

Module 3: Assembling Data

Table of Contents

Module 3: Assembling Data	1
Module 3 Overview	2
Module 3: Introduction	2
Lesson 3-1 Assembling Data	5
Lesson 3-1.1 Assembling Data	5
Lesson 3-2 Data Types	14
Lesson 3-2.1 Data Types.....	14
Lesson 3-3 More on Functions.....	23
Lesson 3-3.1 More on Functions	23
Lesson 3-4 Packages	36
Lesson 3-4.1 Packages	36
Lesson 3-5 Introduction to Other Data Types.....	47
Lesson 3-5.1 Introduction to Other Data Types.....	47
Lesson 3-6 Creating Date Types	59
Lesson 3-6.1 Creating Date Types	59
Lesson 3-7 Calculations with Dates	71
Lesson 3-7.1 Calculations with Dates	71
Lesson 3-8 Factors	82
Lesson 3-8.1 Factors	82
Lesson 3-9 Logical Type and Relational Operators	96
Lesson 3-9.1 Logical Type and Relational Operators	96
Lesson 3-10 Character Strings.....	108
Lesson 3-10.1 Character Strings	108
Module 3 Conclusion	126
Module 3: Conclusion.....	126

Module 3 Overview

Module 3: Introduction

Photography or videography
is a whole lot more than just
pushing a button on a camera.

[MUSIC] What stands out to you when you see this photo studio? To me, it's the lights. By my count, there are at least 16 lights placed in strategic places around the white backdrop. It also looks like there are at least three different types of lights, umbrella panel and some other kind. The point is that photography is a whole lot more than pushing a button on a camera. Specifically, before photos are taken, a lot of time is spent setting up the lights and creating the right environment. Well, just as photography deals with a whole lot more than pressing a button on a camera, business analytics deals with a whole lot more than just running an algorithm.

Data Assembly Topics

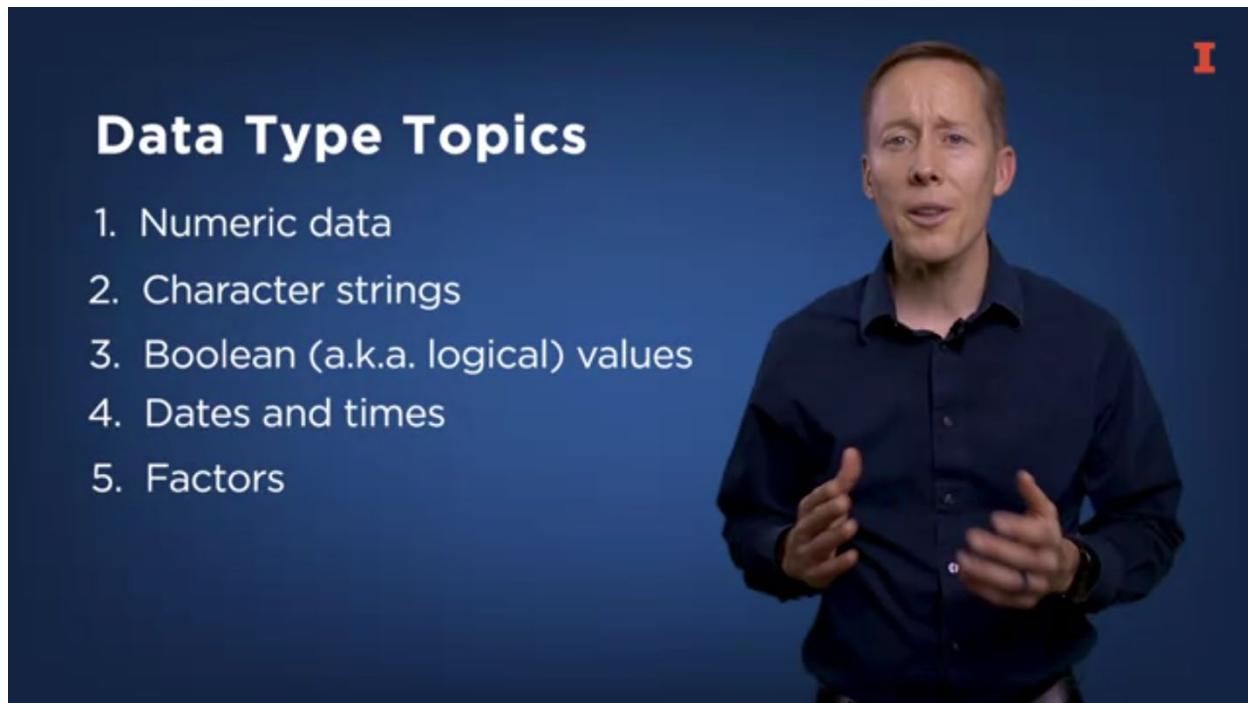
1. Where to find data
2. Approaches for organizing data



One of the key elements that takes place before the algorithm is run is assembling the data. In this module, we will begin by covering some general aspects of assembling data for business analytics, such as where to find data and different approaches for organizing it. There are many ways to organize data, but for our purposes we will stick with tabular data in which an observation is a row and a characteristic is a column. After discussing some general issues, associated with assembling data, we will then focus on the conceptual topic of data types.

Data Type Topics

1. Numeric data
2. Character strings
3. Boolean (a.k.a. logical) values
4. Dates and times
5. Factors



Numeric data is probably the most obvious data type. Numeric data includes data that is stored as an integer or a fraction. You see this a lot in business data. However, it's important to recognize that other data types like character strings, Boolean or logical values, dates and times and factors. These other data types allow you to aggregate data in different ways, such as by product line or, by a time period, like a month or a year. Moreover, Text Analytics is an important topic in business in and of itself. Text Analytics makes it possible to capture trends in social media, business news and managers reports. Text Analytics is also used for creating chatbots. In short, it's important to understand some issues with non numeric data types, so we will also give you firsthand experience working with them.

The video thumbnail features a man with short brown hair, wearing a dark blue button-down shirt, standing against a solid blue background. He is gesturing with his hands, with his thumbs pointing upwards. In the top right corner of the thumbnail, there is a small red letter 'I'. The title 'Function Topics' is displayed in white text at the top left of the thumbnail area.

Function Topics

1. Using arguments in functions
2. Installing packages of functions

Finally, we will take time to take a detailed look at functions, because they will make it easier to process numeric and non numeric data types. Just like an Excel, functions in a data analytic language allow you to perform a task by typing the function name and then entering the argument values. We will also show you how to install and use packages or modules, which contain functions that are used for performing interrelated tasks. For example, there's a package that contains functions to process date and time data, and there's another package that contains functions that are used for processing character strings. As you get into the data assembly process, remember that it's not unusual to spend time preparing data before it can be analyzed. In fact, sometimes great insights come as a result of the effort that goes into assembling the data. [MUSIC]

Lesson 3-1 Assembling Data

[Lesson 3-1.1 Assembling Data](#)



Interview with University of Illinois Alumnus
Luis Guillamo, Director of Analytics and
Application Development at the NFL Buffalo Bills

Question:

How much time do you spend assembling the
data vs. analyzing the data?

In this lesson, you'll learn more details about assembling data. The second step in the fact framework. Assembling data is critical because it often has a direct impact on how effectively the question can be answered. Interestingly, it often requires more time than the actual calculation of the results. I asked Luis Guillermo how much time he spends assembling the data versus analyzing the data. Here's his response.



Yeah, the amount of time that I spend and data preparation is about 80 percent of the time where 20 percent is dedicated to actually doing calculations. Obviously that can vary from time to time. But the main problem is data quality, data accuracy, making sure that it's correct, that it's shaped in the way that you need it to be able to do your analytics, ensuring that you have everything that you need essentially. That's really the key part. Once you have everything in the form that you need it, putting a model on top of it is very simple. If you're adept at interpreting those models, then you'll go quite far, quite quickly. I think it's clear, at least from Louis perspective, the assembling data takes a lot of time. Let's talk about why that's the case.



I Steps for Assembling Data

- Finding data
- Extract, transform, load (ETL)
- Data wrangling

Assembling data typically involves several steps. Finding data, extracting the data, transforming the data, and loading the data, or ETL and data wrangling. The importance of finding relevant data can't be overstated. Data is often protected for safety and regulatory reasons, as well as to protect trade secrets. Oftentimes people don't know what data sets exist, even within their organization. Or even if employees know that a data set exists and may be hard to access due to security and privacy concerns. For instance, it's unlikely that the HR department will share employee salary information with the marketing department. Many companies now employ chief data officers or CDOs.



Chief Data Officers “have been chartered with improving the efficiency and value-generating capacity of their organization’s information ecosystem. That is, they’ve been asked to lead their organization in treating and leveraging information with the same discipline as its other, more traditional assets.”

Laney, Doug. *Infonomics*, p. 9

In his book *Infonomics*, university of Illinois Alumni, Doug Laney suggests that most chief data officers have been chartered with improving the efficiency and value generating capacity of their organization's information ecosystem. That is, they've been asked to lead the organization in treating and leveraging information with the same discipline as its other more traditional assets. Thus, CDOs should not only keep track of data that's available to employees of the organization, but they should also make sure that the right employees know how to find out what data is available, as well as how to get access to it. Data sources may include internal and external data. For instance, if you're a manager who's trying to find out the factors that caused a decrease in sales for various regions. Then you'd probably want to make sure that you have information related to sales and location. But that's not all. You'd also want to gather data about potential factors that could influence sales for each region. Perhaps you'd want to find wheather data and data related to the types of homes, like single-family homes two family homes, and number of rooms per home that are in each region.



A Few Data Sources on the Web

- www.data.gov
- www.google.com/publicdata/directory
- aws.amazon.com/opendata/public-datasets/
- docs.microsoft.com/en-us/azure/sql-database/sql-database-public-data-sets
- <https://www.kaggle.com/datasets>
- <https://data.world/>
- <https://www.gapminder.org/data/>

There are thousands of publicly available data sets that could be considered. Governmental agencies, at least in the United States, make many data sets available to the public, including weather data, financial data for companies whose stocks are traded on US exchanges, census data, tax data, and health data. You can browse thousands of data sets on www.data.gov.

There are many other interesting data sets available for many other websites as well. If the question is important enough, you'll want to gather your own data. Gathering your own data can be done in many ways. For instance, you could create a survey to gather your own data. Or you can create a web crawler to gather data that is dispersed on multiple websites. Or you can start measuring something that previously hadn't been measured. Once you've identified a data set, you'll want to extract data from where it resides, transform it into a structure that meets your needs, and then load it into a data processing tool.

