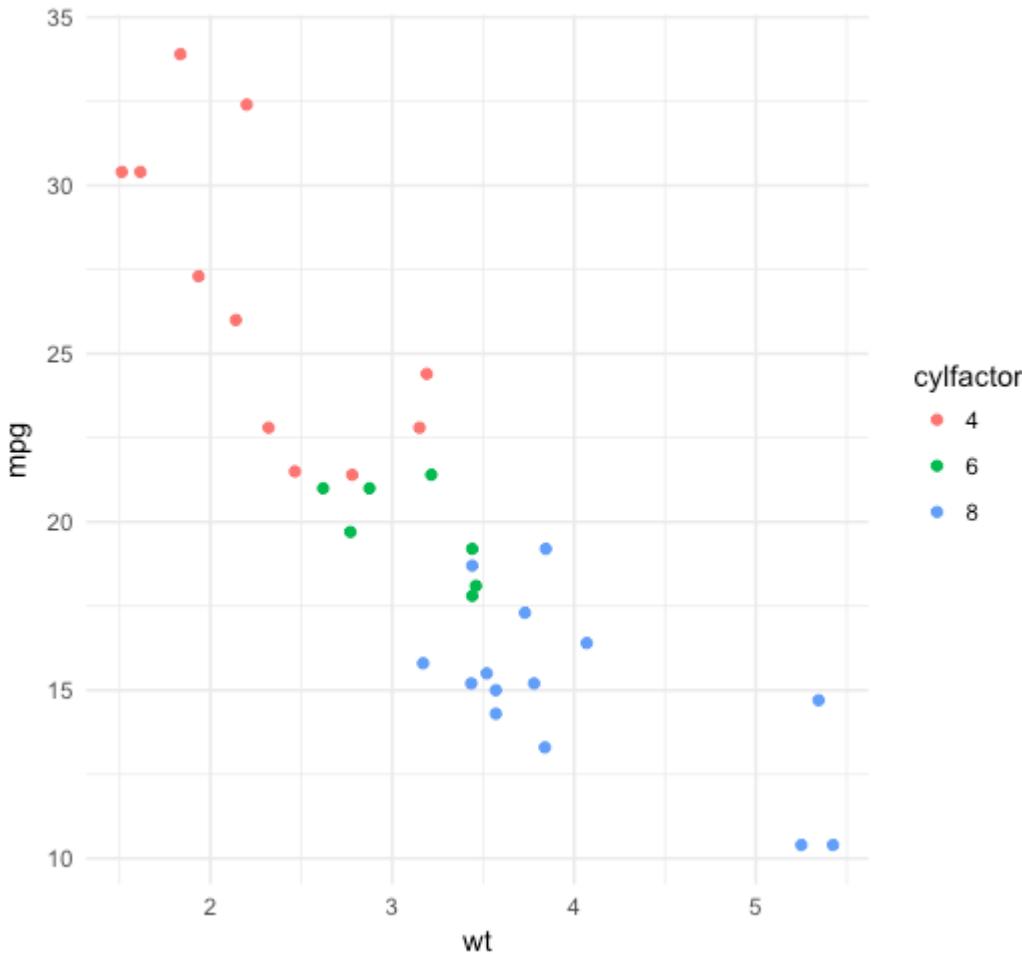
```
mtcars_ggplot + geom_point() +  
  labs(title = "mpg vs. wt",  
       subtitle = "mtcars dataset",  
       x = "weight",  
       caption = "decreasing linear trend")
```



# Extending the Plots with Titles, Axis Labels, & Themes





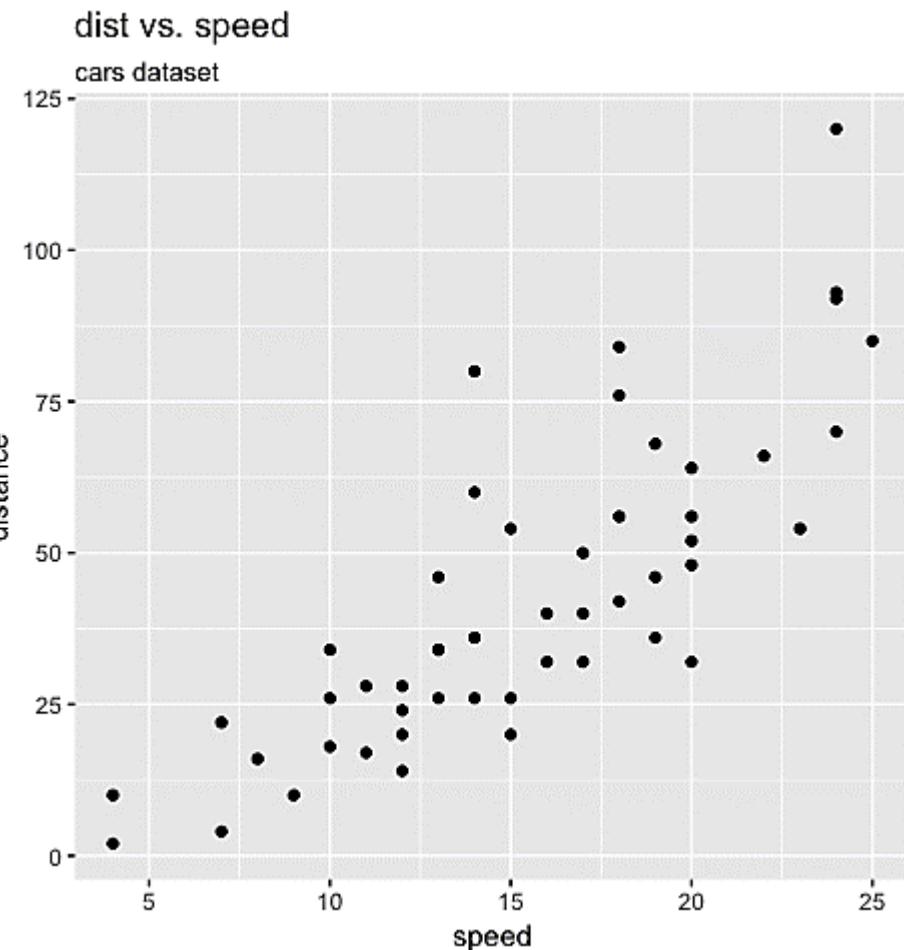


# Extending the Plots with Titles, Axis Labels, & Themes

- Try to recreate the ggplots shown as follows. :
- Execute the following code:

```
library(ggplot2)
ggplot(cars, aes(speed, dist)) + geom_point() +
  labs(title = "dist vs. speed",
       subtitle = "cars dataset",
       y = "distance")
```



