Workshop: Basic R and More

Randy Law
Analytics & Insights Matter
Highlands Ranch, CO

Philip Turk, PhD[1]

Western Data Analytics, LLC; Denver, CO

08/16/2019

[1] <pturk@westerndataanalytics.com>

Introduction

- History of R:
  - Was created by Ross Ihaka and Robert Gentleman in 1992 as an implementation of the S programming language
  - Language is based on C, Fortran, and R itself
  - Initial Windows version released in 1995
  - Comprehensive R Archive Network (CRAN) started in 1997
  - Considered stable enough for production use in 2000
  - Mac OS X version released in 2001
  - Now over 14,000 available contributed packages
- For any credible poll on the Internet regarding the popularity of statistics and data science software, R will be at or near the top.
- I have been using R ever since it was first released. My company uses it the vast majority of the time. Then it's a tossup between SAS and Python, followed by a smattering of a few other things.

Introduction

- This is an intense "get started" workshop on R and more. I assume you know nothing about it.
- R is an open-source (free!) program that is commonly used in statistics and data science. Most of the leading statistical research is first available on R.
    - <https://www.r-project.org>
- R is a command-line/script-based language, so there is no point-and-click interface. The initial learning curve will be steep, but there are several advantages. Two important ones are:
    - You have to understand what you are doing and why you are doing it.
    - You have a record of what steps you performed in your data analysis.

Introduction

- Finding help:
  - Will show you how to do this in R (e.g., `> ?mean`)
  - Google
  - <https://stackoverflow.com>
- You should run R through the program *RStudio* (<https://www.rstudio.com>). This is a free Integrated Development Environment (IDE) that works on Macs, Windows and Linux. It has many advantages we will demonstrate.
- Many people find RStudio Cheat Sheets to be very useful when they are first starting out:
  - <https://www.rstudio.com/resources/cheatsheets/>

Introduction

- 👁 👁 (means we will be switching over to my laptop)
- When you open up RStudio, you will see four windows. My default *starting* configuration starting from the upper-left window moving clockwise is as follows:
  - Source: This is where you write your R code.
  - Environment: You can see what you have created or view your previous commands. This window can also act as an interface to GitHub and Azure Repos.
  - Files: You can see file directories, view plots, see your packages, and access R help.
  - Console: This is where your code from the Source is evaluated by R or you can also perform quick calculations and operations.
- We will begin our workshop coding in the Console window. Later on, we will see more efficient ways to work.

Introduction

- When you start learning how to write R code, you do not type command prompts > nor line spans +. My way of programming in R is to use the symbol # to 'silence' code in the testing phase and to use ## to leave comments in the script. In this workshop, you do not need to type my silenced code and comments to reproduce the results.
- Online courses from Udemy, Coursera, DataCamp, etc., are good, but they will never replace nor be as effective as lots of practice and the subsequent **normal** discomfort of learning, and being able to ask questions face-to-face in real-time.
- So let's get started! This workshop is designed for you to be an active participant. You'll want to follow along and run R on your laptop. It helps to sit near someone who is already familiar with R. For this workshop, I will break for a few minutes at the top of every hour to quickly walk around the room to answer any questions. For those of us that are visiting online, you may email me your questions at <pturk@westerndataanalytics.com> or text them to me at 304-376-5377.

R as a Calculator

- In this workshop, the command prompt and code are multi-colored for clarity and decoration. Your command prompt and code will be black.
- Go to the Console window, type each of the following operations at the command prompt >, hitting Return after each time 👁 👁

```
> 2 + 3 ## Add comments; be kind to your future self!
```

```
[1] 5
```

```
> 2-3 ## R is insensitive to spacing
```

```
[1] -1
```

```
> 2*3
```

```
[1] 6
```

```
> 2/3
```

```
[1] 0.6666667
```

Introduction to Functions

- Honorable mentions: trig (e.g., `sin()`), exponential (`exp()`), log (`log()`), absolute value (`abs()`), and many other functions
- When you call a function, there will be some arguments that are mandatory, some that are optional, and the arguments are separated by a comma.
  - Type `?log` in your console 👁 👁

```
> log(100) ## R will default to natural base e
```

```
[1] 4.60517
```

```
> log(100, base = exp(1)) ## Redundant and unnecessary
```

```
[1] 4.60517
```

```
> log(100, base = 10) ## Override option default to base 10
```

```
[1] 2
```

Introduction to Functions

- Arguments can be specified via the order in which they specified or by naming the arguments. Be careful when you are first starting out learning R!

```
> log(x = 100, base = 10) ## x = 100 <-> 100
```

```
[1] 2
```

```
> log(base = 10, x = 100) ## Weird but it works
```

```
[1] 2
```

```
> log(100, 10) ## R decides x is first, base second
```

```
[1] 2
```

```
> log(10, 100) ## Bad news?
```

```
[1] 0.5
```

Assigment

- Simply typing something into R does not mean R remembers it. You must save it as an object.
- We need to be able to save, or *assign*, a value or values to a variable to be able to use it later. R does this by using an arrow <- or an equal sign = (I prefer the arrow).
- 👁 👁

```
> a <- 2*pi
> a  ## Hit return to see it
```

```
[1] 6.283185
```

Assignment

Here are some rules for naming variables in R:

- Variable names may not include spaces, and are case sensitive. *This last one is a very common mistake*.
- The variable name can be a combination of letters, numbers, period (.), and underscore (_) (e.g., can't have %).
- It must start with a letter or a period (e.g., can't start with a number, can't start with underscore).
- If it starts with a period, then it can't be followed by a number.
- Reserved words in R can't be used (e.g., `TRUE`).

Assignment

```
> b <- 5  ## Look at the Environment tab in RStudio
> b
```

```
[1] 5
```

```
> c <- a*b ## Create a third variable from first two
> c
```

```
[1] 31.41593
```

Assignment

As you work with R you'll often want to re-execute a command which you previously entered. The RStudio console supports the ability to recall previous commands using the arrow keys:

- 👁 👁
- Up will recall previous command(s)
- Down is the reverse of Up

Vectors

- R operates on vectors where a vector is a collection of elements, usually numbers. To do this, we use the `c()` function which ‘combines’ elements into a vector.

```
> c(1, 2, 3, 4)
```

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

- There are other ways to define vectors.

```
> rep(2, 5) ## Repeat 2 five times
```

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

```
> rep(c("A", "B"), 3) ## Repeat A B three times
```

```
[1] "A" "B" "A" "B" "A" "B"
```

- Note that single quotes `'` would work as well when defining character values (e.g., `A`).