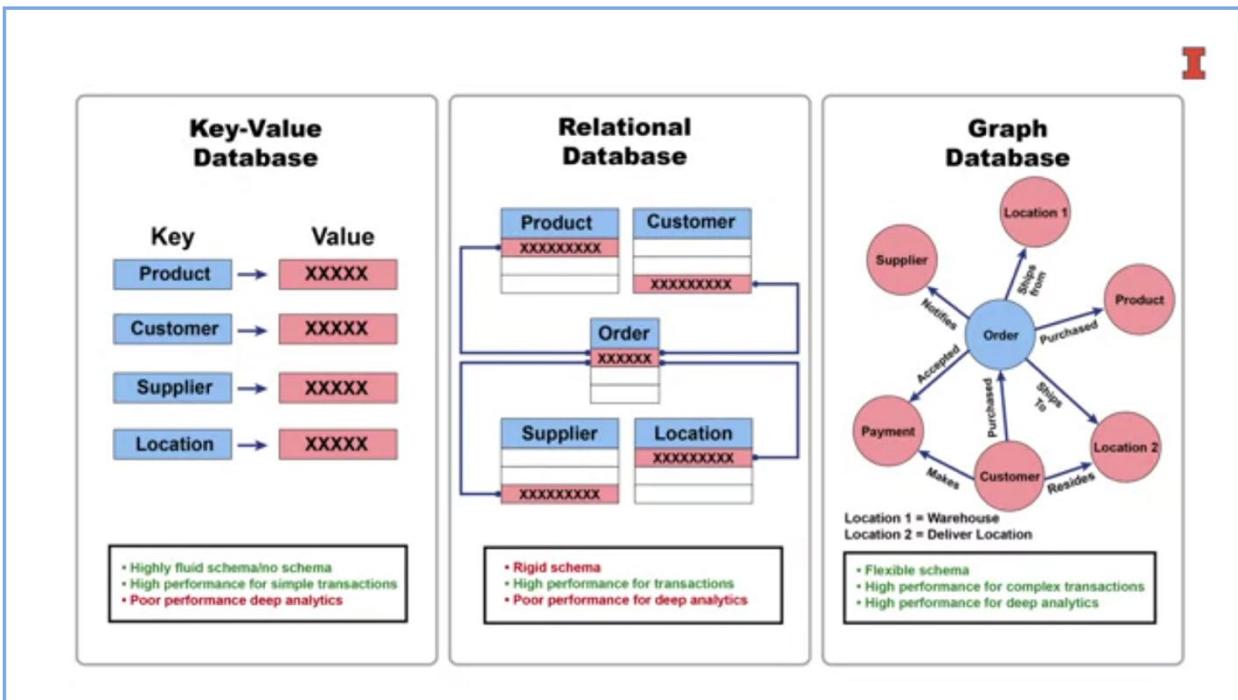


I

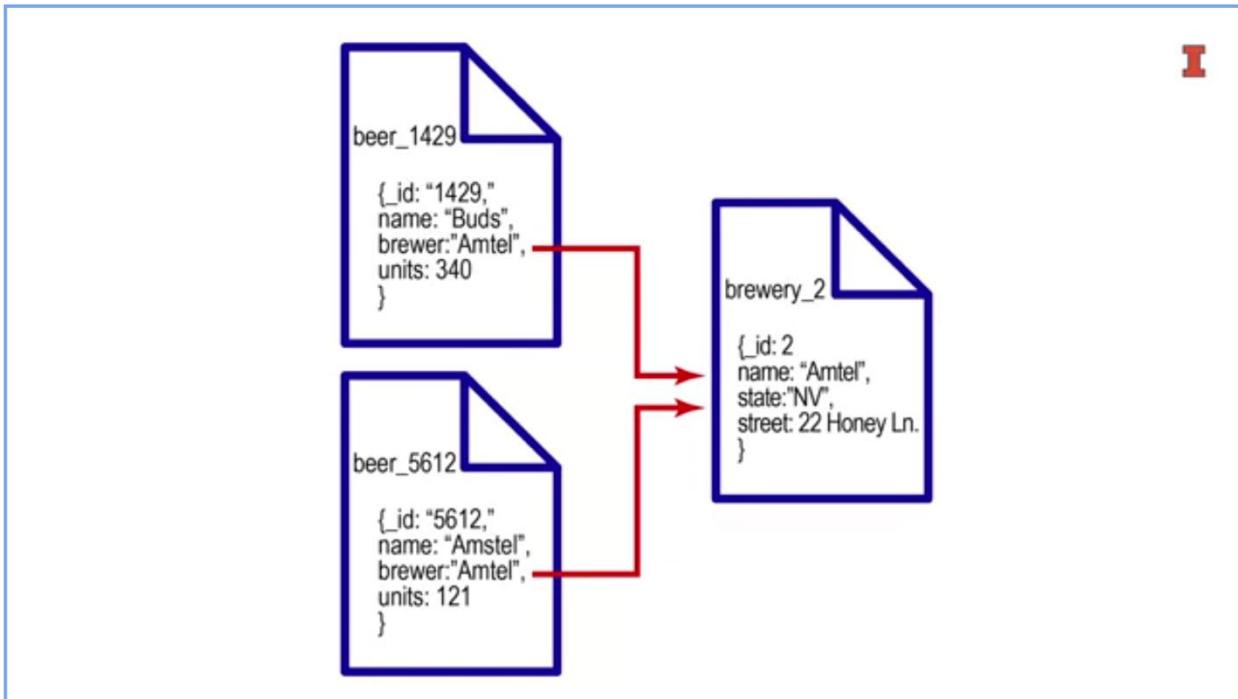
ETL

- Extract
- Transform
- Load

This is frequently abbreviated as ETL for extract, transform, and load. This is important because data repository store data in a variety of formats and then return data to the user in a different variety of formats. Typically, you'll want to transform the data to a table format so that you can perform the visualization or analysis. However, other formats also exist, such as graph database format. If you wanted to analyze graph data and you'd want to transform the data accordingly.



The most commonly known data repository is a relational database, which stores data in multiple tables. A Structured Query Language, or SQL, is often used to extract the desired subset of data which is then returned in a table format. This is nice because the data is already in a table format and it's ready to use. Here's some other ways in which data is stored and extracted. If you're gathering data from websites, then you'll need to gather HTML or XML data, which has lots of tags to identify the different pieces of data.



You'll then have to extract the key pieces of information and then transform it to a table format. Sometimes data is stored in a relational database, but it's accessed using an Application Programming Interface or API, which has a structure like a website rather than a SQL query. The data is often returned in JavaScript Object Notation or JSON format. This format is a series of embedded lists separated by curly braces, colons, and commas. As with HTML and XML data, the key pieces of data would need to be converted into a table format before it can be analyzed. Once the data is in a table format, you'll most likely need to then go through a data wrangling or data munging process.

This process includes cleaning data, combining data with other data, cleaning it again, perhaps combining it again, and then cleaning it again and then changing its shape. While I realized that you probably don't appreciate many of these terms and this process right now, I hope you get the idea that it's not a straightforward linear process.



Data Wrangling

- Cleaning data
 - Combining data with other data
 - Cleaning data again
 - Combining data with other data again

These steps associated with assembling Data are often iterated. During the data wrangling process, you may realize that some of the data you found has too many errors. You may decide

that you need to find a different data set and then you start the process over. The underlying point is that assembling the right data is so important and it often requires a lot of work.

Lesson 3-2 Data Types

[Lesson 3-2.1 Data Types](#)



Data Types in This Lesson

- Character strings
- Numeric
- Datetimes
- Factors

As you've probably noticed, it's imperative that all values within a column of a data frame are all the same type. There are a variety of data types such as character strings, booleans, factors, numeric, integers, and dates. In this lesson, we'll talk about four different data types; character strings, numeric, dates, and factors. Let's start with character strings. Character strings are simply text characters that are strung together. In the English language, strings are most often made up of one or more alphabetic characters. However, digits can also be represented as a string. For instance, the word income is a string, and so is income statement for the year ended 12/31/2020 even though it has numeric characters in it.

The screenshot shows a Microsoft Excel spreadsheet. In cell A34, the text "Income Statement" is followed by a formula bar showing "=CONCATENATE(A34, " ",B34)". In cell B34, the text "Year Ended 12/31/2020" is displayed. The formula bar also shows the expanded formula: "=CONCATENATE(text1, [text2], [text3], [text4],)". The formula bar has a dropdown arrow icon.

Character strings are often manipulated. For instance, they're often combined to make larger character strings. You may have done this in Excel using the concatenate function. Other times, character strings are broken apart or are parsed in a number of ways. Ideally a delimiter like the pipe symbol or a comma can be used to create sub-strings.

The screenshot shows a Microsoft Excel spreadsheet with the Data tab selected in the ribbon. The ribbon also includes Formulas, Review, View, and Developer tabs. The main area displays two columns of data: column S contains names (Musk, Barra, Hackett, Ghosn, Fields, Ford Jr., Ford, Marchionne, Lutz, Zetsche, Trotman) and column T contains their corresponding titles (Elon, Mary T., Jim, Carlos, Mark, William Clay, Henry, Sergio, Bob, Dieter, Alexander). The Data tab ribbon includes options for Connections, Properties, Edit Links, Sort, Filter, Advanced, Text to Columns, Flash Fill, Remove Duplicates, Data Validation, Consolidate, What-if Analysis, Group, Ungroup, and Subtotal.

You may have used other functions in Excel to extract the first or the last characters or to extract certain patterns or sub-strings. Now let's talk about numeric types. You're already

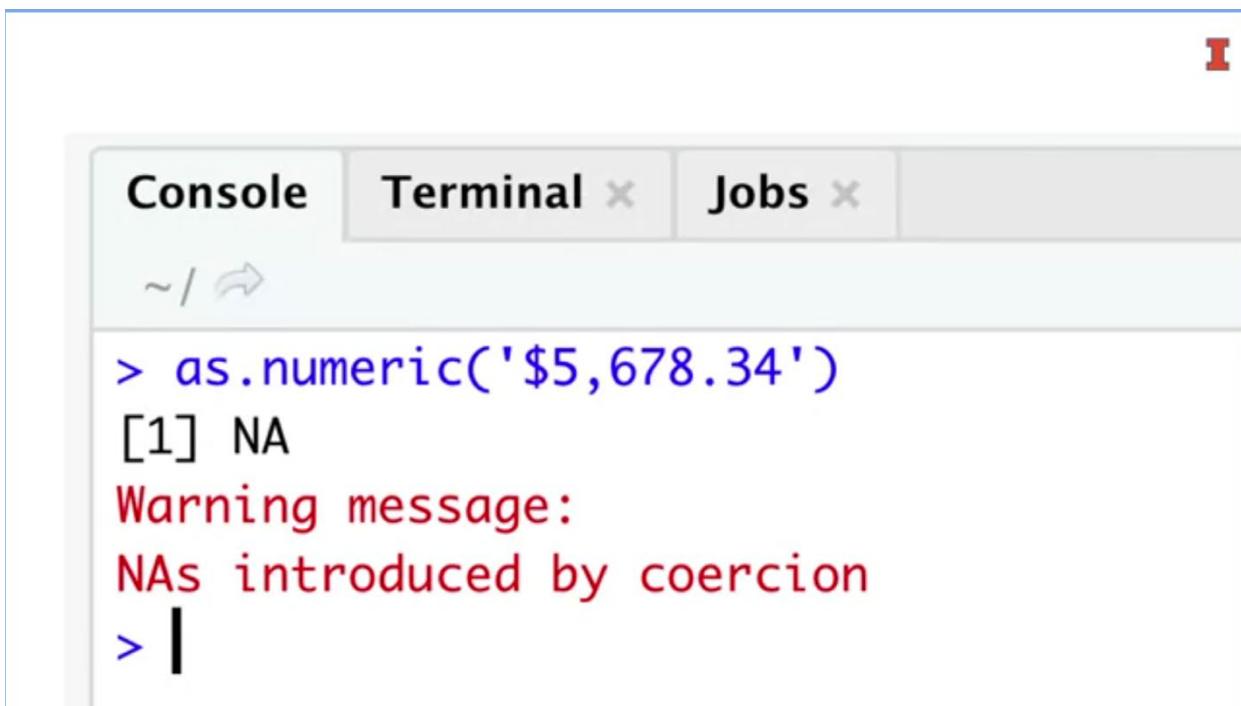
familiar with this data type. This is probably the most important data type because all data science is based on numeric data, including image detection and natural language processing.



Data Types in This Lesson

- Numeric

Essentially, all data needs to be converted to a numeric value before it can be used in an algorithm. Business data obviously has lots of numbers in it. However, one issue that often occurs is that numeric data can be read in as character strings.



The screenshot shows an R console interface. The title bar has tabs for "Console", "Terminal", and "Jobs". The main area of the console shows the following R session:

```
> as.numeric('$5,678.34')
[1] NA
Warning message:
NAs introduced by coercion
> |
```

The output shows that the function `as.numeric` failed to convert the string '\$5,678.34' into a numeric value, resulting in NA. A warning message is also displayed, stating 'NAs introduced by coercion'. The cursor is positioned at the end of the command line.

This is because we often display numeric data in the form of currency, which includes a currency sign as well as commas after every three digits. This obviously makes the data easy for humans to read and process. However, machines can't perform mathematical functions until those strings and commas are removed and they're converted to a numeric type.



I

Data Types in This Lesson

- Character strings
- Numeric
- **Datetimes**
- Factors

The third data type that I want to discuss are date-time types. Dates and times are stored in a special integer format, even though they're often displayed as strings. Specifically, dates are stored as the number of days that have passed since a specific reference state or epoch. Excel uses the beginning of the 20th century as the epoch. It would encode the date of March 3rd, 2005 as 38,415 because that's how many days have passed since the 20th century began at midnight on January 1st, 1900. In R, and I believe most programming languages, the reference point or epoch is January 1st, 1970.



Dates and Times

- Stored in a special integer format that displays them as dates
- Number of days that have passed since the start of a specific epoch
- March 3, 2005 is stored as the integer 38,415 in Excel.
- Timestamps represent the number of seconds that have passed since the start of an epoch.