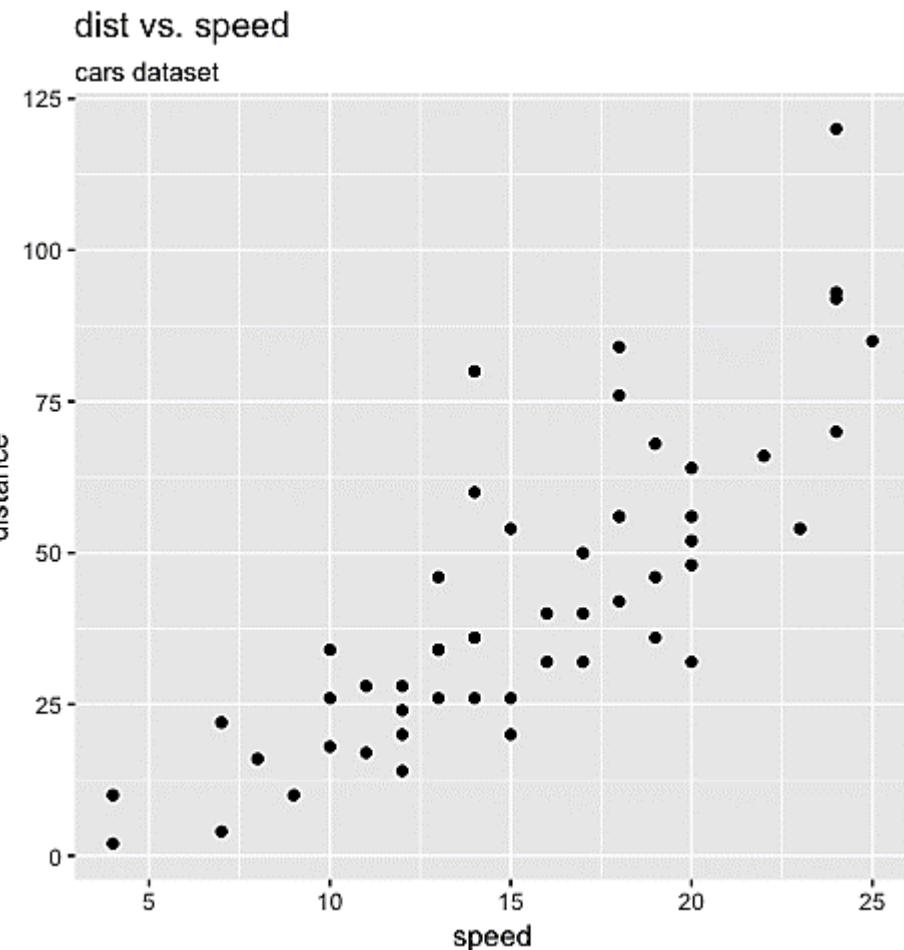


# Extending the Plots with Titles, Axis Labels, & Themes

- Execute the following code:

```
ggplot(cars, aes(speed, dist)) + geom_point() +  
  ggtitle("dist vs. speed",  
          subtitle = "cars dataset") +  
  ylab("distance")
```

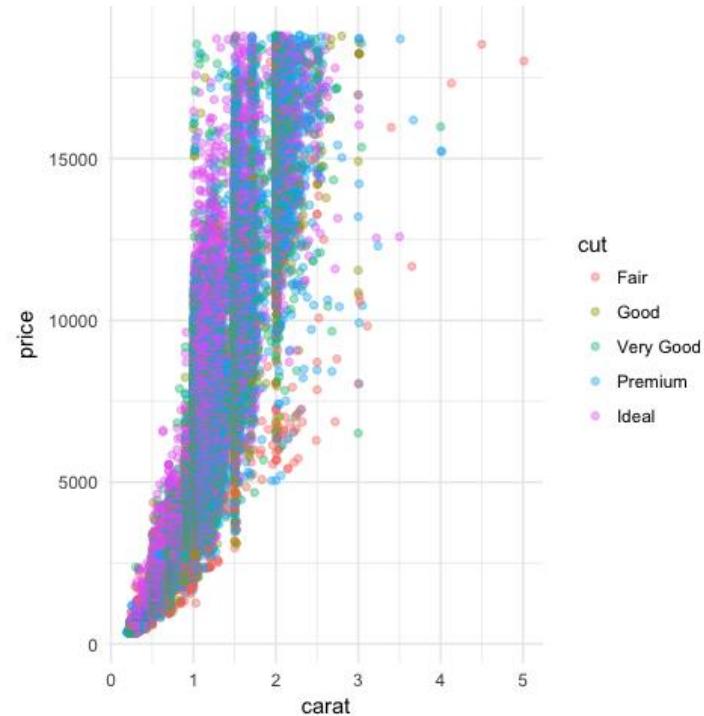




# Extending the Plots with Titles, Axis Labels, & Themes

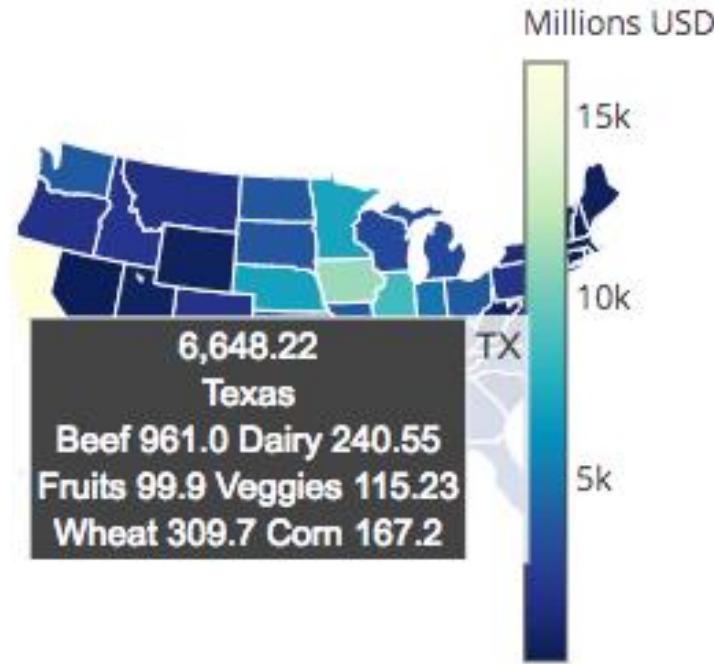
- Execute the following code:  

```
ggplot(diamonds, aes(carat, price,  
col = cut)) + geom_point(alpha =  
0.4) + theme_minimal()
```
- The output of the code will be  
as follows:



# Interactive Plots

- Plotly is an R package designed to allow you to create interactive plots online.
- It integrates with ggplot2



# Shiny

- Shiny is a product built by RStudio that allows you to build interactive web apps straight out of RStudio.
- They can be used on the web, embedded into R Markdown documents, and like Plotly charts, can be extended with CSS, HTML, and JavaScript.

# Exploring Shiny and Plotly

- To explore the Shiny and Plotly tools, in your web browser, navigate to the following URLs:

<https://shiny.rstudio.com/gallery/>

<https://plot.ly/r/>



# Summary

- Graphing in R will be crucial in your data science work, and we have covered most of the basics here.
- However, graphing is one of those things where, most of the time, there are always going to be different types of graphs you haven't heard of yet and options you haven't yet selected
- It's important to know where to look for assistance and how to keep learning.

# 3. Data Management



# Data Management

By the end of this lesson, you will be able to:

- Create and reorder factor variables
- Generate pivot tables
- Aggregate data using the base and dplyr packages
- Use various methods to split, apply, and combine data in R
- Split character strings using the stringr package
- Merge and join different datasets using base R and the dplyr methods



# Factor Variables

- A factor variable in R is an explicitly declared categorical variable, or one that defines different categories or levels.
- Some common examples of factor variables include a variable describing sex, month, or one designating low/medium/high.