Vectors

- We can also define a *sequence* of numbers using the `seq(from, to, by, length.out)` function which expects the user to supply 3 out of 4 possible arguments. '`From`' is the starting point of the sequence, '`to`' is the end point, '`by`' is the difference between any two successive elements, and '`length.out`', or just '`length`', is the total number of elements in the vector.

```
> seq(1, 4) ## 'by' default is 1
```

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

```
> ## 1:4 by itself will also work
> seq(1, 4, by = 0.5)
```

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

```
> seq(6, 12, length.out = 5) ## Doesn't have to be integers
```

```
[1]  6.0  7.5  9.0 10.5 12.0
```

Vectors

- If we have two vectors and we wish to combine them, we can again use the `c()` function.

```
> vec1 <- rep(2, 5)
> vec2 <- seq(6, 10, length.out = 3)
> vec3 <- c(vec1, vec2)
> vec3
```

```
[1]  2  2  2  2  2  6  8 10
```

Accessing Vector Elements

- To pull out a certain elements or several elements from a vector, we use [] notation.

```
> vec3[6] ## Sixth element in vec3
```

```
[1] 6
```

```
> vec3[c(5, 7)] ## Elements 5 and 7 in vec3
```

```
[1] 2 8
```

```
> vec3[5:7] ## Elements 5 through 7 in vec3
```

```
[1] 2 6 8
```

```
> vec3[-7] ## All of vec3 except element 7
```

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

Accessing Vector Elements

- Now the [1] notation you've been seeing all along in the output should make sense. Because vectors are often very long and might span multiple lines, R will help you by telling you the index number of the left most value. If we have a very long vector, the second line of values will start with the index of the first value on the second line.
- For example, submit the following and see what happens. Your answers will vary depending on the size of your Console window 👁 👁

```
> letters
```

Scalar Functions Applied to Vectors

- It is very common to want to perform some operation on all the elements of a vector simultaneously. Scalar functions that are defined on single values will almost always apply the function to each element of the vector (“element-wise”) if given a vector.

```
> x <- -5:5
> x
```

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

```
> abs(x) ## Take absolute value of each element
```

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

```
> x^2 ## Square each element
```

```
 [1] 25 16  9  4  1  0  1  4  9 16 25
```

Vector Algebra

- All algebra done with vectors will be done element-wise by default.
    - For matrix and vector multiplication as usually defined by mathematicians, do **not** use this approach.

```
> x <- 1:4
> x
```

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

```
> y <- 5:8
> y
```

```
[1] 5 6 7 8
```

```
> x + y
```

```
[1]  6  8 10 12
```

```
> x*y
```

```
[1]  5 12 21 32
```

Commonly Used Scalar Functions

- R has many common scalar functions already in the base version of R (base R). For example, the sample variance S^2 is easily computed in R's `var()` function. Nevertheless, we can easily write our own functions if we have to. To show how scalars, vectors, and functions of them work together, we will show how to use R to compute the sample variance S^2 by hand. Recall:

$$S^2 = \frac{\sum_{i=1}^{n}(X_i - \overline{X})^2}{n-1}$$

- Suppose we have the sample $\{2, 4, 6, 8, 10\}$. What is the sample variance?

Commonly Used Scalar Functions

```
> x <- c(2, 4, 6, 8, 10)
> xbar <- mean(x) ## Compute the mean
> xbar
```

```
[1] 6
```

```
> x - xbar ## Compute the 'errors'
```

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

```
> (x - xbar)^2 ## Square the errors
```

```
[1] 16  4  0  4 16
```

```
> sum((x - xbar)^2) ## Sum of the squared errors
```

```
[1] 40
```

Commonly Used Scalar Functions

```
> n <- length(x) ## How many data do we have?
> n
```

```
[1] 5
```

```
> sum((x - xbar)^2)/(n - 1)
```

```
[1] 10
```

```
> var(x) ## Same as R's built-in function
```

```
[1] 10
```

Matrices

- We can store numerical data in *matrices*. These will have two dimensions, rows and columns.

```
> W <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
> W
```

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

- The order that R fills the matrix is to fill in the first column, then the second, and then the third. If we wanted to fill the matrix in order of the rows first, then we'd use the optional `byrow = TRUE` argument (default is set to `FALSE`).
- Technically, either `nrow` or `ncol` could be left off in this example.

Matrices

- Accessing a part of a matrix is done in a similar manner as with vectors, using the `[]` notation, but this time we must specify which row and which column using `[row, col]`.

```
> W[1, 2] ## Row 1, column 2
```

```
[1] 3
```

```
> W[1, 1:2] ## Row 1, columns 1 and 2; c(1, 2) also works
```

```
[1] 1 3
```

```
> W[1, ] ## All of row 1
```

```
[1] 1 3 5
```

```
> W[ ,2] ## All of column 2
```

```
[1] 3 4
```

Matrices

- We can also assign names for the columns and rows.

```
> colnames(W) <- c("T1", "T2", "T3") ## Set column names
> rownames(W) <- c("CO", "NC") ## Set row names
> W
```

```
   T1 T2 T3
CO  1  3  5
NC  2  4  6
```

- Note: This is actually an odd and outdated way of setting the attributes of the object `W` because it looks like we are evaluating a function and assigning some value to the function output.

Data Frames

- R stores collections of variables (of different types) in a data structure known as a *data frame*. (Nowadays, you will hear people refer to *tibbles*, which are "optimized" data frames for newer packages.)
- The columns are for different variables, and the rows are for different units (e.g., person, thing, etc.) Hence, looking across a given row, each column represents some information about that unit.
- As a reminder, do not type the line spans + in the following code:

```
> Scores <- data.frame(
+   Name  = c("Susie", "Phil", "Margarida", "Stephen"),
+   Sex = c("F", "M", "F", "M"),
+   Exam.1 = c(90, 75, 92, 85),
+   Exam.2 = c(87, 71, 95, 81)
+ )
```

Data Frames

```
> Scores
```

```
       Name Sex Exam.1 Exam.2
1     Susie   F     90     87
2      Phil   M     75     71
3 Margarida   F     92     95
4   Stephen   M     85     81
```

- Here is where RStudio shines. An attractive option is to be able to show the data in an spreadsheet and even interact with it 👁 👁

Data Frames

- There are two different ways to access the components of a data frame. The first way allows for the previous 'matrix'-type of indexing using [].

```
> Scores[1, 3] ## Row 1, column 3
```

```
[1] 90
```

```
> Scores[1, ] ## All of row 1
```

```
   Name Sex Exam.1 Exam.2
1 Susie   F     90     87
```

```
> Scores[, 3] ## All of column 3
```

```
[1] 90 75 92 85
```

Data Frames

- Because we have created column names, we can access a column by its name as opposed to the column number. The $ sign can be thought of as a pathway: *data frame name$column name within data frame*

```
> Scores$Exam.1
```

```
[1] 90 75 92 85
```

- We can use both the column name followed by `[]`.

```
> Scores$Exam.1[2]
```

```
[1] 75
```

- Notice the automatic completion feature in the Console window

Importing Data

- For small data sets, we want to read data from an external source where the data are set up like a data frame.
- Every time you start up RStudio, it picks an appropriate working directory. This is the directory where it will first look for .R/.Rmd files or data files. By default, when you double click on a .R/.Rmd file to launch RStudio, it will set the working directory to be the directory that the file was in.
- There are three different possibilities where we tell R the data file resides:
    - a web address
    - an absolute path on your computer
    - a path relative to the location of your .R/.Rmd file

Importing Data

- The simplest file format for storage is the *comma separated values* text file, or .csv file. Each of the 'cells' of data are separated by a comma. Ideally, there are column names. Any missing data are designated using a period sign. MS Excel can save a workbook as a .csv file.
- Typically when you open up such a file on a computer with MS Excel installed, Excel will open up the file assuming it is a spreadsheet and put each element in its own cell. However, you can also open the file using a more 'primitive' text editor. My favorite is the free version of Sublime because it works very nicely with Python and Git (https://www.sublimetext.com).
- Let's take a look at a toy data set called *bears.csv* that is completely ready to be imported into R

Importing Data

- To import *bears.csv* from a path relative to the location of this .Rmd file, one approach is the following 👁 👁

```
> my.data <- read.csv("bears.csv")
```

- If *bears.csv* was instead located in some far-flung location of my computer, then I would extend the path accordingly. Notice the use of forward slashes.

```
> my.data <- read.csv("/Users/pturk/.../bears.csv")
```

- Caution: Mac OS uses / in paths; Windows uses \\ in paths

Importing Data

- To read a .csv text file from a web address, we substitute the file path with the web URL. For example, try a.) *carefully typing*, or b.) *copying and pasting* “https://raw.githubusercontent.com/philturk/R_Wshop/master/bears.csv” as the argument for the previous `read.csv()` function and hit Return. Fingers crossed!