Timestamps are recorded in a similar way as dates, except that they are based on the number of minutes, seconds, or milliseconds that have passed with respect to the epoch. At any rate, if you want to be able to do any calculations with dates, like calculate how many days are between them, or what the date will be in x number of days, then you'll need to make sure that they're stored in a date format. Another related issue with dates is that they can be displayed and recorded using a variety of patterns.



What does 03/04/05 represent?

March 4, 2005

April 3, 2005

April 5, 2003

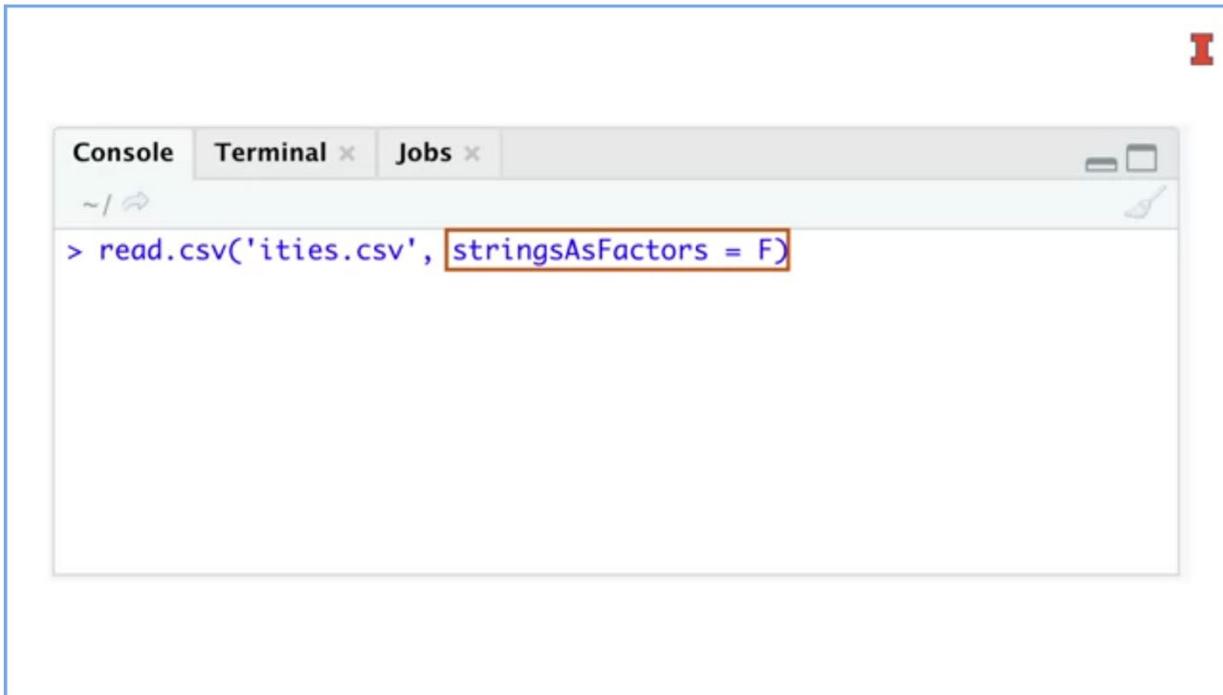
For instance, 03, 04, 05 could be referring to at least three different dates. March 4th, 2005, April 3rd, 2005, or April 5th, 2003 among others. As long as dates are recorded using a consistent pattern, then you can use built-in functions that quickly and easily convert date strings to a date-time format. Problems often arise, however, when merging data sets that use two different date patterns. When this happens, it takes creativity to find an elegant way to convert all the dates to a date type. Dates and times have many other issues. For instance, it gets more complex when you have to do with time zones, daylight savings time, and leap year.

Additional Considerations for Dates and Times

I

- Time zones
- Daylight savings time
- Leap year
- Number of work days in a quarter
- Weekends

The months in the Gregorian calendar often make things tricky as well, because the number of days changes per month. For example, consider calculations relating to earnings per workday for each quarter during the year 2019. The first quarter has 90 days, the second has 91, and the last two have 92. Then you have to factor in weekends and holidays. Dates and times are complex. The last data type to discuss in this lesson is the factor type. Remember the strings as factors option when reading in data from an external file. That's the datatype that I'm talking about now, when I talked about numeric data some minutes ago, I mentioned that all data needs to be converted to a numeric value for data science algorithms. Since R is intended primarily for statistical analysis, it aggressively strives to convert data to a format that can be used in these algorithms. Thus, when our encounters a categorical variable like gender, country, or hemisphere, the default is to convert it to a special numeric type that can be used in the algorithms. How is a factor type different than a string type and a numeric type?



Well, it's like a date-time format in the sense that factors are stored as numeric types but they're displayed as a string. If a vector of strings has two categories, like male and female, then it assigns an integer to each category based on the order in which they appear in the data. For instance, if male were to appear before female than male would be assigned the number 1 and female would be assigned the number 2. But it would display the character strings of male and female, or M and F rather than the numbers. R would also report that the vector is a factor datatype with two levels. You can see then that the number of levels tells you the number of different categories in a column of data. In conclusion, datatype is often different from the way it's displayed. It's important to understand the different data types so that you can process it in a meaningful way. You already know how to deal with numeric data types. We'll spend separate lessons on dates, factors, and strings, and how to deal with them in R

Lesson 3-3 More on Functions

Lesson 3-3.1 More on Functions



Whether you're doing with data types, reading, and data, or some other analytic process. A fundamental principle when using a data analytic language and really programming, in general, is that you should not repeat your code. This concept is often known as DRY, which is an acronym, which stands for Don't Repeat Yourself. The opposite of dry code is WET code, WET is also an acronym, which stands for Write Every Time.

WET Code Is Inefficient

- It takes extra time to write everything out
- It takes extra time to maintain



Wet code is inefficient, first of all, when you create the code, it takes a lot of time to manually type out or even copy and paste the code over and over again. The second reason why it's inefficient is because if there's a problem in your code, we have to update it for some reason in that process. Then you have to go and find every time that you've used that process and update it or fix it.

A screenshot of an R help browser window. The title bar says "d 3: How can I create and use functions to help with data prep? / rFiles/". The main content area shows the help page for the `log` function. The text includes:

```
R: Logarithms and Exponentials
log1p(x) computes log(1+x) accurately also for |x| << 1.
exp computes the exponential function.
expm1(x) computes exp(x) - 1 accurately also for |x| << 1.
```

The `log` function is highlighted with a blue box:

```
log(x, base = exp(1))
```

Below the function signature, the help text continues:

```
log(x)
log2(x)

log1p(x)

exp(x)
expm1(x)

Arguments
x      a numeric or complex vector;
base   a positive or complex number: the base with respect to
       which logarithms are computed. Defaults to
       e=exp(1).
```

At the bottom of the page, there are sections for "Details" and a note: "All except logb are generic functions: methods can be defined."

In this lesson, I want to talk at a deeper level about functions, functions you should already be a little bit familiar with, but I want to talk about the arguments in a function and how those can vary, and then even create a function of our own. Let's first read the help documentation for the log function. As you know, in the help documentation, there's a section that first described intuitively what this function can do and in this case, it gives variations of this function. Then there's a section on usage and that shows how this function is used and lets for a minute pause and look at this first example here it says log and then in parentheses, the first argument is x, the second argument is the base and it says the default value for the base is Euler's constant, basically E.

d 3: How can I create and use functions to help with data prep?/rElles/
ULTS #####

R: Logarithms and Exponentials - End of Topic

log(x)

Arguments

x a numeric or complex vector.

base a positive or complex number: the base with respect to which logarithms are computed. Defaults to $e=\exp(1)$.

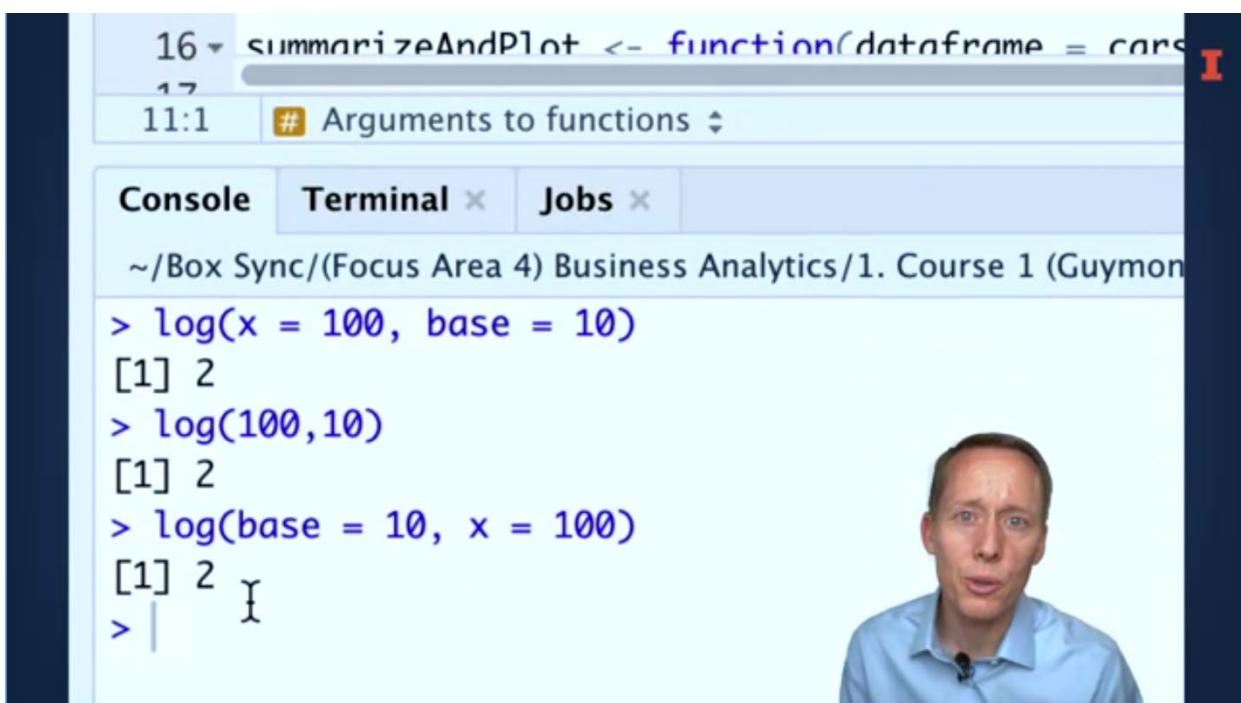
for $\log_b(x)$ individually or via the `Math` group generic.

`log10` and `log2` are only convenience wrappers, but logs to bases 10 and 2 (whether computed via `log` or the wrappers) will be computed more efficiently and accurately where supported by the OS. Methods can be set for them individually (and otherwise methods for `log` will be used).

`logb` is a wrapper for `log` for compatibility with S. If (S3 or S4) methods are set for `log` they will be dispatched. Do not set S4 methods on `logb` itself.

All except `logb` are primitive functions.

Now if you use other versions of this function, you don't necessarily need to identify the base because it is identified in the name of the function itself. Another key aspect of the documentation is the argument section, so in this section, it tells us more details about what those arguments can be. For instance, x is a numeric or complex vector, so it tells us the data type and then that it can be a vector. Base is a positive or complex number and then it describes that in more detail.



```
16 <- summarizeAndPlot <- function(dataframe = cars)
17
11:1 # Arguments to functions
```

Console Terminal × Jobs ×

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon)

```
> log(x = 100, base = 10)
[1] 2
> log(100,10)
[1] 2
> log(base = 10, x = 100)
[1] 2
```

Let's test out the log function, I'll type out log open parenthesis, and then we'll explicitly say that x is going to equal 100 and the base will equal 10, so in other words, 10 to what power equals 100? If I run this, it tells me the answer is two, so 10 squared raised to the second power equals 100. Now, a neat thing about functions is that you don't have to explicitly identify what the values mean if you put them in the order that the arguments are taken. I can say log of 100, 10, and that also returns the same value of two. If I do want to put them out of order or skip an argument then I can do so if I just explicitly define what the argument values are, so in this case I can say the base and want to be 10 and the x, I want to be 100 and again we get the same value too.