# Factor Variables

- Let's return to the mtcars and iris datasets, both of which we've used previously.
- (They're very common examples of datasets that are used in R, if you haven't caught on to that yet!) After loading, let's examine each dataset with the method `str()`, as follows:

```
data("mtcars")
str(mtcars)
data("iris")
str(iris)
```



# Factor Variables

- mtcars has no factor variables specified out of the box, but the Species variable in the iris dataset is explicitly declared to be a factor variable with three levels: setosa, versicolor, and, if we could see it, virginica.
- We can see all three by using the levels() function, as shown in the following screenshot:

```
> levels(iris$Species)
[1] "setosa"     "versicolor" "virginica"
> |
```

# Factor Variables

- For example, let's build a linear regression model to examine the relationship between the number of cylinders (cyl) and miles per gallon (mpg) in cars in the mtcars dataset.
- We'll use cyl both as an integer variable and as a factor variable.
- We can use cyl as an integer variable as follows:

```
summary(lm(mpg ~ cyl, data = mtcars))
```

# Factor Variables

Call:

```
lm(formula = mpg ~ cyl, data = mtcars)
```

Residuals:

Min	1Q	Median	3Q	Max
-4.9814	-2.1185	0.2217	1.0717	7.5186

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	37.8846	2.0738	18.27	< 2e-16 ***
cyl	-2.8758	0.3224	-8.92	6.11e-10 ***

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.206 on 30 degrees of freedom

Multiple R-squared: 0.7262, Adjusted R-squared: 0.7171

F-statistic: 79.56 on 1 and 30 DF, p-value: 6.113e-10



# Factor Variables

- We can use cyl as a factor variable as follows:

```
summary(lm(mpg ~  
          as.factor(cyl), data = mtcars))
```

- The output is as:

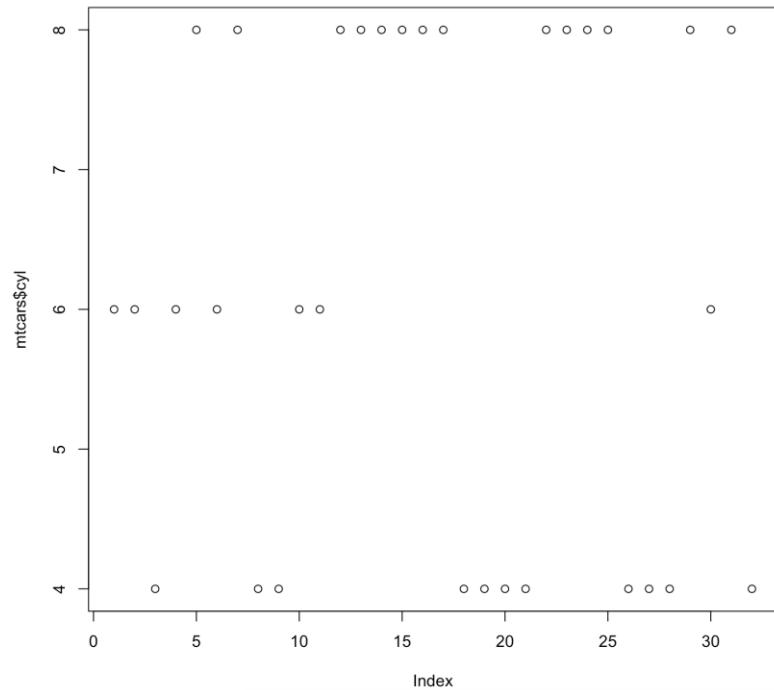
```
Call:  
lm(formula = mpg ~ as.factor(cyl), data = mtcars)  
  
Residuals:  
    Min      1Q  Median      3Q     Max  
-5.2636 -1.8357  0.0286  1.3893  7.2364  
  
Coefficients:  
              Estimate Std. Error t value Pr(>|t|)  
(Intercept) 26.6636    0.9718  27.437 < 2e-16 ***  
as.factor(cyl)6 -6.9208    1.5583  -4.441 0.000119 ***  
as.factor(cyl)8 -11.5636   1.2986  -8.905 8.57e-10 ***  
---  
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1  
  
Residual standard error: 3.223 on 29 degrees of freedom  
Multiple R-squared:  0.7325, Adjusted R-squared:  0.714  
F-statistic: 39.7 on 2 and 29 DF,  p-value: 4.979e-09
```

# Factor Variables

- We can rerun the code to plot the cyl variable without transforming it into a factor, as follows:

```
plot(mtcars$cyl)
```

- We get the scatterplot as an output, as shown in the screenshot:



# Factor Variables

- Similarly, if we try to create a graph using ggplot2, for example, by using a boxplot of mpg by cyl without transforming it into a factor, we'll get a warning:

```
> ggplot(mtcars, aes(cyl, mpg)) + geom_boxplot()
```

Warning message:

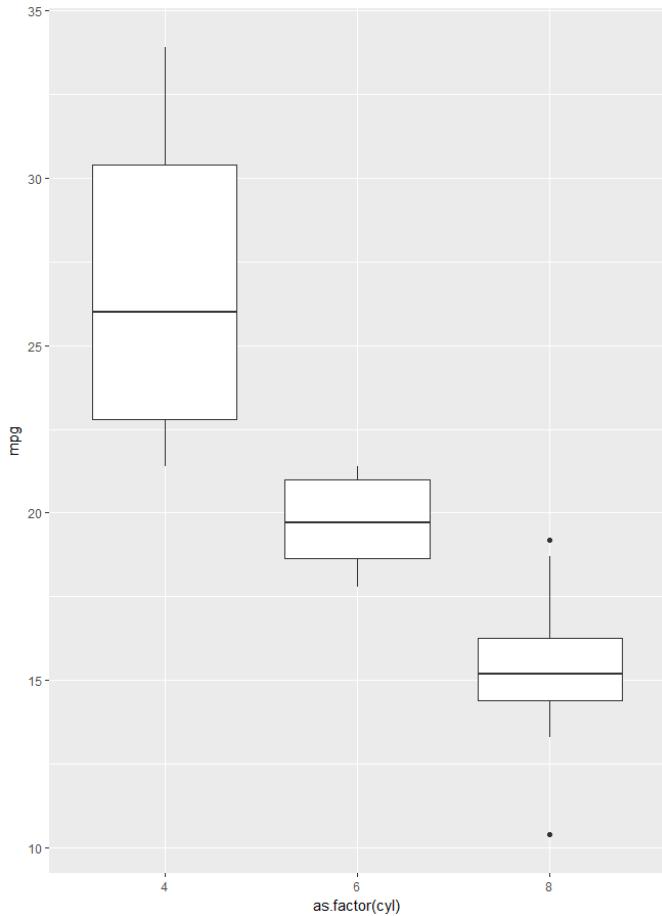
Continuous x aesthetic -- did you forget aes(group=...)?  
'

# Factor Variables

- we should change cyl into a factor variable using `as.factor()`, as follows:

```
ggplot(mtcars, aes(as.factor(cyl),  
mpg)) + geom_boxplot()
```

- Here is the boxplot we are looking for:



# Factor Variables

- Overwrite the cyl variable and create it as a factor using the following code:

```
mtcars$cyl <- as.factor(mtcars$cyl)
```

- Create a second variable, cyl2, which will be a factor version of the original cyl variable as follows:

```
mtcars$cyl2 <- as.factor(mtcars$cyl)
```



# Factor Variables

- For example, the variables cyl, am, and gear in the mtcars dataset are all categorical and should be transformed to factors.
- A good way to do this is by using the following code:

```
factors <- c("cyl", "am", "gear")
mtcars[,factors] <- data.frame(apply(mtcars[,factors], 2,
as.factor))
```

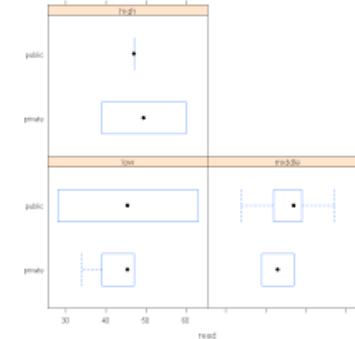
# Factor Variables

- We can check to be sure this worked by using `str()` as follows:

`str(mtcars)`

- We see that the variables cyl, am, and gear are now all factor variables, as shown in the screenshot:

```
> str(mtcars)
'data.frame': 32 obs. of 11 variables:
 $ mpg : num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : Factor w/ 3 levels "4","6","8": 2 2 1 2 3 2 3 1 1 2 ...
 $ disp: num 160 160 108 258 360 ...
 $ hp : num 110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt : num 2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num 16.5 17 18.6 19.4 17 ...
 $ vs : num 0 0 1 1 0 1 0 1 1 1 ...
 $ am : num 1 1 1 0 0 0 0 0 0 0 ...
 $ gear: Factor w/ 3 levels "3","4","5": 2 2 2 1 1 1 1 2 2 2 ...
 $ carb: Factor w/ 6 levels "1","2","3","4",..: 4 4 1 1 2 1 4 2 2 4 ...
```