Packages

- One of the greatest strengths about R is that many people have developed add-on *packages* to do some additional functions above and beyond base R.
- RStudio makes it easy to obtain packages. Go the Packages tab in the lower-right window. Click Install. In the Packages field, start typing the name of your package (e.g., `blandr`). Notice the automatic completion again. When you are done, click Install. Let's do this 👁 👁
- Once a package is installed on your computer, it is available, but not "loaded" into your current R session by default. To improve overall computer performance, only a few packages are loaded by default. You need to explicitly load add-on packages each session using the `library()` function:

```
> library(blandr)
```

Summarizing Data Using `dplyr`

- I use the `tidyverse` package so much that I load it right away in almost every R session (submit `library(tidyverse)`). Actually, when you load “the `tidyverse`”, the suite of core packages (the packages used in most data engineering and analyses) get installed and loaded en masse.
  - <https://www.tidyverse.org>
- It is very important to be able to take a data set, wrangle it, produce summary statistics, etc. Although there are ‘classic’ functions in base R that have some capability of doing these, nowadays I rely on the package `dplyr` (a member of the `tidyverse`). This package also allows me to chain together many common actions to accomplish a particular task in a **fast**, convenient, and consistent way.
- Five functions in `dplyr` are so widely used they are called “The Five Verbs”.
  - `select()`, `filter()`, `mutate()`, `arrange()`, `summarize()`
  - Specialized variants of The Five Verbs exist for specific purposes.

Summarizing Data Using `dplyr`

- The `select()` function returns a subset of the *columns* of a data frame. Let's use our `Scores` data frame.

```
> library(tidyverse)
> select(Scores, 1:3)
```

```
       Name Sex Exam.1
1     Susie   F     90
2      Phil   M     75
3 Margarida   F     92
4   Stephen   M     85
```

```
> ## select(Scores, Name, Sex, Exam.1) returns same result
> ## select(Scores, 3:1) reverses column order
```

Summarizing Data Using `dplyr`

- The `filter()` function returns a subset of the *rows* of a data frame.

```
> filter(Scores, Sex == "F")
```

```
       Name Sex Exam.1 Exam.2
1     Susie   F     90     87
2 Margarida   F     92     95
```

```
> filter(Scores, Exam.1 < 90)
```

```
     Name Sex Exam.1 Exam.2
1    Phil   M     75     71
2 Stephen   M     85     81
```

Summarizing Data Using `dplyr`

- The `mutate()` function allows us to add a new variable or transform a variable.

```
> mutate(Scores, Change = (Exam.2 - Exam.1) / Exam.1)
```

```
       Name Sex Exam.1 Exam.2      Change
1     Susie   F     90     87 -0.03333333
2      Phil   M     75     71 -0.05333333
3 Margarida   F     92     95  0.03260870
4   Stephen   M     85     81 -0.04705882
```

Summarizing Data Using `dplyr`

- You can do multiple calculations within the same `mutate()` command, and you can even refer to columns that were created in the same `mutate()` command.

```
> mutate(Scores, Change = (Exam.2 - Exam.1) / Exam.1,
+        Per.Change = round(Change*100, 2))
```

```
       Name Sex Exam.1 Exam.2      Change Per.Change
1     Susie   F     90     87 -0.03333333      -3.33
2      Phil   M     75     71 -0.05333333      -5.33
3 Margarida   F     92     95  0.03260870       3.26
4   Stephen   M     85     81 -0.04705882      -4.71
```

Summarizing Data Using `dplyr`

- The `arrange()` function allows us to order the rows by one or more columns in ascending or descending order.

```
> arrange(Scores, Exam.1) ## Default is ascending
```

```
      Name Sex Exam.1 Exam.2
1     Phil   M     75     71
2  Stephen   M     85     81
3    Susie   F     90     87
4 Margarida   F     92     95
```

Summarizing Data Using `dplyr`

- We order first by `Sex` and then by `Exam.1`, both in descending *nested* order.

```
> arrange(Scores, desc(Sex), desc(Exam.1))
```

```
       Name Sex Exam.1 Exam.2
1   Stephen   M     85     81
2      Phil   M     75     71
3 Margarida   F     92     95
4     Susie   F     90     87
```

Summarizing Data Using dplyr

- The summarize() function allows us to generate or calculate summary statistics for a given column in the data frame.

```
> summarize(Scores, Exam.1.mean = mean(Exam.1),
+           Exam.1.sd = sd(Exam.1))
```

```
  Exam.1.mean Exam.1.sd
1        85.5  7.593857
```

Summarizing Data Using dplyr

- The group_by() function is not one of The Five Verbs, but it is important in its own right. It tells dplyr functions to perform their actions separately for each ‘group’.

```
> summarize(group_by(Scores, Sex),
+           Exam.1.mean = mean(Exam.1),
+           Low.Score = min(Exam.1))
```

```
# A tibble: 2 x 3
  Sex   Exam.1.mean Low.Score
  <fct>       <dbl>     <dbl>
1 F              91        90
2 M              80        75
```

- Minor caution: be careful using group_by(). Some R gurus tend to use the ungroup() function after every group_by() for a few technical reasons.

Summarizing Data Using `dplyr`

- In `dplyr`, the so-called *pipe operator* `%>%` is a very useful command.
- For example, if we wanted to start with a data frame `x`, and first apply the function `f()`, then `g()`, and then `h()`:
  - Hard way: the usual R command would be `h(g(f(x)))`, a nested sequence of functions starting at the innermost set of parentheses
  - Easy way: using the pipe command `%>%`, this sequence of operations becomes `x %>% f() %>% g() %>% h()`

Summarizing Data Using dplyr

- The hard way (yikes!):

```
> summarize(group_by(filter(Scores, Exam.1 > 75),
+              Sex), Best.Score = max(Exam.2))
```

- The easy way:

```
> Scores %>% filter(Exam.1 > 75) %>% group_by(Sex) %>%
+ summarize(Best.Score = max(Exam.2))

# A tibble: 2 x 2
  Sex   Best.Score
  <fct>      <dbl>
1 F             95
2 M             81
```

Data Reshaping Using `tidyr`

- Many procedures in statistical software packages expect the data to be in the *long view*, where each row is a unit and each column is a distinct variable measured on the unit. Unfortunately, I often get data sets with repeated measurements on the same variable (e.g., Time) strung out horizontally on a single row; that is, the data are in the *wide view*.
- When this happens for a data set, we need to switch, or pivot, from the wide view to the long view in order to do our data analysis.
- An example will make this clear.

Data Reshaping Using `tidyr`

- Consider our `Scores` data set below. It is in the wide view. Specifically, values for `Exam.1` and `Exam.2` could be thought of as repeated measurements on the same variable `Midterm`, say, which has two levels `Exam.1` and `Exam.2`. Therefore, we will *reshape* the `Exam.1` and `Exam.2` columns into two columns: a new categorical variable `Midterm` and a new response variable `Score`.

```
> Scores
```

```
       Name Sex Exam.1 Exam.2
1     Susie   F     90     87
2      Phil   M     75     71
3 Margarida   F     92     95
4   Stephen   M     85     81
```

- The `tidyr` package (in the `tidyverse`) allows us to "tidy" our data frame; specifically, we use the `gather()` function to go from wide to long view for the `Scores` data frame.