The video player shows a man in a blue shirt speaking. To his right is a screenshot of a computer screen displaying the R help documentation for the 'sum' function. The title 'Usage' is at the top, followed by the code snippet 'sum(..., na.rm = FALSE)'. Below the code is a note about 'na.rm': 'logical. Should missing values (including NaN) be removed?'. Under the heading 'Details', it says: 'This is a generic function: methods can be defined for it directly or via the [Summary](#) group generic. For this to work properly, the arguments ... should be unnamed, and dispatch is on the first argument.' At the bottom, there is a note: 'If na.rm is FALSE an NA or NaN value in any of the arguments will cause a value of NA or NaN to be returned, otherwise NA and NaN values are ignored.'

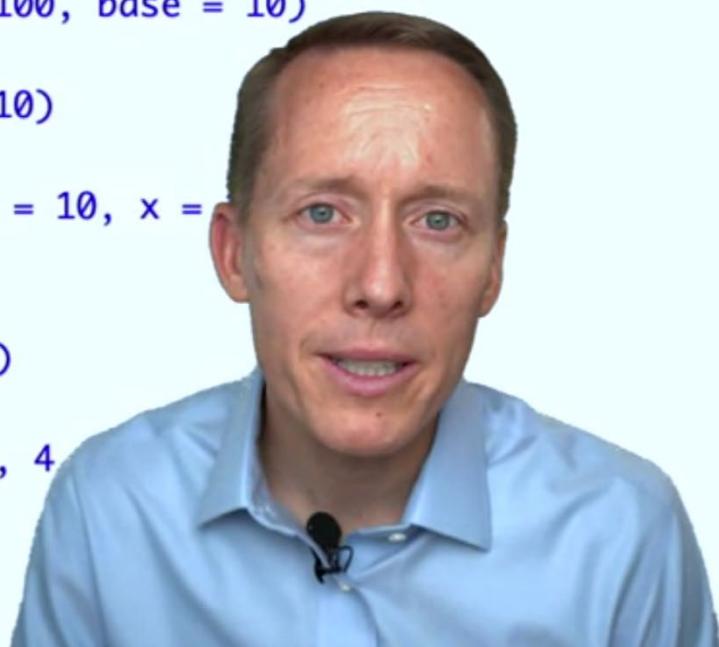
Now let's read the help documentation for the sum function. Now this function gives us an intuitive description and then it describes how it should be used and it says sum and then open close parentheses, and then it has dot-dot-dots for one argument and a.rm as a second argument. Now let's look at this dot-dot-dot, that's an ellipses. In this case, ellipses and actually in almost every programming case means there can be more than one argument here, so this needs to be numeric, complex, or logical vector. Then any dot Rm is a logical value which indicates whether missing data should be included.

```
> log(x = 100, base = 10)
[1] 2
> log(100,10)
[1] 2
> log(base = 10, x = 100)
[1] 2
> ?sum
> sum(1, 3)
[1] 4
> sum(1, 3, 4, 6, 7)
[1] 21
>
```

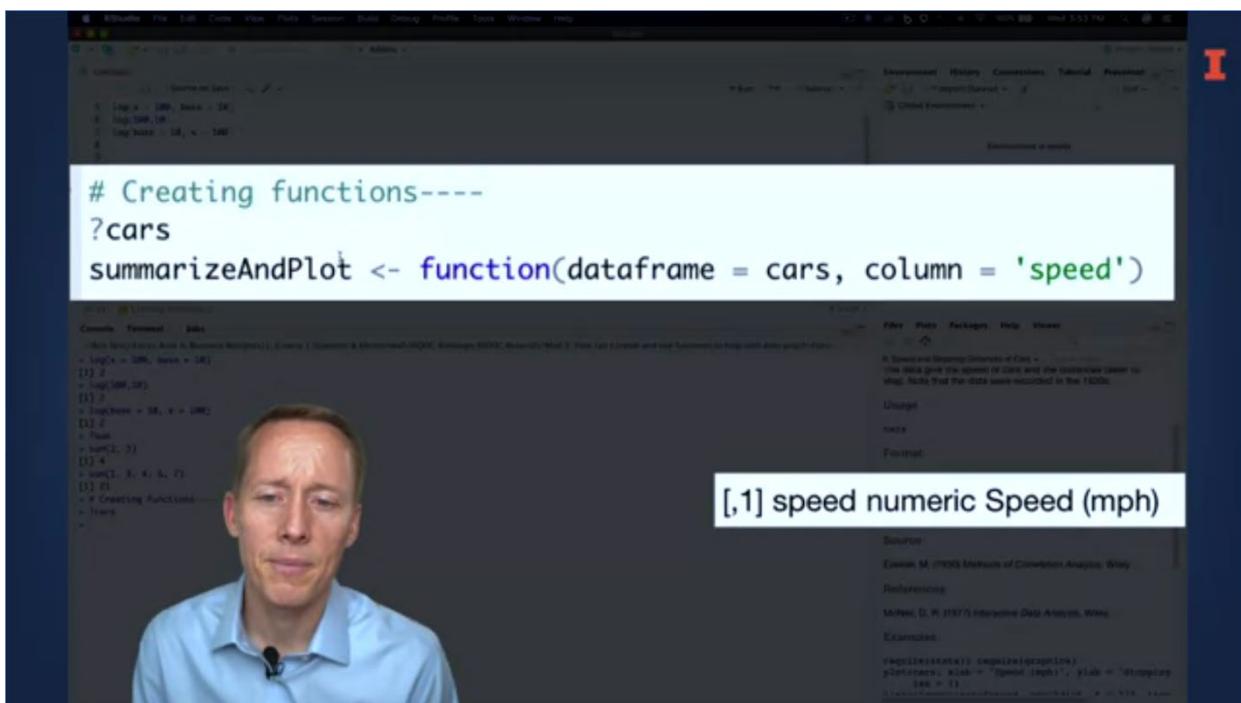


For instance, I can use the sum function with just one and three and I get four, or I can add in four again, six, seven add in a lot of other numbers and it still works, so those ellipses just indicate you can use more than one variable. Now, that word I just said, variable is important with Functions and that's the whole reason why Functions are useful. They take a process, perform the same steps, but they use different values for the arguments, that's why functions make code DRY, I don't have to go in and do whatever processes is being taken to create the sum and the log. Now we can create our own functions.

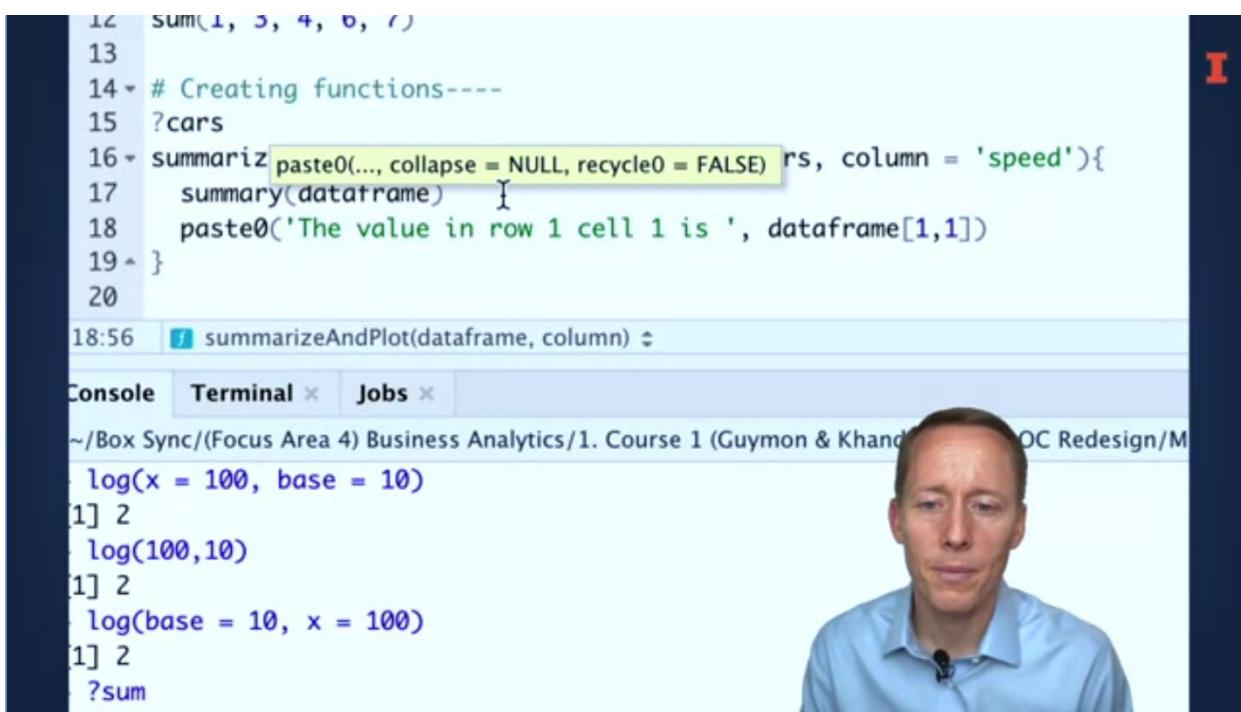
```
> log(x = 100, base = 10)
[1] 2
> log(100,10)
[1] 2
> log(base = 10, x = 100)
[1] 2
> ?sum
> sum(1, 3)
[1] 4
> sum(1, 3, 4)
[1] 10
> sum(1, 3, 4, 6, 7)
[1] 21
>
```



Let's create our own function that has a data analytic purpose for whatever reason, let's say that with a lot of different datasets, look at the descriptive statistics for each column, print out a value for two different cells within the data frame, and then create a histogram for a certain column of the data frame that may vary each time I use that function. That's a process I often go through and rather than typing out each of those steps every time, I can package them together into one function. I'm going to use this to start with on the built-in cars dataset. If you want to see what that dataset is, you can type out, "Question mark cars" and in the Help pane it tells us a description of this data, it's recorded from the 1920s, and there are two columns, speed, and distance, which is the stopping distance.



Now let's go ahead and create the function, let's call this the summarize and plot function and to indicate that it's a function, I need to type out the word function and then I'll type parenthesis and within these parentheses, I will indicate what the arguments are. The first argument, in this case, will be data frame, I want this to be generalizable to any data frame and I'm going to say that the default value will be the cars data frame. The other argument is going to be the column, so I will type out column and I will set it to be the default value of speed.



```

12 sum(1, 3, 4, 5, 7)
13
14 # Creating functions----
15 ?cars
16 summarize(paste0(..., collapse = NULL, recycle0 = FALSE) ~ rs, column = 'speed'){
17   summary(dataframe) }
18   paste0('The value in row 1 cell 1 is ', dataframe[1,1])
19 }
20
18:56 # summarizeAndPlot(dataframe, column) <-

```

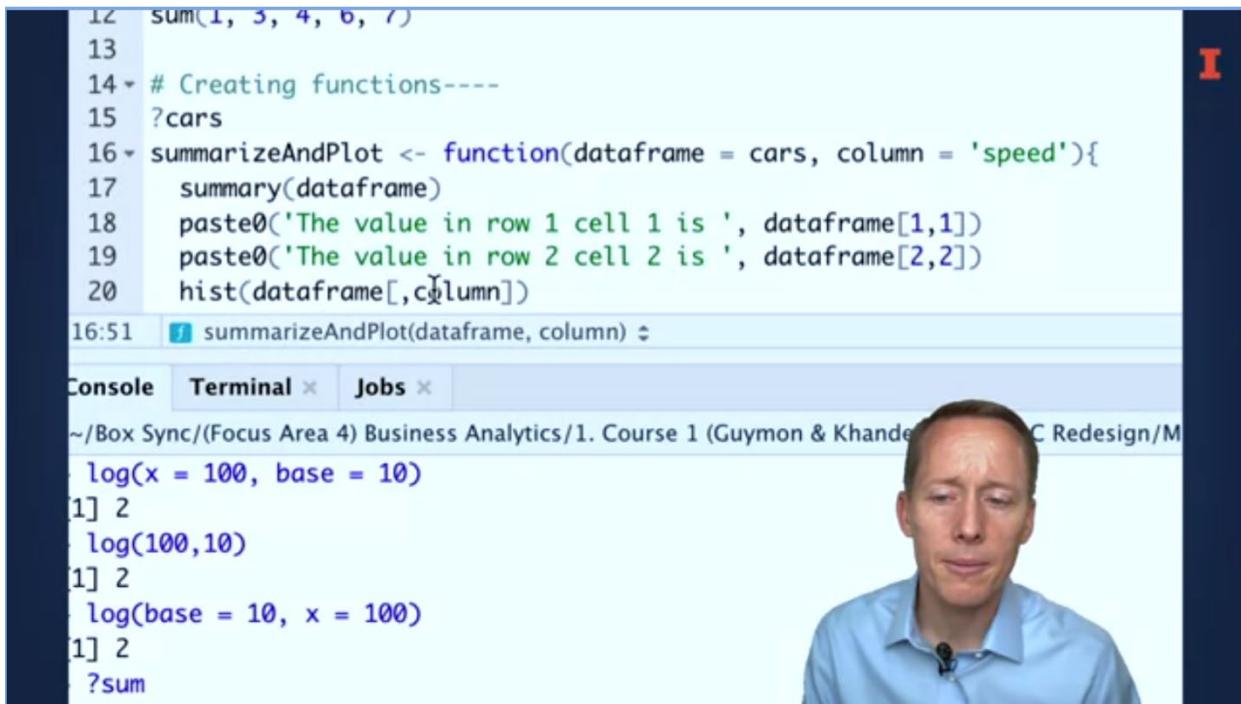
Console Terminal × Jobs ×

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandani) - DOC Redesign/M
log(x = 100, base = 10)
[1] 2
log(100,10)
[1] 2
log(base = 10, x = 100)
[1] 2
?sum

```

At this point, I need to now indicate what steps I want to be taken when I run this function. I'll type out curly braces, and everything that comes in-between these curly braces, will be performed by the function. The first thing I want this function to do, is to calculate descriptive statistics for every column. Notice that I'm typing out `dataframe` there because it's not always going to be the curly's `dataframe`.