# Creating Factor Variables in a Dataset

- Load the datasets library:

```
library(datasets)
```

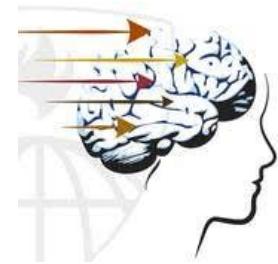
- Load the midwest dataset and examine it with str():

```
data(midwest)
```

```
str(midwest)
```

- Convert the state variable to a factor by using as.factor():

```
midwest$state <- as.factor(midwest$state)
```



# Creating Factor Variables in a Dataset

- Load the band\_instruments dataset and examine it with str():

```
data(band_instruments)  
str(band_instruments)
```

- Transform both variables in band\_instruments to factor variables using apply():

```
band_instruments <-  
  data.frame(apply(band_instruments, 2, as.factor))
```

- Double-check that Step 5 worked using str():

```
str(band_instruments)
```

# Creating Factor Variables in a Dataset

```
> str(midwest)
Classes 'tbl_df', 'tbl' and 'data.frame':      437 obs. of  28 variables:
 $ PID                  : int  561 562 563 564 565 566 567 568 569 570 ...
 $ county               : chr  "ADAMS" "ALEXANDER" "BOND" "BOONE" ...
 $ state                : chr  "IL" "IL" "IL" "IL" ...
 $ area                 : num  0.052 0.014 0.022 0.017 0.018 0.05 0.017 0.027 0.024 0
 .058 ...
 $ poptotal             : int  66090 10626 14991 30806 5836 35688 5322 16805 13437 17
 3025 ...
 $ popdensity            : num  1271 759 681 1812 324 ...
 $ popwhite              : int  63917 7054 14477 29344 5264 35157 5298 16519 13384 146
 506 ...
 $ popblack              : int  1702 3496 429 127 547 50 1 111 16 16559 ...
 $ popamerindian         : int  98 19 35 46 14 65 8 30 8 331 ...
 $ popasian              : int  249 48 16 150 5 195 15 61 23 8033 ...
```

# Creating Factor Variables in a Dataset

The following is the output of the code mentioned in Step 4:

```
> str(band_instruments)
Classes 'tbl_df', 'tbl' and 'data.frame':      3 obs. of  2 variables:
 $ name : chr  "John" "Paul" "Keith"
 $ plays: chr  "guitar" "bass" "guitar"
```



# Creating Factor Variables in a Dataset

The following is the output of the code mentioned in Step 5:

```
> str(band_instruments)
'data.frame': 3 obs. of 2 variables:
 $ name : Factor w/ 3 levels "John","Keith",...: 1 3 2
 $ plays: Factor w/ 2 levels "bass","guitar": 2 1 2
```



# Creating Factor Variables in a Dataset

- Alternatively, you can check the class using `class()` or use `str()` to view either the entire dataset's variable names and types (if you input the dataset name) or just the one variable (if you only input that):

```
> str(iris$Species)
Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 ...
```

# Creating Factor Variables in a Dataset

## What are the Levels of a Factor, and How Can You Change Them?

- The levels of a factor are the particular categories for that variable.
- They are a special attribute of factor objects in R.
- You can view them with the `levels()` function, as shown in the following example:

`levels(iris$Species)`

- It returns the three species of irises indicated in the Species variable column, as follows:

```
> levels(iris$Species)
[1] "setosa"      "versicolor"   "virginica"
>
```

# Using ifelse() Statements

- The following code will change the representation of the three species to numbers:

```
iris$Species2 <- ifelse(iris$Species == "setosa", 1,  
ifelse(iris$Species == "versicolor", 2, 3))
```



# Using ifelse() Statements

- We can verify if it has worked by running the table() function as follows (more on this function in the next section!):

```
table(iris$Species)
```

- Thus, we will get the following output:

	Setosa	versicolor	verginica
	50	50	50



# Using ifelse() Statements

- We can also execute the following code to verify whether the representation has changed:

```
table(iris$Species2)
```

- Here is the output that we will get:

1

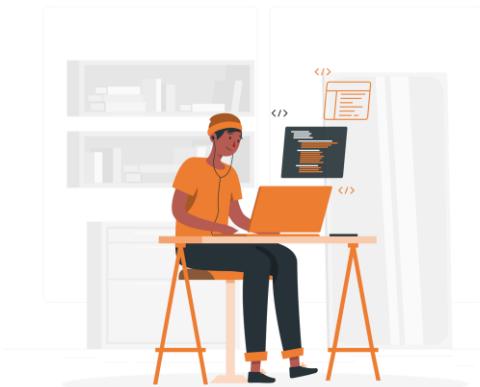
2

3

50

50

50



# Using the recode() Function

- The recode() function, available in the dplyr package, can change the level of the factor by using more readable code, as follows:

```
library(dplyr)  
iris$Species3 <- recode(iris$Species,  
  "setosa" = 1,  
  "versicolor" = 2,  
  "virginica" = 3)
```

# Examining and Changing the Levels of Pre-existing Factor Variables

- Load the dplyr library. Use levels() to see how many levels of band\_instruments\$plays exist, as follows:

```
levels(band_instruments$plays)
```

- Create a new variable, plays2, using ifelse() to change the levels bass and guitar to 1 and 2 using the following code:

```
band_instruments$plays2 <- ifelse(band_instruments$plays  
== "bass", 1,  
ifelse(band_instruments$plays == "guitar", 2,  
band_instruments$plays))
```

- Use levels() to see how many levels of midwest\$state exist as follows:

```
levels(midwest$state)
```

- Load the dplyr library. Create a new variable, state2, by using recode() to change the levels of the state variable to the states' full names:

```
library(dplyr)
```

```
midwest$state2 <- recode(midwest$state,  
  "IL" = "Illinois",  
  "IN" = "Indiana",  
  "MI" = "Michigan",  
  "OH" = "Ohio",  
  "WI" = "Wisconsin")
```



# Examining and Changing the Levels of Pre-existing Factor Variables

**Output:** The following is the output of the code mentioned in **Step 1**:

```
> levels(band_instruments$plays)
[1] "bass"    "guitar"
```

The following is the output of the code mentioned in **Step 3**:

```
> levels(midwest$state)
[1] "IL" "IN" "MI" "OH" "WI"
```



# Examining and Changing the Levels of Pre-existing Factor Variables

- We'll need to set this variable as a factor. When we do so, the code will be as follows:

```
speed <- rep(c("low", "medium", "high"), times = 11)  
speed <- speed[-1]  
mtcars$speed <- factor(speed, levels = c("low",  
"medium", "high"), ordered = TRUE)
```

- Now, when we view the class with the class() function, we see that it is now as follows:  
[1] "ordered" "factor"



# Creating an Ordered Factor Variable

- Create a vector called gas\_price using the following code:

```
gas_price <- rep(c("low", "medium", "high"), times = 146)  
gas_price <- gas_price[-1]
```

It will indicate if gas prices in that area are low, medium, or high on average.