Data Reshaping Using tidyr

- In the gather() function, key is the new categorical variable, value is the new response variable, and the last argument tells R which columns to 'gather' and apply this to.

```
> tidy.Scores <- Scores %>% gather(key = Midterm,
+                   value = Score, Exam.1:Exam.2)
> tidy.Scores
```

```
       Name Sex Midterm Score
1     Susie   F  Exam.1    90
2      Phil   M  Exam.1    75
3 Margarida   F  Exam.1    92
4   Stephen   M  Exam.1    85
5     Susie   F  Exam.2    87
6      Phil   M  Exam.2    71
7 Margarida   F  Exam.2    95
8   Stephen   M  Exam.2    81
```

Data Reshaping Using `tidyr`

- The `gather()` and `spread()` will likely be phased out in the future.
- They will be supplanted by the `pivot_longer()` and `pivot_wider()` functions.
- See the following vignette for details:
  - <https://tidyr.tidyverse.org/dev/articles/pivot.html>

Table Joins Using dplyr

- We often need to join together two data frames that do not have the same number of rows. Suppose we also have the following data frame and want to join it with our new data frame `tidy.Scores`.

```
> Scores02 <- data.frame(
+   Name  = c("Susie", "Joe", "Margarida", "Stephen"),
+   Area = c("CO", "NC", "ESP", "NC"),
+   Hobby = c("Horse", "Poker", "Horse", "Fish")
+ )
> Scores02
```

```
       Name Area Hobby
1     Susie   CO Horse
2       Joe   NC Poker
3 Margarida  ESP Horse
4   Stephen   NC  Fish
```

Table Joins Using `dplyr`

- We will use the function `full_join()` from the `dplyr` package to join these two data frames. It return all rows and all columns from both data frames. Where there are not matching values, it returns `NA` for the value missing.
- You will note that each of these data frames `tidy.Scores` and `Scores02` has a `Name` column. This is good practice to do.
- When we join these two data frames, the row that describes `Stephen`, say, from `Scores02` should be duplicated for each row in `tidy.Scores` that corresponds with `Stephen`.
- Notice `Joe` does not appear in `tidy.Scores` and `Phil` does not appear in `Scores02`. Therefore, after the join, "missing values" will be denoted as `<NA>` (for categories) and `NA` (for numbers) in the appropriate columns.

Table Joins Using `dplyr`

```
> tidy.Scores %>%
+   full_join(Scores02) ## Could specify by = "Name"
```

```
Joining, by = "Name"

       Name  Sex Midterm Score Area Hobby
1     Susie    F  Exam.1    90   CO Horse
2      Phil    M  Exam.1    75 <NA>  <NA>
3 Margarida    F  Exam.1    92  ESP Horse
4   Stephen    M  Exam.1    85   NC  Fish
5     Susie    F  Exam.2    87   CO Horse
6      Phil    M  Exam.2    71 <NA>  <NA>
7 Margarida    F  Exam.2    95  ESP Horse
8   Stephen    M  Exam.2    81   NC  Fish
9       Joe <NA>    <NA>    NA   NC Poker
```

Table Joins Using dplyr

- There are other types of joins we often use in practice. For example, A %>% left_join(B) return all rows from A, and all columns from A and B. Rows in A with no match in B will have NA values in the new columns. If there are multiple matches between A and B, all combinations of the matches are returned 👁 👁

```
> Scores02 %>% left_join(tidy.Scores) ## I chose order

Joining, by = "Name"

       Name Area Hobby  Sex Midterm Score
1     Susie   CO Horse    F  Exam.1    90
2     Susie   CO Horse    F  Exam.2    87
3       Joe   NC Poker <NA>    <NA>    NA
4 Margarida  ESP Horse    F  Exam.1    92
5 Margarida  ESP Horse    F  Exam.2    95
6   Stephen   NC  Fish    M  Exam.1    85
7   Stephen   NC  Fish    M  Exam.2    81
```

Graphs Using `ggplot2`

- The basic plotting commands in R are effective but the commands do not have a way of being combined in easy ways.
- `ggplot2` is a state-of-the-art data visualization package in R (a member of the `tidyverse`).
- On balance, `ggplot2` is every bit the equal of Tableau, Tableau Public, Data Studio, Power BI, etc., for data visualization. All have their pros and cons.
- The package does not fully anticipate what the user wants to do, but rather provides the mechanisms for combining different graphical components which the user chooses.

Graphs Using `ggplot2`

- *The Grammar of Graphics* (Wilkinson, 1999) provided a theoretical framework for the construction of graphics.
- `ggplot2` is based upon a layered grammar of graphics, an extension of Wilkinson's work. This "grammar" gives a framework which enables one to easily update a graphic by modifying a single feature at a time, and suggests aspects of a graph that can be built up component by component.
  - <https://ggplot2.tidyverse.org/reference/index.html>
- Along with `dplyr`, `ggplot2` has grown in use to become one of the most popular R packages. Companies now view the ability to work with these packages as a valuable technological skill.

Graphs Using ggplot2

- Let's create a *bar chart*. Here we consider a data set that gives the fuel efficiency of different classes of vehicles in two different years 1999 and 2008. This is a subset of data that the EPA makes publicly available and is included in the ggplot2 package as mpg.

```
> data(mpg) ## Loads data set
```

- There are two ways to determine what the columns are. First, we could submit str(mpg), which describes the 'structure' of an R object; in this case, our data frame. Second, we could use the Environment tab

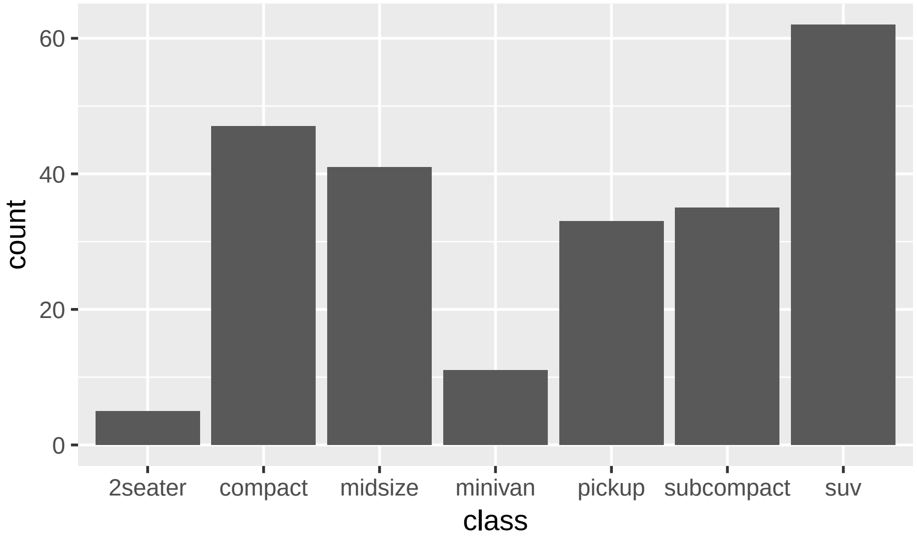
Graphs Using ggplot2

```
> plot01 <- ggplot(mpg, aes(x = class))
> plot01 <- plot01 + geom_bar()
```

- aes stands for *aesthetic*, a visual property of the graph. It says the x-axis will be the car's class, which is indicated by the column named class.
- geom_bar() creates a bar chart. geom stands for *geometric object*. They represent different ways to display variables and the relationship between them.
- When I create graphs using ggplot2, I prefer to code sequentially and save the graph.