```

12 sum(1, 3, 4, 5, 7)
13
14 # Creating functions----
15 ?cars
16 summarizeAndPlot <- function(dataframe = cars, column = 'speed'){
17   summary(dataframe)
18   paste0('The value in row 1 cell 1 is ', dataframe[1,1])
19   paste0('The value in row 2 cell 2 is ', dataframe[2,2])
20   hist(dataframe[,column])
16:51 # summarizeAndPlot(dataframe, column) <-

```

Console Terminal × Jobs ×

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandani) - DOC Redesign/M
log(x = 100, base = 10)
[1] 2
log(100,10)
[1] 2
log(base = 10, x = 100)
[1] 2
?sum

```

I'm going to use the summary function on whatever the dataframe is that I input into the function. The next thing I want to do, is to return a value from a cell on the dataframe. I will indicate this by saying, paste the value in row one, cell one is, and I'll say dataframe and then [1,1] it's taking first row, first column. Let's say I want to do this for another cell in the dataframe. I'll say for row two cell two. I'll change that to two and two. Now I am copying and pasting here. There is a more efficient way to do this, but for our purposes, this will be fine. Finally, I want to create a histogram of a certain column. I'll say hist, and then I'll indicate the DataFrame, and I want to use all the rows, so I'll just put a comma to indicate will take all the rows, and then for the column, I will say, I'm going to take whatever value has been assigned to this column variable here, and use that, so that will be my function. I can create this now, and I have this new object in the global environment under the function section. Let's go ahead and run it.



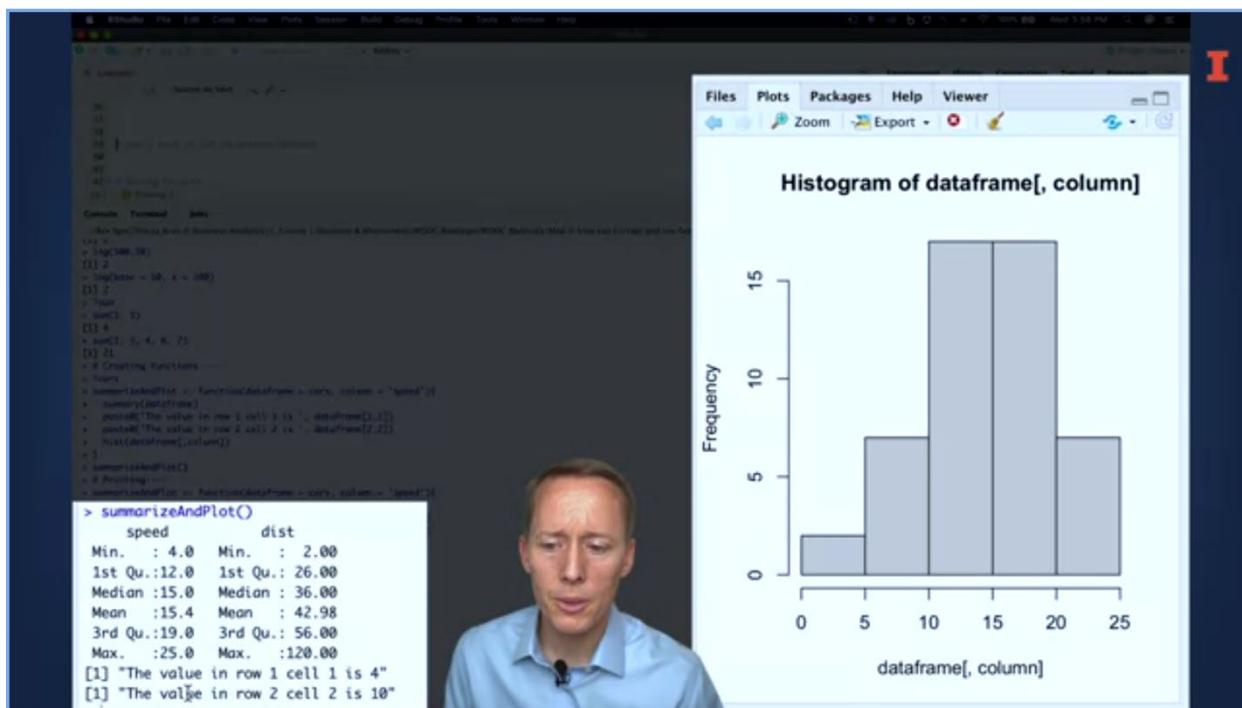
Notice that it printed out a histogram of the speed column there, but it did not print out the text that I wanted it to print out the descriptive statistics, and the values of those two cells. That's because it didn't know that we wanted to print those values out to the console. We need to explicitly indicate that these values should be printed.



```
R Untitled1* Source on Save
25 # Printing----
26 summarizeAndPlot <- function(dataframe = cars, column = 'speed')
27   print(summary(dataframe))
28   print(paste0('The value in row 1 cell 1 is ', dataframe[1,1]))
29   print(paste0('The value in row 2 cell 2 is ', dataframe[2,2]))
30   hist(dataframe[,column])
31 }
26:6 # summarizeAndPlot(dataframe, column) ↴

Console Terminal × Jobs ×
~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khan) /MOOC Rec
> log(x = 100, base = 10)
[1] 2
> log(100,10)
[1] 2
> log(base = 10, x = 100)
```

Let's update our function, by indicating that we do want all of these values to be printed. I will copy and paste this function, and I will wrap each of these first three processes of the function within the print function. I'll go ahead and create this, and then I will run this function.



This time, it did print out descriptive statistics for both columns as well as the values in those two cells, and the histogram is there again. Let's test this out on another dataset. You can see in my working directory here, that I've got the Jan 17 items.CSV file. Let's go ahead and read that in as j17i, and let's make sure we know what the columns are in this DataFrame.



The screenshot shows an RStudio environment. The code editor window (Untitled1*) contains the following R code:

```
37
38
39 # Let's test it out on another dataset
40 j17i <- read.csv("Jan 17 items.CSV")
41
42 summarizeAndPlot(j17i, 'Tax')
43
```

The line `summarizeAndPlot(j17i, 'Tax')` is highlighted with a yellow box. The console window below shows:

```
42:28 # Printing
Console Terminal Jobs
~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandani) Redesign/MO
L+J <
> log(base = 10, x = 100)
[1] 2
> ?sum
> sum(1, 3)
[1] 4
> sum(1, 3, 4, 6, 7)
[1] 21
```

There's all the columns. Let's go ahead and run the summarize and plot function on the j17i DataFrame, and let's use the tax column. You can see that it has printed out descriptive statistics for each column there, as well as the values in those two cells that we indicated, and the histogram of the tax column. We created a function that allows us to perform all those steps with basically a single line of code. You can take the output of a function, and save it as a new object. In this case, I'm going to save it to the A object.

```
> a
$breaks
[1] -40 -20   0  20  40  60  80 100 120 140 160 180 200 220 240 260 280

$counts
[1] 2   19 8520   11    3    1    0    0    1    0    0    0    0    0    0    0    1

$density
[1] 1.168497e-05 1.110072e-04 4.977799e-02 6.426735e-05 1.752746e-05 5.842487e-06 0.000000e+00
[11] 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 5.842487e-06

$mid
[1] -30 -10   10  30  50  70  90 110 130 150 170 190 210 230 250 270

$xname
[1] "dataframe[, column]"

$equidist
[1] TRUE

attr("class")
[1] "histogram"
>
```



Let's go ahead and take a look at what is in that A object. It's not very clear here, but these are basically the elements that are used to create the histogram. Now, rather than returning the elements that make up the histogram, let's say that I want to return the second sentence there that tells us what's in row two, column two.



```
Untitled1* 
Source on Save | 
48 * # Returning results----
49 * summarizeAndPlot <- function(dataframe = cars, column = 'speed'){
50   print(summary(dataframe))
51   print(paste0('The value in row 1 cell 1 is ', dataframe[1,1]))
52   return(print(paste0('The value in row 2 cell 2 is ', dataframe[2,2])))
53   hist(dataframe[,column])
54 * }
52:73  summarizAndPlot(dataframe, column) <

Console Terminal Jobs 
~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Materials/MOOC Materials
```

Quantity	Modifiers	Subtotal	Discount	NetTotal
Min. : 1.000	Min. :-2.5200	Min. :-322.80	Min. :-3	Min. :-3
1st Qu.: 1.000	1st Qu.: 0.0100	1st Qu.: 5.24	1st Qu.: 5.24	1st Qu.: 5.24
Median : 1.000	Median : 0.0100	Median : 12.12	Median : 12.12	Median : 12.12
Mean : 1.000	Mean : 0.0100	Mean : 12.12	Mean : 12.12	Mean : 12.12
3rd Qu.: 1.000	3rd Qu.: 0.0100	3rd Qu.: 5.24	3rd Qu.: 5.24	3rd Qu.: 5.24
Max. : 1.000	Max. : 0.0100	Max. : 12.12	Max. : 12.12	Max. : 12.12

I'll copy and paste that function, and then I can wrap whatever I want to be saved within a function, which is the return function. I will recreate this, summarize and plot function, and I'll rerun it on the j17i.