- Add the gas\_price variable to the midwest dataset as follows:

```
midwest$gas_price <- factor(gas_price,  
                           levels = c("low", "medium", "high"),  
                           ordered = TRUE)
```

# Creating an Ordered Factor Variable

- Verify that the variable has been added to the dataset successfully using table() as follows:

```
table(midwest$gas_price)
```



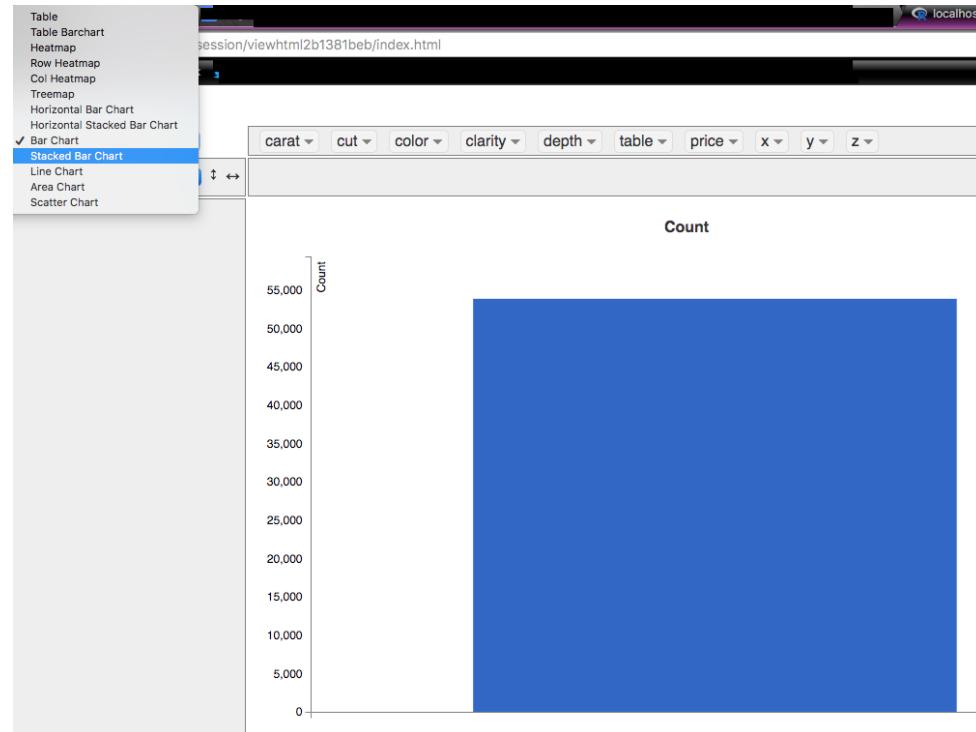
# Complete Activity: Creating and Manipulating Factor Variables



# Summarizing Data

- A huge component of data management and cleaning is summarizing your data.
- It's hard to know what's really inside data just by looking at it.
- If you're a frequent user of Microsoft Excel, you might be familiar with creating pivot tables to summarize data and get a feel for what's inside your dataset.

# Data Summarization Tables



# Data Summarization Tables

- If you install it along with the htmlwidgets package, you can create pivot tables with multiple options and look at stats.
- It's interesting, but you'll also need to know how to manipulate data yourself.
- The rest of this section will focus on building the various kinds of summaries using tools other than rpivotTable.

# Tables in R

Indicator Name	2011	2012	2013	2014	2015	2016	Average	Improvement
Prevalence of Obesity	19.1	23.6	23.3	20.5	24.0	23.2	22.28	-21.47
Prevalence of Tobacco Use	17.4	15.0	15.3	12.2	16.6	16.7	15.53	4.02
Prevalence of Cardiovascular Disease	5.0	4.9	1.5	4.4	4.9	6.2	4.48	-24.00
Prevalence of Diabetes	8.0	7.2	9.3	7.2	7.5	10.4	8.27	-30.00

- Tables in R are very helpful when you want to create a grid of counts of one or two categorical variables.
- They can be saved as an object for export or in combination with other summary tables.
- We've used the iris dataset numerous times by now, and have observed that there are 50 of each species of flower in the data.
- To create a table to verify it, use the table() function.

# Creating Different Tables Using the table() Function

- Load the iris dataset and create a one-way table of the Species variable using the following code:  
`table(iris$Species)`
- Load the diamonds dataset and create a two-way table of the cut and color variables using the following code:  
`table(diamonds$cut, diamonds$color)`



# Creating Different Tables Using the table() Function

- Create a three-way table of the cut, color, and clarity variables from the diamonds dataset as follows:

```
table(diamonds$cut, diamonds$color, diamonds$clarity)
```

- Load the mtcars dataset if it is not already loaded in your environment.
- Create a table of the mpg variable as follows:

```
table(mtcars$mpg)
```

**Output:** The following is the output we get as we execute the code mentioned in **Step 1:**

Setosa	versicolor	virginica
50	50	50

The following is the output we get as we execute the code mentioned in **Step 2:**

	D	E	F	G	H	I	J
Fair	163	224	312	314	303	175	119
Good	662	933	909	871	702	522	307
Very Good	1513	2400	2164	2299	1824	1204	678
Premium	1603	2337	2331	2924	2360	1428	808
Ideal	2834	3903	3826	4884	3115	2093	896



# Creating Different Tables Using the table() Function

, , = I1

	D	E	F	G	H	I	J
Fair	4	9	35	53	52	34	23
Good	8	23	19	19	14	9	4
Very Good	5	22	13	16	12	8	8
Premium	12	30	34	46	46	24	13
Ideal	13	18	42	16	38	17	2

, , = SI2

	D	E	F	G	H	I	J
Fair	56	78	89	80	91	45	27
Good	223	202	201	163	158	81	53
Very Good	314	445	343	327	343	200	128
Premium	421	519	523	492	521	312	161
Ideal	356	469	453	486	450	274	110

# Creating Different Tables Using the table() Function

- The following is the output we get as we execute the code mentioned in Step 4:

10.4	13.3	14.3	14.7	15	15.2	15.5	15.8	16.4	17.3	17.8	18.1	18.7	19.2	19.7	21	
2	1	1	1	1	2	1	1	1	1	1	1	1	1	2	1	2
21.4	21.5	22.8	24.4	26	27.3	30.4	32.4	33.9								
2	1	2	1	1	1	2	1	1								



# Using dplyr Methods to Create Data Summary Tables

- Load the diamonds dataset using the following code:

```
data(diamonds)
```

- Group the data by cut, color, and clarity, and find the number of observations at each combination of the three variables, as follows:

```
diamonds %>% group_by(cut, color, clarity) %>%  
summarise(n())
```

# Using dplyr Methods to Create Data Summary Tables

- Find the mean and median price of diamonds by using the dplyr functions group\_by() and summarise() as follows:

```
diamonds %>% group_by(cut) %>%  
summarise(mean = mean(price), median =  
median(price))
```



# Using dplyr Methods to Create Data Summary Tables

- We can also filter out data we're not interested in quickly using dplyr methods.
- Say we don't want any diamonds with color D or J.
- We can find the mean price by cutting all of the diamonds left in the dataset after removing them:

```
diamonds %>% filter(color != "D" & color != "J")  
%>% group_by(cut) %>% summarise(mean =  
mean(price))
```

# Using dplyr Methods to Create Data Summary Tables

```
# A tibble: 276 x 4
# Groups: cut, color [?]
  cut   color clarity `n()`
  <ord> <ord> <ord>    <int>
1 Fair  D     I1      4
2 Fair  D     SI2     56
3 Fair  D     SI1     58
4 Fair  D     VS2     25
5 Fair  D     VS1      5
6 Fair  D     VVS2     9
7 Fair  D     VVS1     3
8 Fair  D     IF       3
9 Fair  E     I1      9
10 Fair E     SI2     78
# ... with 266 more rows
```

The mean and median price of diamonds is as follows:

```
# A tibble: 5 x 3
  cut      mean   median
  <ord>    <dbl>   <dbl>
1 Fair    4358.758 3282.0
2 Good   3928.864 3050.5
3 Very Good 3981.760 2648.0
4 Premium 4584.258 3185.0
5 Ideal   3457.542 1810.0
```

The mean price by **cut** of all of the diamonds left in the dataset after removing them is as follows:

```
# A tibble: 5 x 2
  cut      mean
  <ord>    <dbl>
1 Fair    4311.788
2 Good   3966.567
3 Very Good 3983.078
4 Premium 4597.057
5 Ideal   3515.849
```



# Complete Activity: Creating Data Summarization Tables



# Summarizing Data with the Apply Family

- The apply family of functions is an incredibly powerful set of R functions that you should learn early on in your R programming journey.
- It'll be very helpful to be skilled in summarizing across many variables at once.
- This is where the apply family of functions comes in.