Graphs Using ggplot2

```
> plot01
```



Graphs Using ggplot2

- Suppose we were interested in the distribution of highway miles per gallon (hwy).

```
> plot02 <- ggplot(mpg, aes(x = hwy))
> plot02 <- plot02 + geom_histogram()
```

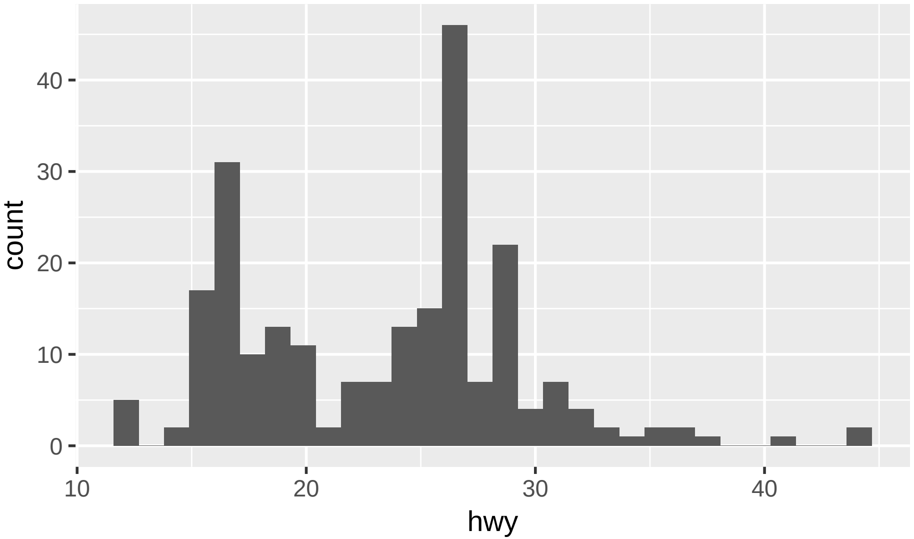
- geom_histogram() creates a frequency histogram.
- You will get an ominous message about the default of 30 bins, which gives us too fine a partition of the data. To get a sensible number of classes for a histogram, use the Freedman-Diaconis rule:

```
> nclass.FD(mpg$hwy)
```

```
[1] 11
```

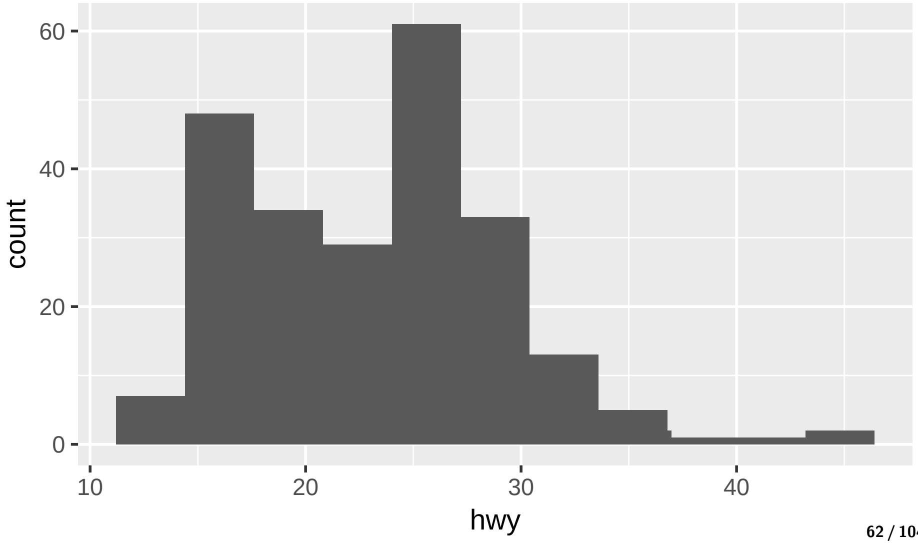
Graphs Using `ggplot2`

```
> plot02
```



Graphs Using ggplot2

```
> plot02 <- plot02 + geom_histogram(bins = 11)
> plot02
```



Graphs Using ggplot2

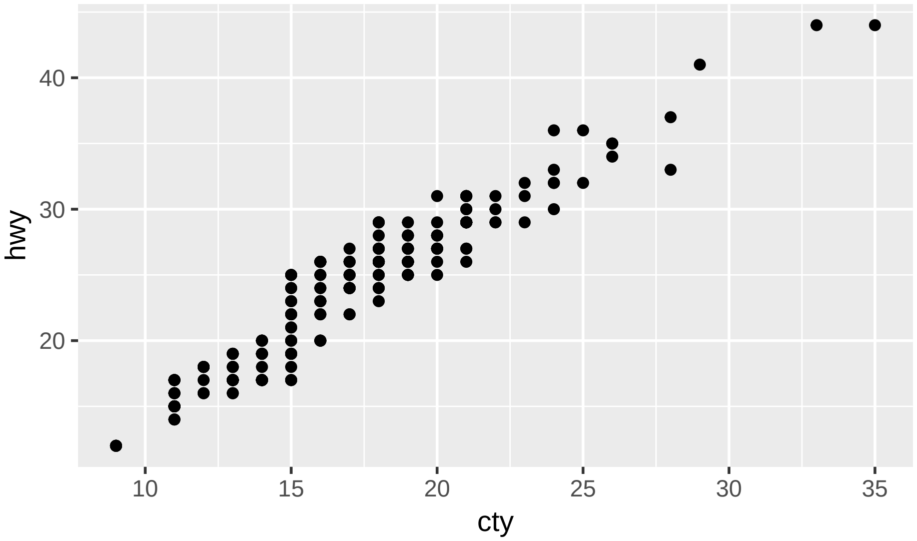
- Suppose we were also interested in the relationship between city miles per gallon (cty) and hwy.

```
> plot03 <- ggplot(mpg, aes(x = cty, y = hwy))
> plot03 <- plot03 + geom_point()
```

- geom_point() draws points for a scatter plot.
- There will be a minor *overplotting* problem. Counterintuitively, adding random noise to a plot can sometimes make it easier to read. *Jittering* is particularly useful for small data sets that are discretized. We will demonstrate jittering momentarily.

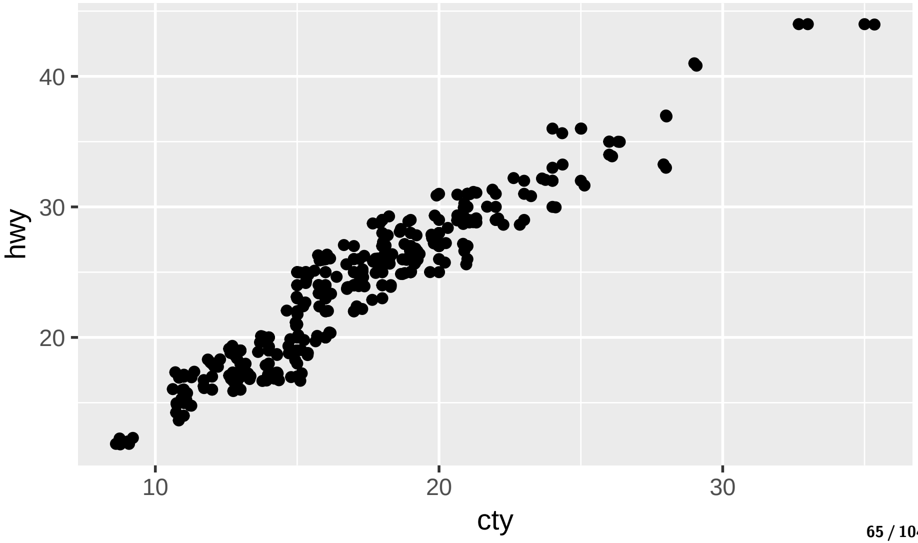
Graphs Using `ggplot2`

```
> plot03
```