Quantity	Modifiers	Subtotal
Min. : 1.000	Min. :-2.5200	Min. :-322.80
1st Qu.: 1.000	1st Qu.: 0.0100	1st Qu.: 5.24
Median : 1.000	Median : 0.0100	Median : 13.10
Mean : 1.177	Mean : 0.6548	Mean : 15.10
3rd Qu.: 1.000	3rd Qu.: 1.0800	3rd Qu.: 15.76
Max. : 36.000	Max. : 57.5500	Max. : 3328.12

```
[1] "The value in row 1 cell 1 is 2017-01-26T21:18:  

[1] "The value in row 2 cell 2 is SALE"  

> a  

[1] "The value in row 2 cell 2 is SALE"  

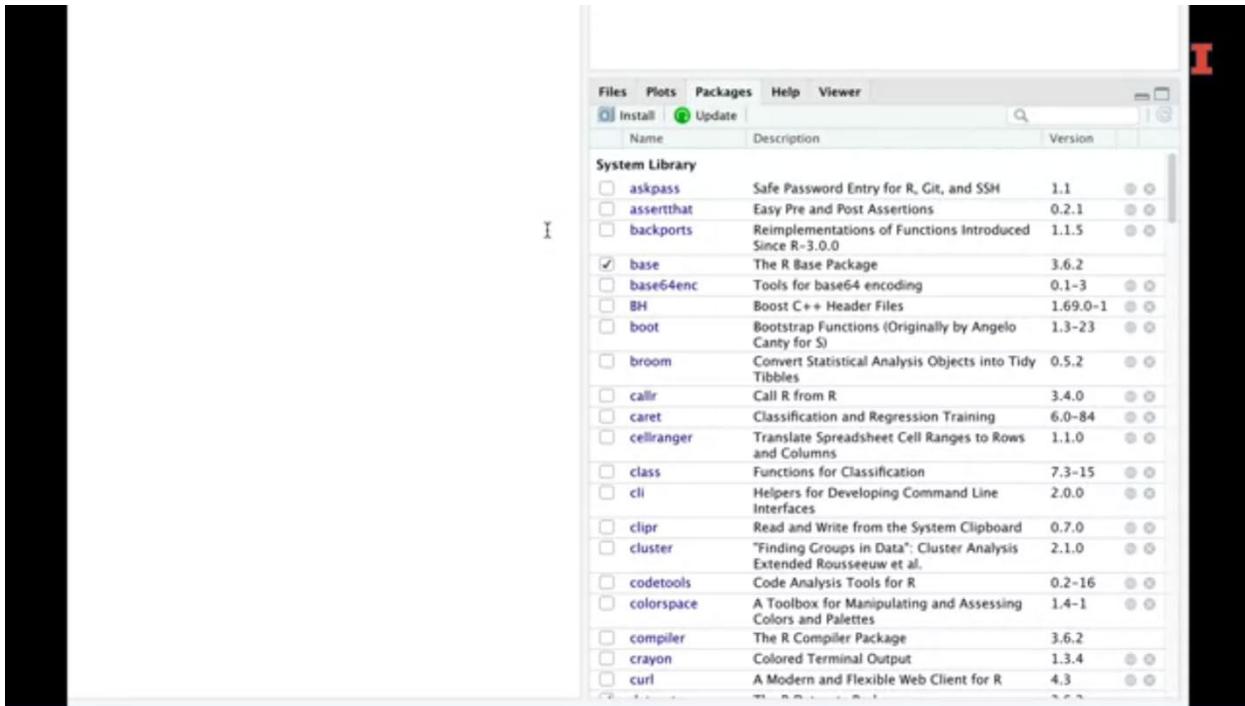
>
```



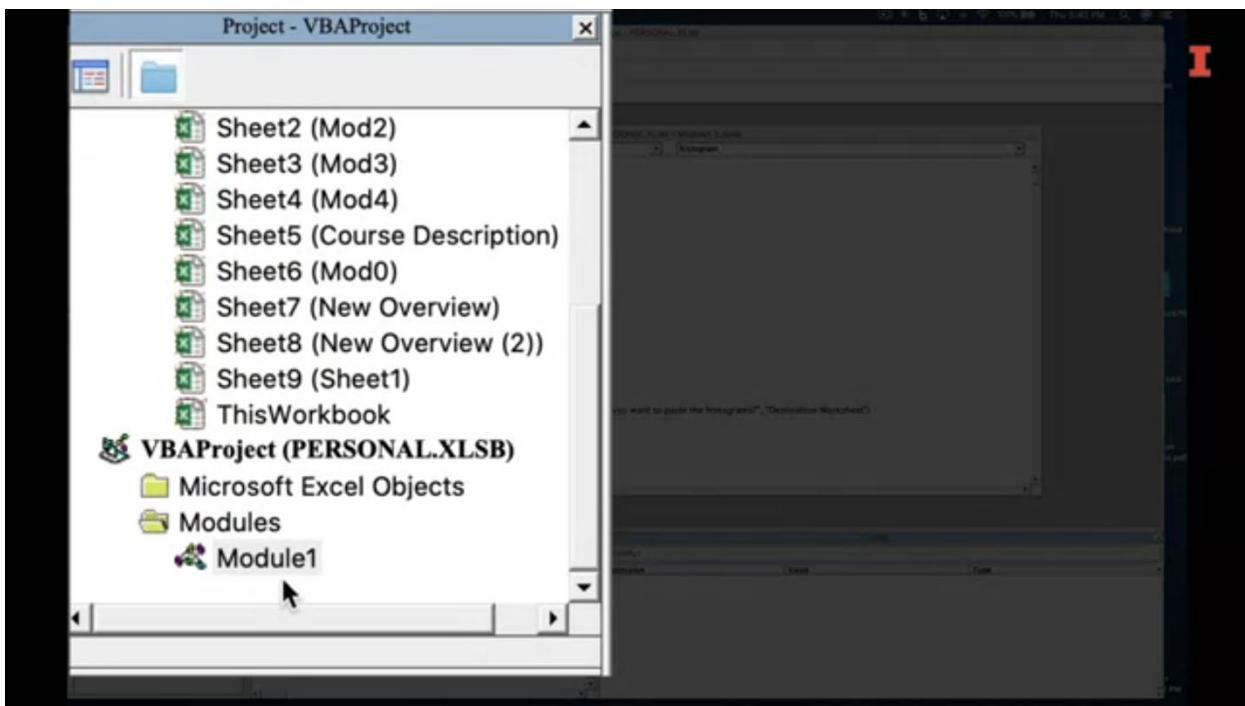
Just to mix things up, rather than use the tax column, let's use the subtotal column. You can see that the histogram, you can't actually tell, because it looks very similar, but the numbers are different here, and if I run now A, see what's stored in A, you can see that it's the second sentence here. The value in row two, cell two is sale. Now you can return more than one object from a function, and the way you do that is by creating lists, but that's beyond the scope of this lesson here. There's a little bit deeper dive into functions. I hope that make sense how they can be useful for creating dry code.

Lesson 3-4 Packages

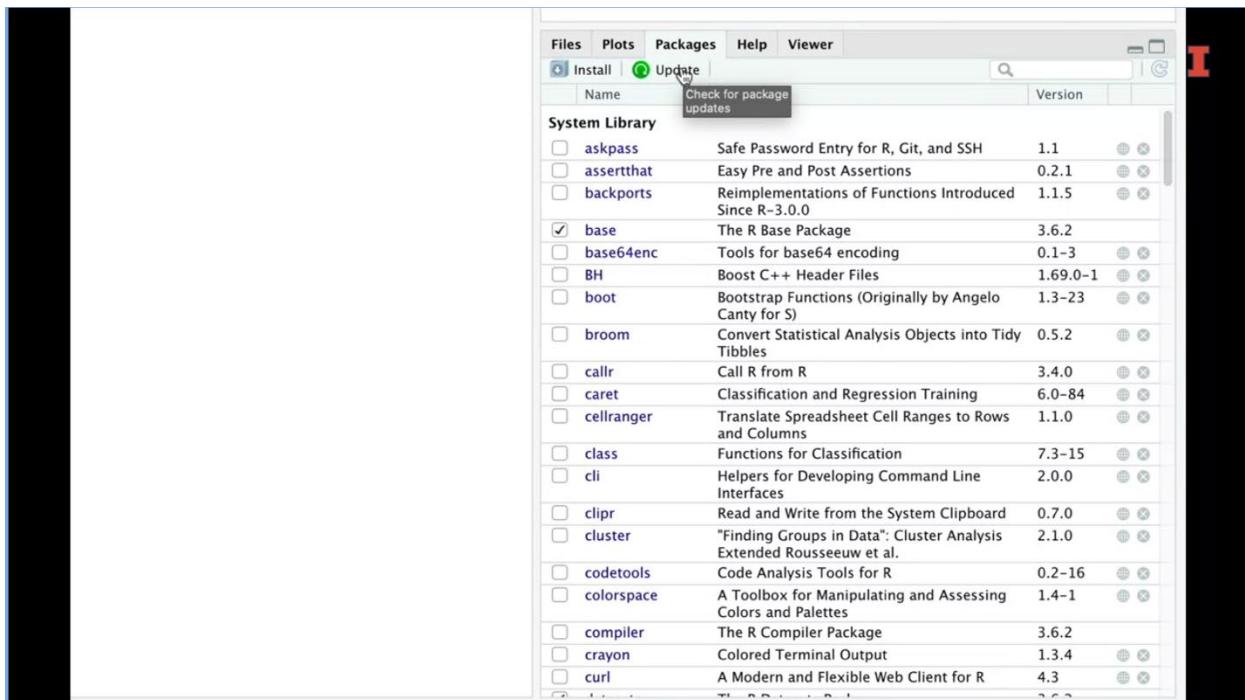
Lesson 3-4.1 Packages



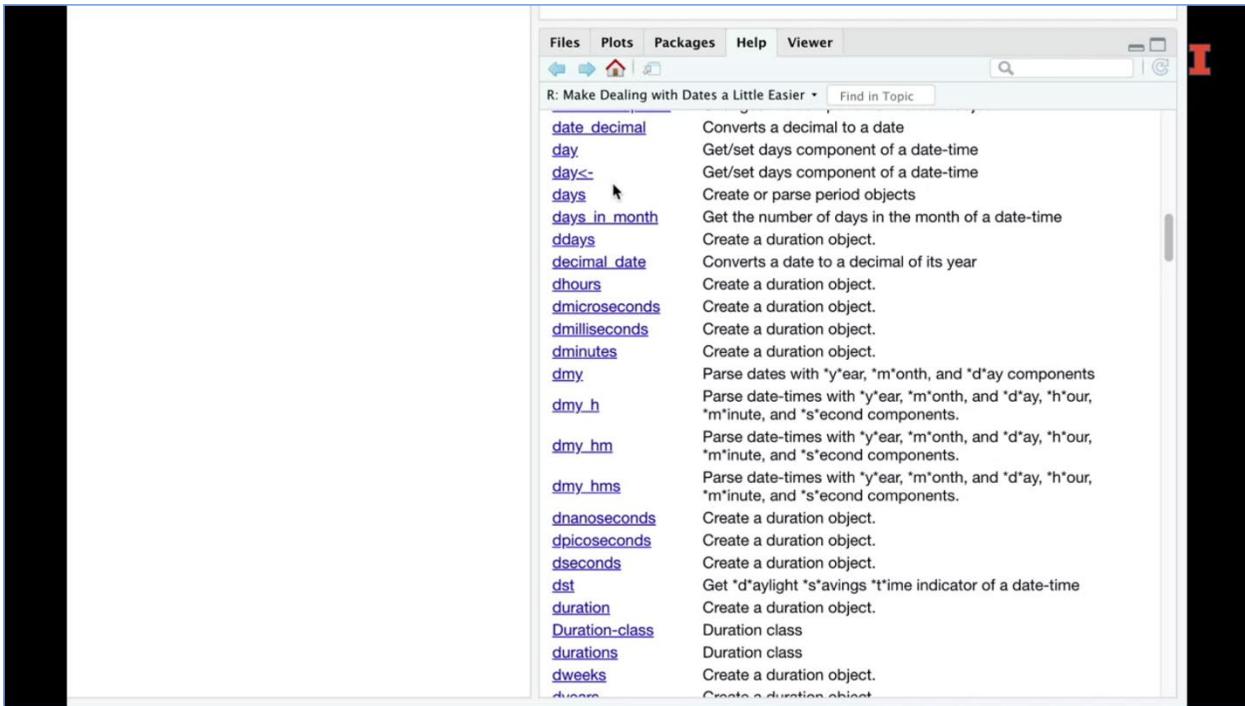
This section of the course is about data types. But before jumping into different data types and learning how to deal with those, it's worth your while to learn how to manage packages. Because doing so will make it much easier for you to complete some common tasks associated with data types, as well as many other aspects of data analysis. Now, as the name implies, a package you can think of really as a gift that someone has created for you.



And within that package, there is a list of functions that perform some related tasks. In relation to Excel, if you've ever created a macro that has saved you a lot of time, it really can feel like a gift. And if you've ever stepped into the code for that macro and looked at it in the visual basic editor, you may notice that in the project pane, there is a folder called Modules, and then a variety of scripts.



And in these scripts is the code for the different macros, so that same idea applies to packages in R. Let's talk about how to use this packages' pane. You'll see at the top that there are tools to install, update, and search for packages. And then there's a list of packages by name, a description, the version, and a couple of other things. R comes with some packages that are pre-loaded in it. And then you can install more packages based on what tasks you need to perform. If you want to learn more about the packages that are already installed in your project or on your machine, you can simply either scroll down to a certain package. Or search for a package by name or by function.