# Summarizing Data with the Apply Family

- Let's look at a few examples of how to use the apply family to summarize data.
- One example of the use of the apply() function would be the following:

```
numbers <- rbind(c(1:5), c(2:6)) apply(numbers, 2,  
mean)
```

- The output that we get is the small matrix called numbers, which is represented as follows:

```
> numbers  
 [,1] [,2] [,3] [,4] [,5]  
[1,] 1 2 3 4 5  
[2,] 2 3 4 5 6
```



# Summarizing Data with the Apply Family

- We used `apply()` here to calculate the mean of every column of the numbers matrix:  
`apply(numbers, 2, mean)`

- Thus, we get the output as follows:

```
[1] 1.5 2.5 3.5 4.5 5.5
```

# Summarizing Data with the Apply Family

- You can also use multiple functions with `apply()`.  
Here's an example of that:

```
apply(numbers, 2, function(x) c(median(x), var(x)))
```

- The output is as follows:

```
[,1] [,2] [,3] [,4] [,5]  
[1] 1.5 2.5 3.5 4.5 5.5  
[2] 0.5 0.5 0.5 0.5 0.5
```



# Using apply() Function to Create Numeric Data Summaries

- Load the iris dataset using the following code:

```
data("iris")
```

- Find the mean of all of the columns of the iris dataset except the fifth column (the Species column, which isn't numeric) with the following code:

```
apply(iris[,-c(5)], 2, FUN = mean)
```



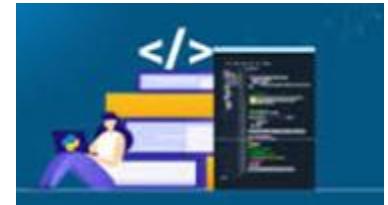
# Using apply() Function to Create Numeric Data Summaries

- Find the mean and variance of all of the columns of iris except the fifth column as follows:

```
apply(iris[,-c(5)], 2, function(x) c(mean(x), var(x)))
```

- Find the mean of all the rows of iris as follows:

```
apply(iris[,-c(5)], 1, FUN = mean)
```



# Using apply() Function to Create Numeric Data Summaries

**Output:** The following is the output we get as we execute the code mentioned in the second step:

```
> apply(iris[,-c(5)], 2, FUN = mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
      5.843333     3.057333     3.758000     1.199333
```

The following is the output we get as we execute the code mentioned in the third step:

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width
[1,]      5.843333     3.057333     3.758000     1.199333
[2,]      0.6856935    0.1899794    3.116278     0.5810063
```



# Using apply() Function to Create Numeric Data Summaries

```
[1] 2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225 2.400 2.700 2.500  
[13] 2.325 2.125 2.800 3.000 2.750 2.575 2.875 2.675 2.675 2.675 2.350 2.650  
[25] 2.575 2.450 2.600 2.600 2.550 2.425 2.425 2.675 2.725 2.825 2.425 2.400  
[37] 2.625 2.500 2.225 2.550 2.525 2.100 2.275 2.675 2.800 2.375 2.675 2.350  
[49] 2.675 2.475 4.075 3.900 4.100 3.275 3.850 3.575 3.975 2.900 3.850 3.300  
[61] 2.875 3.650 3.300 3.775 3.350 3.900 3.650 3.400 3.600 3.275 3.925 3.550  
[73] 3.800 3.700 3.725 3.850 3.950 4.100 3.725 3.200 3.200 3.150 3.400 3.850  
[85] 3.600 3.875 4.000 3.575 3.500 3.325 3.425 3.775 3.400 2.900 3.450 3.525  
[97] 3.525 3.675 2.925 3.475 4.525 3.875 4.525 4.150 4.375 4.825 3.400 4.575  
[109] 4.200 4.850 4.200 4.075 4.350 3.800 4.025 4.300 4.200 5.100 4.875 3.675  
[121] 4.525 3.825 4.800 3.925 4.450 4.550 3.900 3.950 4.225 4.400 4.550 5.025  
[133] 4.250 3.925 3.925 4.775 4.425 4.200 3.900 4.375 4.450 4.350 3.875 4.550  
[145] 4.550 4.300 3.925 4.175 4.325 3.950
```

# Complete Activity: Implementing Data Summary

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0., main = "", xlab = "", ylab = "")
if(orientation == "vertical")
  dx2 <- (dx - min(dx)) / max(dx)
  x[1.]
  dy2 <- (dx - min(dy)) / max(dy)
  y[1.]
  seqbelow <- rep(y[1.], length(dx))
  if(Fill == T)
    confshade(dx2, seqbelow, dy2)
```

# Splitting, Combining, Merging, and Joining Datasets

## Splitting and Combining Data and Datasets

- Splitting and unsplitting data is provided for in the base package of R with functions named `split()` and `unsplit()`.
- Combining data is usually done using the base functions `rbind()` and `cbind()`, which combine by row and column, respectively.
- Let's look at how to split, unsplit, and combine data in R.

# Splitting Datasets into Lists and Then Back Again

- Load the iris dataset if it is not currently loaded using the following code:

`data(iris)`

- Split the iris dataset by species & This creates three lists of dataframes, each of which will only contain the information about one species of iris represented in the data.
- Verify that `iris_species` is a list by checking its type and check the class of `iris_species[[1]]` & This can be done with the help of the following code:

```
iris_species <- split(iris, iris$Species)  
typeof(iris_species)  
class(iris_species[[1]])
```

# Splitting Datasets into Lists and Then Back Again

- Print the head of the second dataframe, which contains all the versicolor iris data using the following code:

```
head(iris_species[[2]])
```

- Assign each dataframe into its own separate data object.
- Name the dataframes after the species of iris contained inside, as follows:

```
iris_setosa <- iris_species[[1]]
```

```
iris_versicolor <- iris_species[[2]]
```

```
iris_virginica <- iris_species[[3]]
```



# Splitting Datasets into Lists and Then Back Again

- Use `unsplit()` to recombine `iris_species` into `iris_back`, which should be identical to the original `iris` dataset.
- Verify that they are identical using `all_equal()` from `dplyr`, which compares every aspect of the two dataframes.
- It can be done using the following code:

```
iris_back <- unsplit(iris_species, iris$Species)  
library(dplyr)  
all_equal(iris, iris_back)
```



# Splitting Datasets into Lists and Then Back Again

- Since dplyr is now loaded, recreate the three different iris datasets using filter() on iris to retain only one species of iris at a time.
- This method involves less code than using split() to create a list of dataframes by allowing you to create each dataframe directly:

```
iris_setosa_2 <- iris %>% filter(Species == "setosa")
iris_versicolor_2 <- iris %>% filter(Species == "versicolor")
iris_virginica_2 <- iris %>% filter(Species == "virginica")
```

# Splitting Datasets into Lists and Then Back Again

- Rejoin the three new iris dataframes by using rbind.data.frame(), and verify that it's the same as iris by using all\_equal():

```
iris_back_2 <- rbind.data.frame(iris_setosa_2,  
iris_versicolor_2, iris_virginica_2)  
all_equal(iris, iris_back_2)
```