Graphs Using ggplot2

```
> plot03 <- plot03 + geom_jitter()
> plot03
```



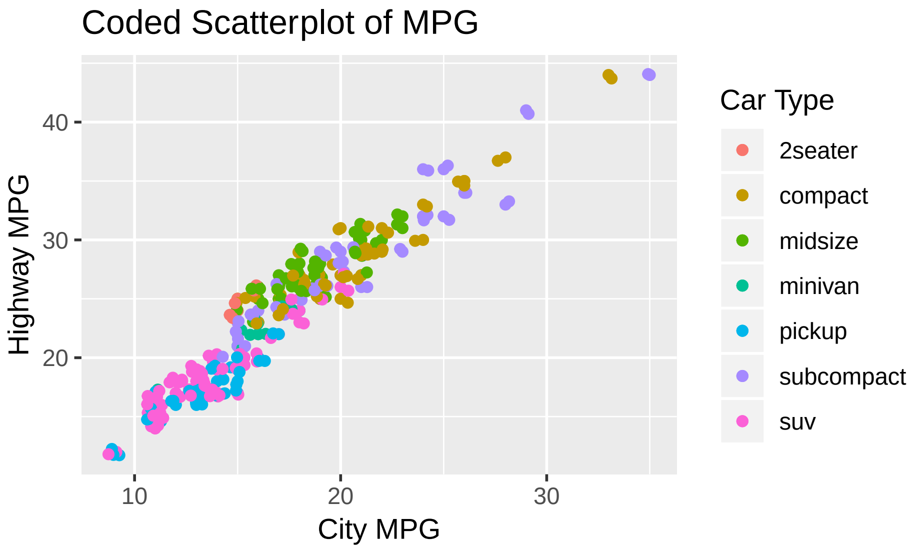
Graphs Using ggplot2

- We can also make a coded scatter plot. For example, suppose we were interested in the relationship between cty and hwy by class.

```
> plot03 <- ggplot(mpg, aes(x = cty, y = hwy,
+                           color = class))
> plot03 <- plot03 + geom_point()
> plot03 <- plot03 + geom_jitter()
> plot03 <- plot03 + labs(x = "City MPG", ## Add bling :)
+           y = "Highway MPG",
+           title = "Coded Scatterplot of MPG",
+           color = "Car Type")
```

Graphs Using ggplot2

```
> plot03
```



Graphs Using ggplot2

- Lastly, we can also make side-by-side box plots. For example, suppose we were interested in the relationship between cty by year, where year is the year of manufacture ({1998, 2008}).

```
> plot04 <- ggplot(mpg, aes(x = factor(year),
+                           y = cty))
> plot04 <- plot04 + geom_boxplot()
```

- factor() tells R to treat year as a category, not an integer.
- geom_boxplot() creates box plots.

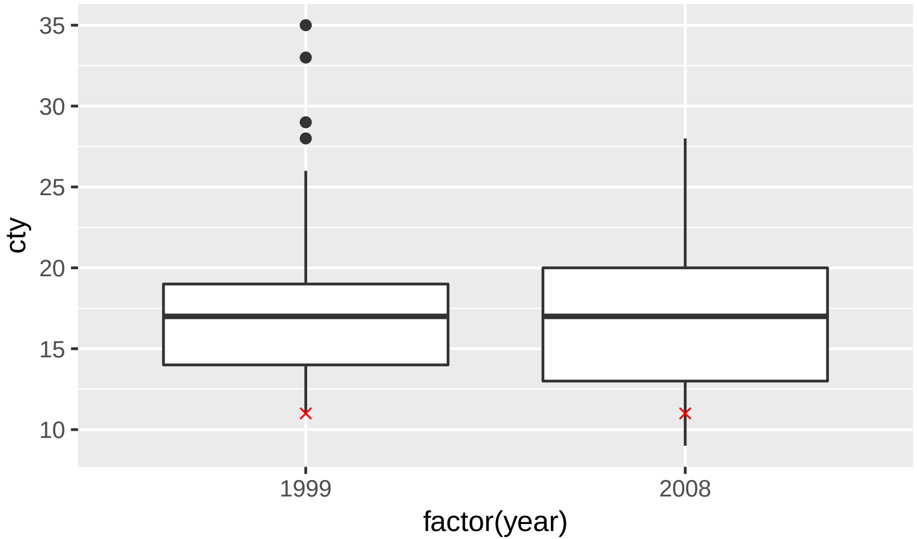
Graphs Using ggplot2

- ggplot2 is a powerful, extensible way to obtain even the most complicated (but beautiful!) graphs in order to visualize your data.
- For example, let's suppose you work for the EPA and are interested in the 5th percentile city miles per gallon for each year. You then want to superimpose these values directly on each box plot. This is straightforward to do in ggplot2.

```
> plot04 <- plot04 + stat_summary(fun.y = quantile,
+           fun.args = list(probs = 0.05),
+           geom = "point", color = "red",
+           shape = "cross")
```

Graphs Using `ggplot2`

```
> plot04
```



Useful Basic Programming

- It is often necessary to write R code that can perform different actions depending on the data or to automate a task that must be repeated many times. We now cover just a glimpse of these tools.
- Here is an '`if`' statement. (The '`else`' part is sometimes optional.)

```
> flip <- rbinom(n = 1, size = 1, prob = 0.5)
> flip ## Flip a coin, and get a 0 or 1
```

```
[1] 0
```

```
> if(flip == 0){ ## Convert the 0/1 to Tail/Head
+   flip <- "Tail"
+ }else{ ## Keep else on same line as closing } for if
+   flip <- "Head"
+ }
> flip
```

```
[1] "Tail"
```

Useful Basic Programming

- In my experience, the `ifelse()` function is very useful because it operates on vectors.

```
> x <- 1:5
> x
```

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

```
> ifelse(x <= 2, "Small", "Large")
```

```
[1] "Small" "Small" "Large" "Large" "Large"
```

- Be aware that there is also an `if_else()` function in `dplyr` that is faster (but a bit more temperamental).

Useful Basic Programming

- A `while` loop will repeat a section of R code over and over until a stopping condition is met.

```
> x <- pi ## Initialize x
> while(x < 50){ ## Multiple lines of R code
+   x <- 2*x ## Important to update x
+   print(x)
+ }
```

```
[1] 6.283185
[1] 12.56637
[1] 25.13274
[1] 50.26548
```

Useful Basic Programming

- When we know in advance how many times we should go through a loop of R code, a ‘for’ loop can be quite useful.

```
> stuff <- NULL ## Create an empty object
> for(i in 1:4){ ## Add elements to stuff
+   stuff[i] <- i*i
+   print(stuff)
+ }
```

```
[1] 1
[1] 1 4
[1] 1 4 9
[1]  1  4  9 16
```

- In the for loop, ‘i’ can be called anything. It’s a variable that you choose that is a placeholder.

Useful Basic Programming

- Sometimes it can be useful to define your own functions. A rigorous treatment of R programming would require its own short course :)
- Let's start by defining a function `F_to_C` that converts temperatures from Fahrenheit to Celsius.

```
> F_to_C <- function(temp_F){ ## temp_F is placeholder
+   temp_C <- ((temp_F - 32) * (5 / 9)) ## Do conversion
+   return(temp_C) ## Show result temp_C
+ }
```

Useful Basic Programming

- Calling our own function is no different from calling any other function already in R. We can do it for one temperature (°F) or a vector of temperatures (°F).

```
> F_to_C(32) ## Check  it!
```

```
[1] 0
```

```
> lots.temps <- c(0, 32, 68)
> F_to_C(lots.temps)
```

```
[1] -17.77778   0.00000  20.00000
```

```
> my.Cs <- F_to_C(lots.temps)
> my.Cs ## Can assign function output to object
```

```
[1] -17.77778   0.00000  20.00000
```

Working With Strings Using `stringr`

- Character strings can be an important class of data. Data being read into R often come in the form of character strings where different parts might mean different things.
  - For example a sample ID of "DSL06150902" might be a so-called "Event Code" where DSL represents a training program acronym, 061509 is the date in MMDDYY format, and 02 represents a building designation for some location.
- We need to have a set of utilities that allow us to split, combine, and examine character strings in a easy and consistent fashion. These will come in handy when managing and wrangling data and in more sophisticated use (e.g., text mining).
- Here we will introduce a few of the more commonly used functions from the `stringr` package from the `tidyverse`.