So there's a text or I can search for date, we're going to be talking about some date functions in a little while here. And I could click on the name of that package, so I'll click on lubridate. And here's some documentation for that, and it starts by listing all the different functions within that package. And then if I want detailed help with a certain function, I can go ahead and just click on the name of the function, and I get all the help associated with that function. I'm going to go out of the help pane back into the Packages pane and let's talk about these icons on the far right.

The screenshot shows the official documentation for the `lubridate` package, which is part of the tidyverse. The page includes sections for Overview, Installation (with R code examples), Cheatsheet (linking to a full-page cheat sheet), and various links for Reference, News, and developer information like License, Community, Citation, Developers, and Dev status.

The furthest to the right allows you to remove a package if you don't want it anymore. This one next to it tells you where you can get additional help related to packages. So if I click on that icon, it will open up in a browser a website that has additional documentation. And some documentation is better than others, this one happens to have a cheat sheet associated with it. Anyway, there's another way that you can get help with learning how to use the functions in that package.

The screenshot shows the main page of the CRAN (Comprehensive R Archive Network). It features a sidebar with links for CRAN Mirrors, What's new?, Task Views, Search, About R, R Homepage, The R Journal, Software, R Sources, R Binaries, Packages, Documentation, Manuals, FAQs, and Contributed. The main content area provides information on Download and Install R, source code availability, and submission guidelines for packages.

What if you want to perform a data analytic task, but you don't know how to do it and you'd like some help. And you want to see if someone else has made a package for you? I'd like to show you a couple of websites to search for packages to perform a task. The first website is the r-project.org website. This is where we went to install R in the first place. So if I follow the same path initially to install R, go to CRAN, then click the 0-Cloud server. And now, instead of downloading R, I'm going to go to the left, sidebar here and click on Packages. On this page, it tells me about available packages and how to install packages.

CRAN Task View: Empirical Finance

Maintainer: Dirk Eddelbuettel
Contact: Dirk.Eddelbuettel at R-project.org
Version: 2019-03-07
URL: <https://CRAN.R-project.org/view=Finance>

This CRAN Task View contains a list of packages useful for empirical work in Finance, grouped by topic.

Besides these packages, a very wide variety of functions suitable for empirical work in Finance is provided by both the basic R system (and its set of recommended core packages) on the Comprehensive R Archive Network (CRAN). Consequently, several of the other CRAN Task Views may contain suitable packages, in particular the [Econometrics](#), [Optimization](#), [Robust](#), [SocialSciences](#) and [TimeSeries](#) Task Views.

The `ctv` package supports these Task Views. Its functions `install.views` and `update.views` allow, respectively, installation or update of packages from a given Task View or restrict operations to packages labeled as `core` below.

Contributions are always welcome, and encouraged. Since the start of this CRAN task view in April 2005, most contributions have arrived as email suggestions. The source file now also reside in a GitHub repository (see below) so that pull requests are also possible.

Standard regression models

- A detailed overview of the available regression methodologies is provided by the [Econometrics](#) task view. This is complemented by the [Robust](#) task view, which focuses on robust methods.
- Linear models such as ordinary least squares (OLS) can be estimated by `lm()` (from the stats package contained in the basic R distribution). Maximum Likelihood estimation can be undertaken with the standard `optim()` function. Many other suitable methods are listed in the [Optimization](#) view. Non-linear least squares can be estimated with the `nls()` and `nlsme()` from the [nlsme](#) package.
- For the linear model, a variety of regression diagnostic tests are provided by the `car`, `lmtest`, `strucchange`, `urca`, and `sandwich` packages. The `Rcmdr` and `Zelig` packages may be of interest as well.

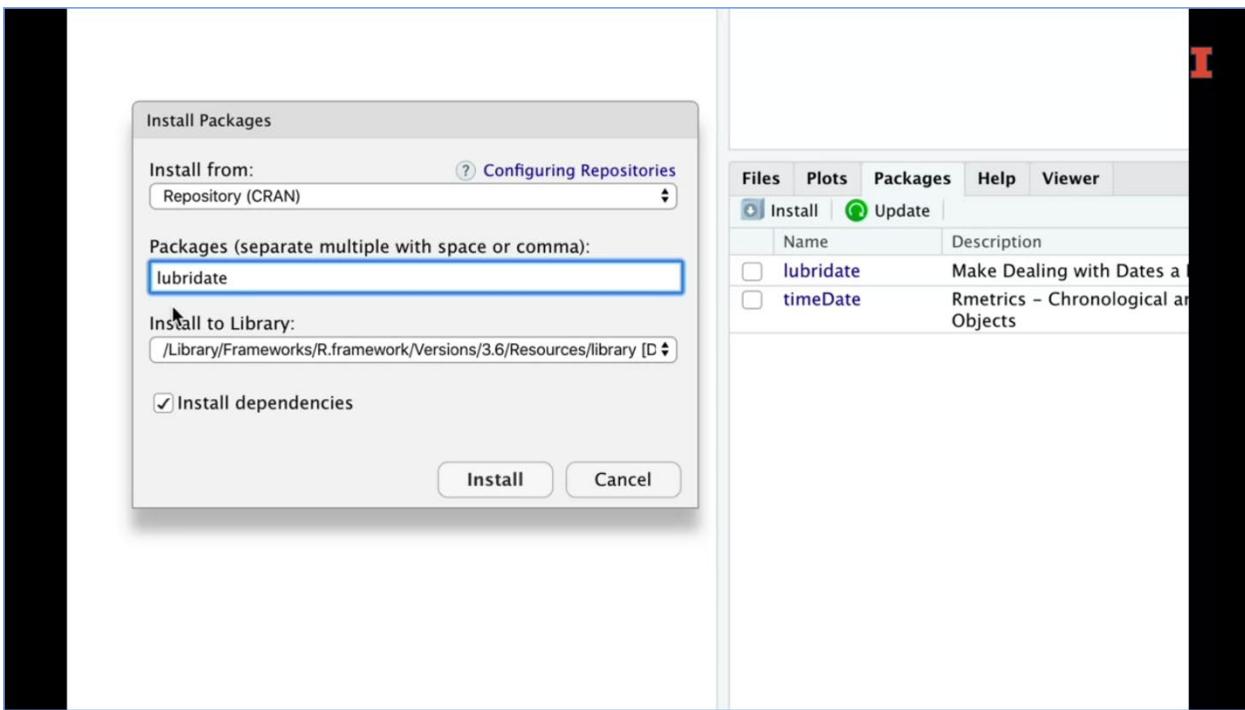
Time series

- A detailed overview of tools for time series analysis can be found in the [TimeSeries](#) task view. Below a brief overview of the most important methods in finance is given.
- Classical time series functionality is provided by the `arima()` and `KalmanLike()` commands in the basic R distribution.
- The `dse` and `timsac` packages provide a variety of more advanced estimation methods; `fracdiff` can estimate fractionally integrated series; `longmemo` covers related methods for long-memory time series modeling functionality.
- For volatility modeling, the standard GARCH(1,1) model can be estimated with the `garch()` function in the `tsseries` package. Rmetrics (see below) contains the [fGarch](#) package. The `rugarch` package can be used to model a variety of univariate GARCH models with extensions such as ARFIMA, in-mean, external regressors and various methods for fit, forecast, simulation, inference and plotting are provided too. The `rmgarch` builds on it to provide the ability to estimate several multivariate GARCH models. The `bayesGARCH` package can estimate and simulate the Beta-t-EGARCH model by Harvey. The `gets` package can perform Bayesian estimation of a GARCH(1,1) model with multivariate models. The `gogarch` package provides functions for generalized orthogonal GARCH models. The `GEVStableGarch` package can fit ARMA-GARCH or AR-GARCH models with automated general-to-specific model selection of the mean and log-volatility of a log-ARCH-X model.

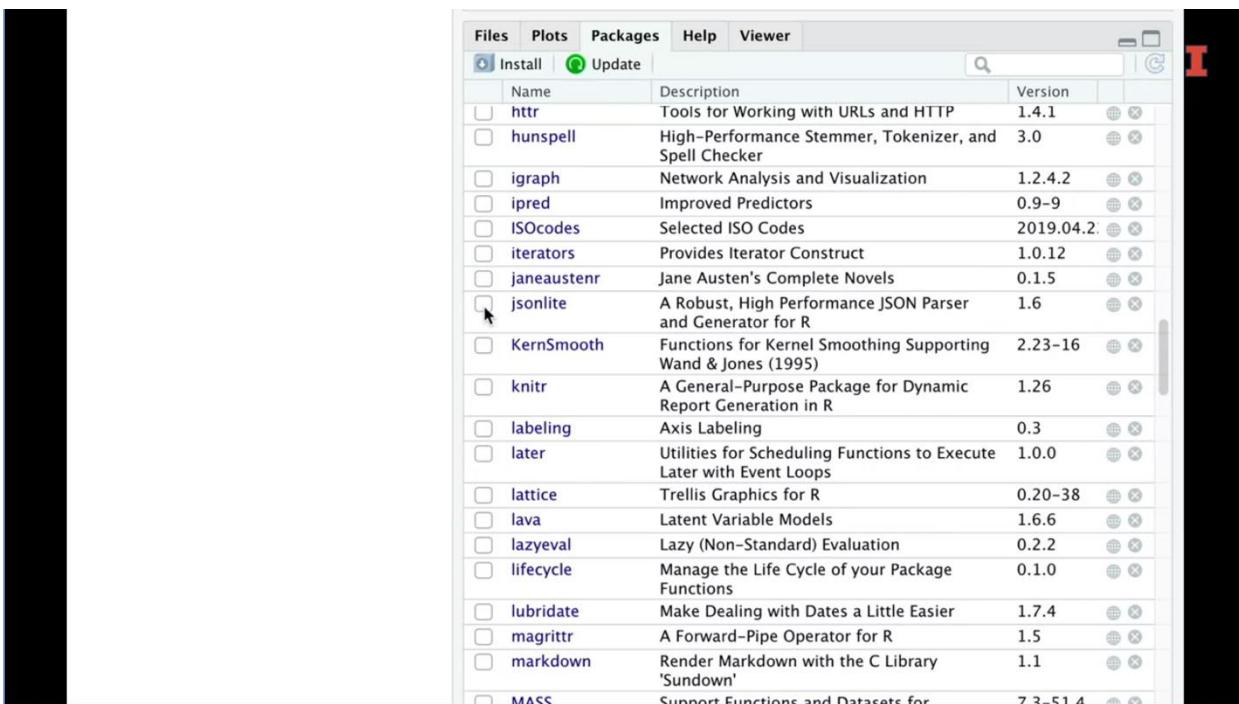
And you can see that you can search for packages by date of publication or by name. But what I find to be most helpful is to search for packages based on the functionality, which is what this link does. So if I click on that, there are about 41 different topics here, general topics. And let's say I want to learn more about finance, I can click on the finance link. And this takes me to some additional details and a link to some other packages related to financial applications, all right? So there's a whole bunch there. So that's one website.

The screenshot shows the rdocumentation.org website for the 'date' package version 1.2-39. At the top, there's a navigation bar with links for 'Enterprise Training', 'package', 'Leaderboard', and 'Sign in'. Below the navigation is a search bar and a link to the package's GitHub repository. The main content area has a header 'date v1.2-39' and a sub-header 'Functions for Handling Dates'. It lists several functions: as.date, date.mdy, date.mmdy, date.mmdyyy, date.ddmmyy, date.object, and mdy.date. Each function has a brief description and a link to its documentation. To the right, there's a 'Monthly downloads > 99.99th Percentile' badge and a 'Copy' button. A 'Last month downloads' section is shown below the functions, followed by a 'No Data Available' message. At the bottom, there's a call-to-action button 'Start Now'.