# Splitting Datasets into Lists and Then Back Again

- Output: The following is the output we get as we execute the code from the second step:

```
[1] "list"
```

```
[1] "dat.frame"
```

# Splitting Datasets into Lists and Then Back Again

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
51	7.0	3.2	4.7	1.4	versicolor
52	6.4	3.2	4.5	1.5	versicolor
53	6.9	3.1	4.9	1.5	versicolor
54	5.5	2.3	4.0	1.3	versicolor
55	6.5	2.8	4.6	1.5	versicolor
56	5.7	2.8	4.5	1.3	versicolor

# Splitting Datasets into Lists and Then Back Again

[1]

TRUE

# Splitting Datasets into Lists and Then Back Again

[1]

TRUE

# Combining Data with rbind()

- Install and load the ggplot2 package, as it contains the midwest dataset:

```
install.packages("ggplot2") library(ggplot2)
```

- Load the midwest data and examine its contents with str():

```
data("midwest") str(midwest)
```



# Combining Data with rbind()

- We'll first need to split the data in order to combine it.
- Let's split it evenly, in half, to create midwest\_1 and midwest\_2.
- We can calculate directly in our subsetting method to get half of the number of rows of midwest in each dataset:

```
midwest1 <- midwest[1:round(nrow(midwest)/2),]  
midwest2 <-  
midwest[(round(nrow(midwest)/2)+1):nrow(midwest),]
```

# Combining Data with rbind()

- Recombine midwest into midwest\_back using rbind() to combine by rows (because we split in half by rows!):

```
midwest_back <- rbind(midwest1, midwest2)
```

- Check to see if midwest\_back is the same as midwest using all\_equal(), like we did previously:

```
all_equal(midwest, midwest_back)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':      437 obs. of  28 variables:  
 $ PID           : int  561 562 563 564 565 566 567 568 569 570 ...  
 $ county        : chr "ADAMS" "ALEXANDER" "BOND" "BOONE" ...  
 $ state         : chr "IL" "IL" "IL" "IL" ...  
 $ area          : num  0.052 0.014 0.022 0.017 0.018 0.05 0.017 0.027 0.024 0.058 ...  
 $ poptotal      : int  66090 10626 14991 30806 5836 35688 5322 16805 13437 173025 ...  
 $ popdensity    : num  1271 759 681 1812 324 ...  
 $ popwhite      : int  63917 7054 14477 29344 5264 35157 5298 16519 13384 146506 ...  
 $ popblack      : int  1702 3496 429 127 547 50 1 111 16 16559 ...  
 $ popamerindian: int  98 19 35 46 14 65 8 30 8 331 ...  
 $ popasian      : int  249 48 16 150 5 195 15 61 23 8033 ...  
 $ popother      : int  124 9 34 1139 6 221 0 84 6 1596 ...  
 $ percwhite     : num  96.7 66.4 96.6 95.3 90.2 ...  
 $ percblack     : num  2.575 32.9 2.862 0.412 9.373 ...  
 $ percamerindan: num  0.148 0.179 0.233 0.149 0.24 ...  
 $ percasiain    : num  0.3768 0.4517 0.1067 0.4869 0.0857 ...  
 $ percother     : num  0.1876 0.0847 0.2268 3.6973 0.1028 ...  
 $ popadults     : int  43298 6724 9669 19272 3979 23444 3583 11323 8825 95971 ...  
 $ perchsd       : num  75.1 59.7 69.3 75.5 68.9 ...  
 $ percollege    : num  19.6 11.2 17 17.3 14.5 ...  
 $ percprof      : num  4.36 2.87 4.49 4.2 3.37 ...  
 $ poppovertyknown: int  63628 10529 14235 30337 4815 35107 5241 16455 13081 154934 ...  
 $ percpovertyknown: num  96.3 99.1 95 98.5 82.5 ...  
 $ percbelowpoverty: num  13.15 32.24 12.07 7.21 13.52 ...  
 $ percchildbelowpovert: num  18 45.8 14 11.2 13 ...  
 $ percadultpoverty: num  11.01 27.39 10.85 5.54 11.14 ...  
 $ percelderlypoverty: num  12.44 25.23 12.7 6.22 19.2 ...  
 $ inmetro        : int  0 0 0 1 0 0 0 0 1 ...  
 $ category       : chr "AAR" "LHR" "AAR" "ALU" ...
```



# Combining Data with rbind()

```
[1] TRUE
```

# Combining Matrices of Objects into Dataframes

- Create one, two, three, and four, which are all vectors of sequential numbers:

```
one <- 1:15
```

```
two <- 16:30
```

```
three <- 31:45
```

```
four <- 46:60
```

- Create all1 and all2 from one, two, three, and four. all1 should be combined by rows, while all2 should be combined by columns:

```
all1 <- rbind(one, two, three, four)
```

```
all2 <- cbind(one, two, three, four)
```



# Combining Matrices of Objects into Dataframes

- Check the class of all1:

```
class(all1)
```

- Recombine one, two, three, and four into data.frames and look at the class of all3:

```
all3 <- rbind.data.frame(one, two, three, four)
```

```
all4 <- cbind.data.frame(one, two, three, four)
```

```
class(all3)
```

# Combining Matrices of Objects into Dataframes

- Output: The following is the output we get as we execute the code class(all1):

```
[1] "Matrix"
```

- The following is the output we get as we execute the code mentioned in the last Step 4:

```
[1] "data.frame"
```

# Using stringr Package to Manipulate a Vector of Names

- Install and then load the stringr package:

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

- Create the names vector, a list of various names, and check its length to see how many names it contains:

```
names <- c("Danelle Lewison", "Reyna Wieczorek", "Jaques Sola", "Marcus Huling", "Elvis Driver", "Chandra Picone", "Alejandro Caffey", "Shawnna Lomato", "Masako Hice", "Wally Ota", "Phillip Batten", "Denae Rizzuto", "Joseph Merlos", "Maurice Debelak", "Carina Gunning", "Tama Moody")  
length(names)
```

# Using stringr Package to Manipulate a Vector of Names

- Use `str_split()` to separate each name into first name and surname and save it as an object called `names_split`. `str_split()` takes two arguments: the vector (or character string) you plan to split, and a pattern to split on:

```
names_split <- str_split(names, pattern = " ")
```



# Using stringr Package to Manipulate a Vector of Names

- Examine the first split name in names\_split. Then, look at the first name.
- Remember to use list indexing, as names\_split is a list of the split first names and surnames:

```
names_split[[1]]  
names_split[[1]][1]
```



# Using stringr Package to Manipulate a Vector of Names

- Split create names\_split\_a, which splits names at any a in each name.
- You only have to change one of the inputs to str\_split() that you used previously:

```
names_split_a <- str_split(names, pattern = "a")
```

- Examine the first split name and the second half of the first split name in names\_split\_a once more.
- How has it been split differently?