Working With Strings Using stringr

- The str_c() allows us to concatenate two strings or two vectors of strings. This is demonstrated through the following example.

```
> first.names <- c("Michael", "Sarah", "Mitch", "Jennifer")
> last.names <- c("Long", "Pearson", "Jones", "Smith")
> full.names <- str_c(first.names, last.names)
> cbind(first.names, last.names, full.names)
```

```
     first.names last.names full.names
[1,] "Michael"   "Long"     "MichaelLong"
[2,] "Sarah"     "Pearson"  "SarahPearson"
[3,] "Mitch"     "Jones"    "MitchJones"
[4,] "Jennifer"  "Smith"    "JenniferSmith"
```

Working With Strings Using stringr

- The str_length() function calculates the length of each string in the vector of strings passed to it. This could be important in text analytics.

```
> str_length(full.names)
```

```
[1] 11 12 10 13
```

- A powerful string operation is to take a string and match some pattern within it. For example, we might be interested in mining through full.names to detect if a pattern occurs.

```
> str_detect(full.names, pattern = fixed("Mi"))
```

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

Working With Strings Using `stringr`

- To figure out positions where the first `Mi` patterns are in each string, we can use the `str_locate()` function.

```
> str_locate(full.names, pattern=fixed("Mi"))
```

```
     start end
[1,]     1   2
[2,]    NA  NA
[3,]     1   2
[4,]    NA  NA
```

- One of my favorite `stringr` functions is `str_count()`. It counts the number of designated matches in a string (e.g., the number of vowels/full name). Note that '|' is a logical operator that stands for 'or'.

```
> str_count(full.names, "a|e|i|o|u|A|E|I|O|U")
```

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

Working With Dates and Times Using `lubridate`

- Using dates in any software package can be surprisingly difficult.
- They require some special organization because there are several conventions for how to write them and because the sort order should be in the order that the dates occur in time.
- We are going to use the package `lubridate` (which belongs to the tidyverse) to work with dates and times. `lubridate` gets installed as part of the tidyverse installation. However, because it's so specialized, it does not belong to the *core* tidyverse, so you have to load it explicitly with `library(lubridate)`.

Working With Dates and Times Using `lubridate`

- To create a `Date` object, we need to take a string or number that represents a date and give R a *consistent order* for the date so that it can uniquely identify which piece is the year, the month, and the day.
- We demonstrate the use of the `mdy()` function. Many other functions exist depending on how the date is specified.

```
> library(lubridate)
> mdy('April 16, 2019', '4-16-19', '4-16-2019', '4/16/19')
```

```
[1] "2019-04-16" "2019-04-16" "2019-04-16" "2019-04-16"
```

Working With Dates and Times Using `lubridate`

- Suppose we also had the hours and minutes. Then we can add this time to the date if we wanted to using the `mdy_hm()` function.

```
> mdy_hm('April 16, 2019 1:00 PM', '4-16-2019 13:00')
```

```
[1] "2019-04-16 13:00:00 UTC" "2019-04-16 13:00:00 UTC"
```

- UTC stands for Coordinated Universal Time. However, there are options for specifying a different time zone (e.g., EST, EDT, etc.). See `OlsonNames()` for your personal favorite. Notice the `tz` option.

```
> mdy_hm('4-16-2019 13:00', tz = "US/Eastern")
```

```
[1] "2019-04-16 13:00:00 EDT"
```

Working With Dates and Times Using `lubridate`

- The `lubridate` package provides many functions for extracting information from the date and time. We demonstrate just a few of them.

```
> x <- mdy_hm('4-16-2019 13:00', tz = "US/Eastern")
> year(x) ## Return year
```

```
[1] 2019
```

```
> hour(x) ## Return hour of the day
```

```
[1] 13
```

```
> wday(x) ## Return day of the week (Sunday = 1)
```

```
[1] 3
```

```
> yday(x) ## Return day of the year
```

```
[1] 106
```

Working With Dates and Times Using `lubridate`

- Once we have two or more `Date` objects defined, we can perform appropriate mathematical operations. For example, we might want to the know the number of days there are between two dates, aka ‘tenure time’.
- For example, for some person, suppose an event takes place on 2019-9-15 and registration was on 2019-6-23.

```
> Register <- ymd('2019-6-23') ## "time when"
> Event <- ymd('2019-9-15') ## "time when"
> Days.Until <- Event - Register
> Days.Until ## "time duration"
```

```
Time difference of 84 days
```

Working With Dates and Times Using `lubridate`

- Let's pause for a moment. This is a great demonstration of the power of R as a tool in data science. Why?
- We have demonstrated proof of concept for one person. But what if a company had such data for thousands of people? Scaling up to get *everyone's* time difference would present no challenge at all to R. The creation of a 'time difference' variable is sometimes called *feature extraction* among machine learning modelers.
- This feature could then be used in a machine learning model to help us understand and predict some behavior that might be of interest to the company. Of course, the modeling would be done in R in a seamless workflow.

R Scripts

- I always work with script (.R) files instead of typing directly into the console. If you have a data analysis of any substance, it is virtually impossible to write all your R code correctly the first time **and** remember all that you did later on if the need arises.
- Writing a script file fully documents how you did your analysis. Hence, having a script makes it easy to re-run an analysis after a change in the data (additional data values, transformed data, or removal of outliers) or if a journal referee wants revisions.

R Scripts

- It often makes your script more readable if you break a single command up into multiple lines. R will disregard all whitespace (including line breaks) so you can safely spread your command over multiple lines.
- It's also useful to leave comments in the script for things such as explaining a tricky step, who wrote the code and when, or why you chose a particular name for a variable. So 'comment' your R code and be kind to your future self!

R Scripts

- One way to create a new .R script in RStudio is to go to File, New File, R Script. This opens a blank script in the Source window where you can type commands and functions.
- Type your R code in the Source window and then you can execute the code as follows:
  - Click the line of code you want to run, and then press Cmd + Return or click the Run button.
  - Highlight the block of code you want to run, and then press Cmd + Return or click the Run button.
  - Run the entire script by clicking anywhere in the source editor and then pressing Cmd + Shift + Return

R Scripts

- Working with R scripts is an acceptable way of documenting what you did, but the script file doesn't contain the actual results of commands that were run, nor does it show you the plots.
- Also, anytime I want to comment on some output, it needs to be offset with the commenting character #.
- It would be nice to have all the commands, the results, and comments merged into one document. This is what the R Markdown file does for us.

R Markdown

- It is very inefficient to copy output and save figures from RStudio and paste them into an MS Word document.
- An *R Markdown* document (or .Rmd file) is a way of making, or "knitting", a completely self-contained reproducible research document. It has the following:
  - your executable R code in "code chunks"
  - your R output where you want it
  - your text where you want it
- Thus, your entire data analysis and report is forever etched in stone. If you make a mistake, simply go back to the .Rmd file, make the correction, re-run the file, and the entire report is reproduced and updated. (I have actually created the slides for this workshop using RMarkdown.)
- R Markdown is also great for 'automating' the same report with distinct values for various important inputs.