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

# Using stringr Package to Manipulate a Vector of Names

- Now, examine the fifth split name from names\_split\_a.
- What happened with this name that has no a in it?  
**names\_split\_a[[5]]**
- Output: The following is the output we get upon executing the code mentioned in Step 2:

```
[1] 16
```



The following is the output we get upon executing the code mentioned in **Step 4**:

```
[1] "Danelle" "lewison"  
[1] "Danelle"
```

The following is the output we get upon executing the code mentioned in **Step 6**:

```
[1] "Elvis Driver"
```

# Combining Strings Using Base R Methods

- Create variables a, b, and c, which contain character strings:

```
a <- "R" b <- "is" c <- "fun"
```

- Use paste() to combine a, b, and c with an exclamation mark:

```
paste(a, b, c, "!")
```

- Use paste0() to do the same, but without spaces between a, b, c, and the exclamation mark:

```
paste0(a, b, c, "!")
```

# Combining Strings Using Base R Methods

- Use `paste()` to create the string "R is fun x 10" with the objects you've created:

```
paste(a, b, c, "x", 10)
```

- Output: The following is the output we get upon executing the code mentioned in Step 2:

```
[1] "R is fun !"
```



# Combining Strings Using Base R Methods

- The following is the output we get upon executing the code mentioned in Step 3:

```
[1] "Risfun!"
```

- The following is the output we get upon executing the code mentioned in Step 4:

```
[1] "R is fun x 10"
```

# Complete Activity: Demonstrating Splitting and Combining Data



# Merging and Joining Data

Type of Join	Rows	Columns	merge() argument
Inner	All from <b>x</b> where there are matching values of ID in <b>y</b> . If multiple matches of ID exist, there will be multiple rows of that ID.	All from <b>x</b> and <b>y</b>	Default–no additional argument needed
Semi	All from <b>x</b> where there are matching values of ID in <b>y</b> . Never returns multiple matches.	Only those from <b>x</b>	N/A
Left	All from <b>x</b> . If multiple matches of ID exist, there will be multiple rows of that ID.	All from <b>x</b> and <b>y</b>	<code>all.x = TRUE</code>
Right	All from <b>y</b> . If multiple matches of ID exist, there will be multiple rows of that ID.	All from <b>x</b> and <b>y</b>	<code>all.y = TRUE</code>
Full	All from <b>x</b> and <b>y</b> . If no matching values on ID from <b>x</b> and <b>y</b> , there will be an NA for the missing.	All from <b>x</b> and <b>y</b>	<code>all = TRUE</code>

# Demonstrating Merges and Joins in R

- Install and load the `readr` package, which contains functions that read in data much faster than the `baseR` data read functions:

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

- Download the `students` and `students2` datasets from the GitHub repository:

```
students <-  
  read_csv("https://github.com/fenago/students.csv")  
students2 <-  
  read_csv("https://github.com/fenago/students2.csv")
```

# Demonstrating Merges and Joins in R

- Examine both datasets using `str()`. Verify that they each has an ID variable, and take note that `students` has information about 20 students (20 observations)
- While `students2` has information on five additional students (25 observations):

`str(students) str(students2)`



# Demonstrating Merges and Joins in R

- Create students\_combined by merging the two datasets by ID.
- Check the dimensions of students combined to see how many students' information is retained on this inner join.
- There should only be 20 matches on ID between the two datasets on this default inner join:

```
students_combined <- merge(students, students2, by =  
"ID") dim(students_combined)
```

# Demonstrating Merges and Joins in R

- Create `students_combined2`, this time performing a right join using `merge()`, which should retain all of the possible students' information.
- Check the dimensions to see how much of students' information is in the combined dataset.
- Does it match up with your expectations?

```
students_combined2 <- merge(students, students2, by = "ID", all.y = TRUE) dim(students_combined2)
```

~ / R\_directory / packt\_introDSR / BeginningDSwRCodeFiles - master - RStudio Source Editor

students\_combined

Filter

ID Height\_inches Weight\_lbs EyeColor HairColor USMensShoeSize Gender Grade Sport

ID	Height_inches	Weight_lbs	EyeColor	HairColor	USMensShoeSize	Gender	Grade	Sport
1	1	65	120	Blue	Brown	9	F	9 Basketball
2	2	55	135	Brown	Blond	5	F	9 Track
3	3	60	166	Hazel	Black	6	M	12 Tennis
4	4	61	154	Brown	Brown	7	M	11 None
5	5	62	189	Green	Blond	8	M	10 Tennis
6	6	66	200	Green	Red	9	F	12 Tennis
7	7	69	250	Blue	Red	10	F	12 None
8	8	54	122	Blue	Brown	5	M	9 Basketball
9	9	57	101	Blue	Brown	6	F	12 Basketball
10	10	58	178	Brown	Black	4	F	10 Track
11	11	59	199	Hazel	Blond	8	F	10 Track
12	12	59	260	Green	Black	9	F	9 Track
13	13	60	145	Blue	Brown	10	M	11 None
14	14	60	158	Brown	Blond	11	M	10 Basketball
15	15	57	197	Brown	Black	12	M	11 None
16	16	66	126	Blue	Red	6	F	10 Track
17	17	67	278	Green	Brown	5	F	12 Track
18	18	68	225	Hazel	Black	9	F	10 Track
19	19	69	103	Blue	Blond	7	M	11 Basketball
20	20	70	111	Blue	Red	5	M	10 None
21	21	NA	NA	NA	NA	NA	M	9 Tennis
22	22	NA	NA	NA	NA	NA	M	11 Basketball
23	23	NA	NA	NA	NA	NA	M	10 Basketball
24	24	NA	NA	NA	NA	NA	M	11 None
25	25	NA	NA	NA	NA	NA	M	11 Basketball

Showing 1 to 25 of 25 entries

# Demonstrating Merges and Joins in R

- Install and load the dplyr package, if you have not done either of these already:

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

- Create students\_right\_join, performing another right join, but this time using the dplyr join methods.
- Check the dimensions to verify the number of students' information in the joined dataset:

```
students_right_join <- right_join(students, students2, by =  
"ID")
```

```
dim(students_right_join)
```

# Demonstrating Merges and Joins in R

- Create students\_anti\_join similarly and check the dimensions. Based on the preceding table, is the output what you expected?

```
students_anti_join <- anti_join(students, students2, by =  
  "ID") dim(students_anti_join)
```

- If the by variables are named the same things, you can actually do both merges and joins without specifying a by variable:

```
students_merge_noby <- merge(students, students2)  
students_join_noby <- right_join(students, students2)
```

# Demonstrating Merges and Joins in R

- Rename the ID variable on students to be called StudentID.
- Now, merge and join the data using the slightly different by variable names to see how powerful merge and join functions truly are:

```
colnames(students)[6] <- "StudentID"  
students_merge_diff <- merge(students, students2, by.x  
= "StudentID", by.y = "ID")  
students_join_diff <- right_join(students, students2, by =  
c("StudentID" = "ID"))
```

# Demonstrating Merges and Joins in R

- Output: The following is the students dataset as an output:

```
Classes 'tbl_df', 'tbl' and 'data.frame': 20 obs. of 6  
variables: $ Height_inches : int 65 55 60 61 62 66 69 54  
57 58 ... $ Weight_lbs : int 120 135 166 154 189 200  
250 122 101 178 ... $ EyeColor : chr "Blue" "Brown"  
"Hazel" "Brown" ... $ HairColor : chr "Brown" "Blond"  
"Black" "Brown" ... $ USMensShoeSize: int 9 5 6 7 8 9  
10 5 6 4 ... $ ID : int 1 2 3 4 5 6 7 8 9 10 ...
```

# Demonstrating Merges and Joins in R

- The following is the students2 dataset as an output:

```
'data.frame': 25 obs. of 4 variables: $ ID : int 1 2 3 4 5 6  
7 8 9 10 ... $ Gender: Factor w/ 2 levels "F","M": 1 1 1 1  
1 1 1 1 1 2 ... $ Grade : num 10 10 9 10 12 9 12 12 11  
10 ... $ Sport : Factor w/ 4 levels "Basketball","None",...  
4 3 3 1 1 4 4 3 4 3 ... 4. [1] 20 9 5. [1] 25 9 7. [1] 25 9 8.  
[1] 0 6 9b. Joining, by = "ID"
```



# Complete Activity: Merging and Joining Data



# Summary

- Data management is a crucial skill needed for working with data in R, and we have covered many of the basics in this lesson.
- One thing to keep in mind is that there is no prescribed order in which to conduct data management, cleaning, and data visualization.