R Markdown

- As a bit of history, R Markdown is an implementation of so-called *Markdown* syntax that cooperates with R. Markdown started as an easy way to write web pages and give instructions for how to do typesetting with a software package called LaTeX.
  - <https://rmarkdown.rstudio.com>
  - <https://bookdown.org/yihui/rmarkdown/>
- You need the `rmarkdown` package, but you don't need to explicitly install it or load it, as RStudio automatically does both when needed.
- There is another way to do reproducible research (Sweave), but R Markdown is easier to learn and use, and much more popular.

R Markdown

- One way to create a new .Rmd file in RStudio is to go to File, New File, R Markdown. A menu will appear asking you for a title, author, and output format (HTML, PDF, or Word). In order to create a PDF, you'll need to have LaTeX installed, but the default HTML output should work fine.
  - The `TinyTeX` package in R provides a lightweight and easily maintained version of LaTeX. This might be enough to get you by if you want PDF output.
- In my own experience, MS Word output works well but not as good as the other two formats. Be prepared to have a few more challenges along the way.

R Markdown

Let's take a look at the basic structure of an R Markdown file by carefully inspecting the .Rmd file you just created. It has the following components 👁 👁

1. A YAML header
   - The one given to you by default is very basic. There are many options available. We will add a few things to this momentarily.
2. R code chunks
   - Each chunk forms a sandwich of R code wrapped by a pair of triple backticks. The first group of backticks must be followed by {r}.

~~~
```{r}
your lines of R code go here
```
~~~

3. Text, equations, etc., with simple formatting
   - R Markdown is very good at weaving these in and out of your R code and output.

R Markdown

- To compile the .Rmd file into an HTML file, near the upper left-hand corner of the Source window in which you are doing your editing, there is a button entitled Knit (it has a blue ball of yarn on it). Click on it to generate the HTML file
- Go to Help, Markdown Quick Reference. This opens a short guide in the Help window that shows you how to do various forms of text formatting
    - For example, if you cursor down, you will see a brief explanation on how to place images into your .Rmd file (on the web or local files in the same directory).

R Markdown

- Let's try out some modifications. Find some whitespace in your .Rmd file and type the following. When you are done, knit the document again 👁 👁
  - Type `*Hello*` (will give –> *Hello*)
  - Type `**Hello**` (will give –> **Hello**)
- For code chunks, you can insert them manually *or* hit the Insert button icon in the editor toolbar above the Source window 👁 👁
- To test out the code chunk(s), hit the green arrow on the right-hand side of the code chunk *or* use the Run button icon in the editor toolbar above the Source window 👁 👁

R Markdown

- The chunk header structure is ` ```{r optional chunk name or number, comma separated options} `
- One good reason to give your code chunks numbers is that it makes it easier to debug your program.
- There are about 60 *options* that you can use to customize your code chunks. In addition to showing the code and output, you can do other things. For example, using `echo = FALSE` shows only the output in the finished file and not the R code.
- To see/modify the code chunk options, hit the gear icon on the right-hand side of the code chunk.
  - Click on the Output drop-down menu and notice the choices.
  - Click on `Chunk options` and notice what happens.
  - For code chunk 2, erase `echo = FALSE`, knit the document again, and notice what happens 👁 👁

R Markdown

- One nice feature of R Markdown is that it gives you the ability to insert complex mathematical expressions using the powerful typesetting system LaTeX. Since this is specific and complex, I only touch upon it here.
- In your R Markdown document, you can include LaTeX code by enclosing it with dollar signs. LaTeX is not based on "what you see is what you get". So you might type `$\alpha = 0.05$` in your text, but after you knit the document, you'll see $\alpha = 0.05$ 👁 👁
- If you want a mathematical equation to be centered on its own line, then enclose it with double dollar signs. So you might type `$$\bar{X} = \frac{\sum X_i}{n}$$` in your text, but after you knit the document, you'll see:

$$\bar{X} = \frac{\sum X_i}{n}$$

- A good LaTeX resource is at <https://en.wikibooks.org/wiki/LaTeX>

R Markdown

- Rather than accepting R's default output from a code chunk, suppose we want the output to be in the form of a nicer table. Here are your options:
    - Make a table by hand using LaTeX.
    - R Markdown has three basic options: simple, grid, and pipe tables. Of the three, the pipe tables seem to be the easiest to set up. Truth be told, tables are not R Markdown's strong suit. The best documentation seems to be here: <http://pandoc.org/README.html#tables>.
    - There are a couple of different packages that convert a data frame to a beautified table. The most basic approach uses the `kable()` function from the `knitr` package. More complicated approaches use the `pander` or `kableExtra` packages.
    - For example, install the `pander` package. For code chunk 1, add the line `library(pander)`, and then `pander(summary(cars), style = "rmarkdown")`, knit the document again, and notice what happens 👁 👁

R Markdown

- You can control many other document settings by tweaking the parameters of the YAML header.
- Go to <https://github.com/philturk/R_Wshop> and click on the big green button that says `Clone or download`, then click on `Download ZIP`. On my Mac, this saves the entire repository ("repo") as a `R_Wshop-master` folder to my Downloads. Click on `Ex01.Rmd`; it should open up in RStudio. This file is almost identical to your .Rmd file with a few exceptions 👁 👁
- Examine my YAML header (everything between and including the two `---` lines). Modify the `author`, `date`, and `subtitle` fields to your own personal taste.
- Knit the document again, and notice what happens. If you have LaTeX installed on your computer, click on the Knit drop-down menu, click on `Knit to PDF`, and notice what happens 👁 👁

R Markdown

- A huge timesaver is the ability to easily create a bibliography using R Markdown. I will show you one way that imports references from your own master repository.
- To use this feature, specify a bibliography file using the `bibliography` field in your YAML header. The field should contain a path from the directory that contains your .Rmd file to the file that contains the bibliography file.
- For example, in your `R_Wshop-master` folder, I have created a file called `citations.bib`. For now, determine its path (e.g., /Users/pturk/R/Workshop/citations.bib). Copy this path. Go back to the `Ex01.Rmd` file and overwrite the path currently in the `bibliography` field. Finally, uncomment the `bibliography` field, i.e., remove the #. Knit the document again, and notice what happens 👁 👁

R Markdown

- You can use many common bibliography formats but my favorite is BibTeX. If you open up `citations.bib` in a text editor, you will see the references all have a similar construction 👁 👁
- Almost every reference has a BibTex reference. University libraries, Google Scholar, literature databases, etc., have them as ‘ready-to-go’ downloadable files that you then just copy-and-paste into your .bib file.
- Advantages:
    - You need to copy-and-paste or type each reference only once.
    - The style of all your citations in any given document will be consistent.
    - Your bibliography is automatically generated.
    - Your bibliography uses only what you cite.
    - Your bibliography can be customized to journal style (e.g., APA).
- My .bib file has hundreds and hundreds of references and lives in *one location* on my computer.

R Markdown

- To create a bibliography, notice how I typed `## References` at the bottom of `Ex01.Rmd`. You could also use `## Bibliography`. They are two keywords
- Observe how I created the citations in `Ex01.Rmd`. Other options exist but we do not discuss them here
  - Citations go inside square brackets. If there are multiple citations, then they are separated by semicolons. Each citation must be denoted by '`@`' + the citation identifier from the BibTex reference in your .bib file. Use the syntax `[@name of BibTex reference]`.
  - For an in-text citation, remove the square brackets and use `@name of BibTex reference`.

WESTERN DATA ANALYTICS

Thank you for your attention and time!