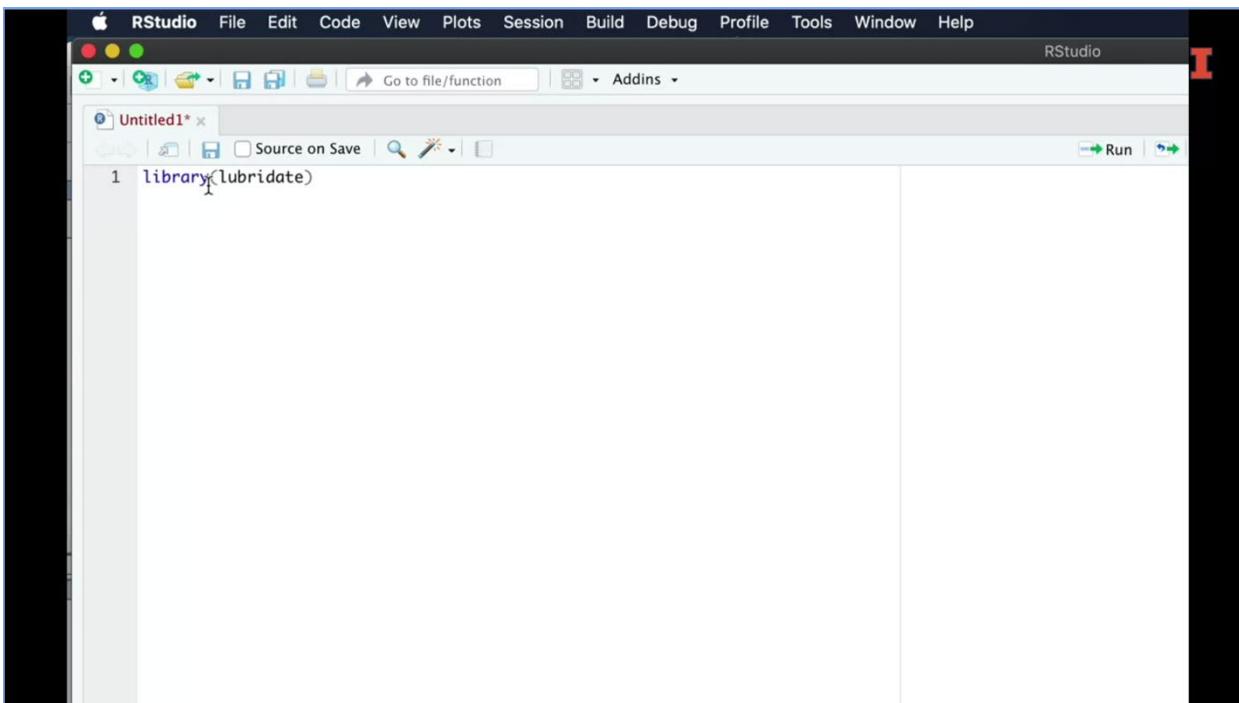
Another website that we've already looked at once before when we talked about functions is rdocumentation.org website. And if I want to search for a package to help me deal with dates, I can start typing in date. And up comes some package names as well as function names, and I can click on a package name here. And it tells me information about this package, how often it's downloaded or the percentile, so this is downloaded quite a bit. And then it tells me what the functions are in this package, as well as some additional information. And I could click on a link for one of those functions, and it gives me the documentation related to that package on this website.



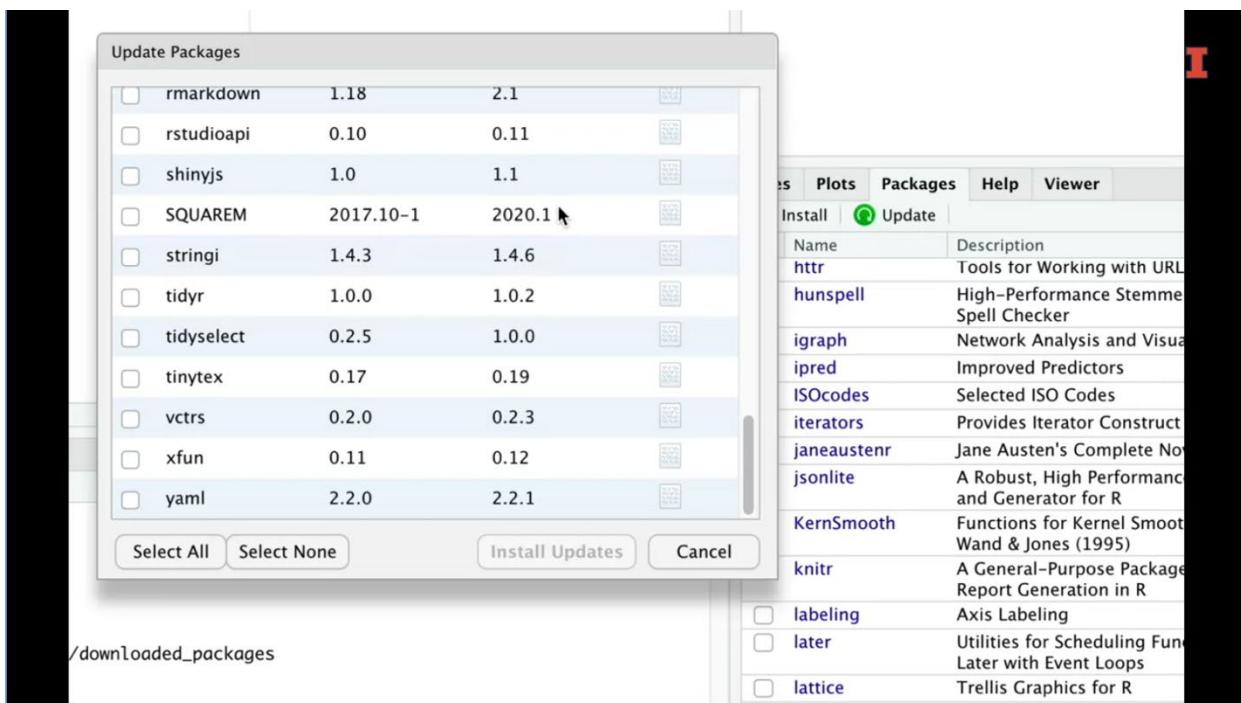
Now let's say I want to install a particular package. Let's say, for example, I know I've heard about the lubridate package and I read about it, it looks awesome and I want to install it. I could go back over to the r-project.org, and this tells me how to install packages, it's very easy to do programmatically. I could just type out `install` that package is and then type out the package name. But in most cases, what I do is I used the utilities built into R Studio. So you can click on this `Install` button in the top left hand corner of the Packages pane. And you can type the name of a package, so I could type date that I saw there and there's date. I could type lubridate, and you can see that lubridate pops up. And let's say I do want to go ahead and install it, then I just click the `Install` button.



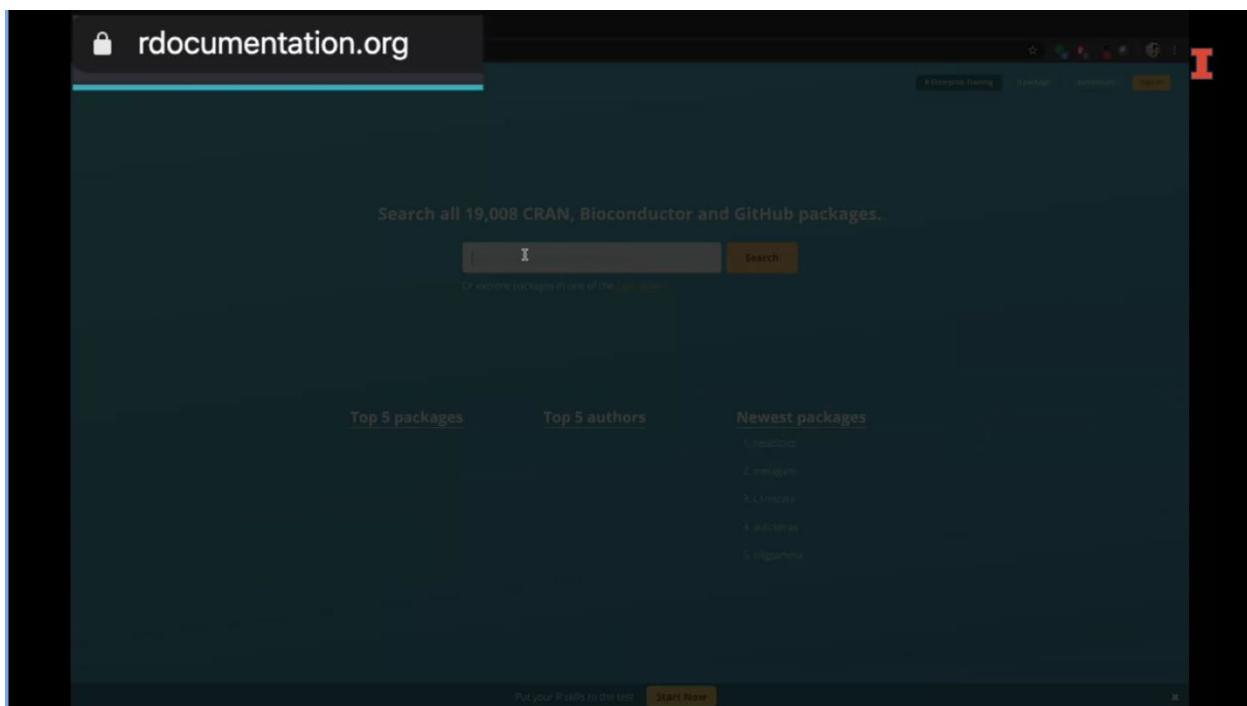
And you can see what's happening over here in the console as it is installing the package. And now it's installed and I can start using it. Let me X out of that subset, and I'll just scroll down. And as I scroll, notice that most of these packages do not have a check mark next to it. So a check mark indicates whether or not a package is loaded for use. So you can't really use the functions in a package until you load those functions into the environment.



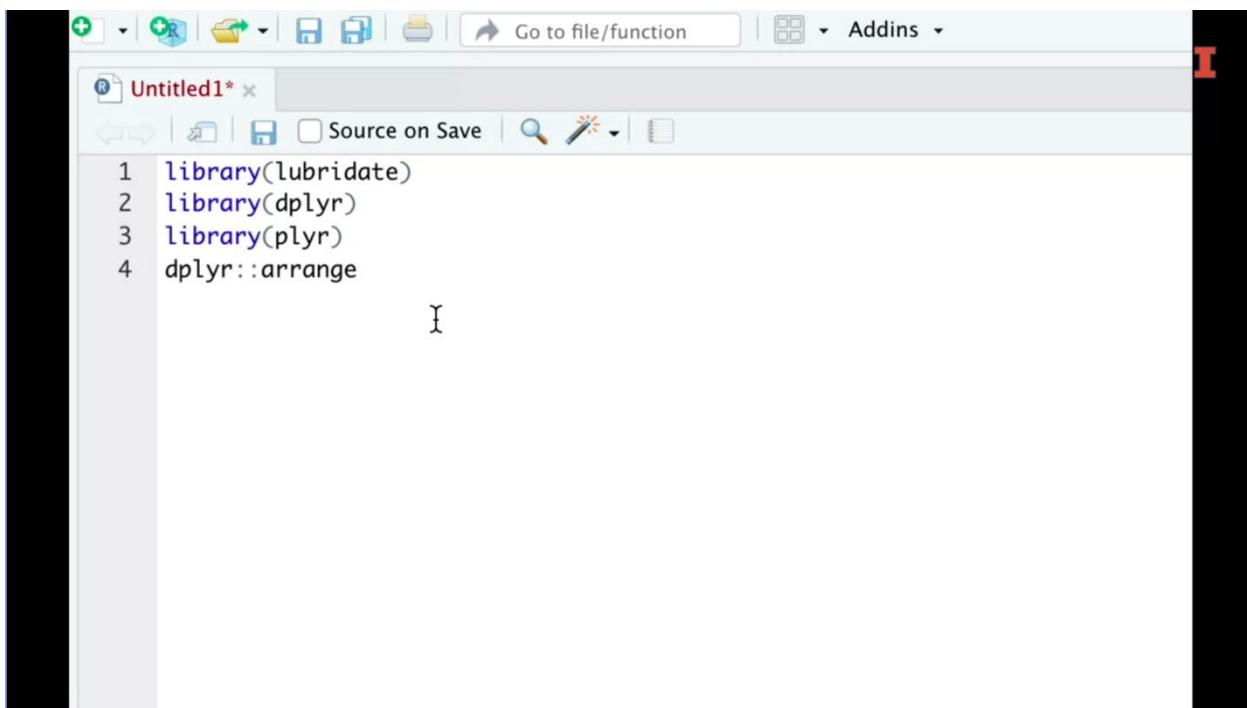
Let me show you how to do that. I'll create a new script, and if I want to load the functions in the lubridate package, I can type out library. And you can see it turns blue, and then it gives me a list of the different packages, and there's lubridate. So I'll select that, press Command+Enter. And you can see that in the console, it told me it has attached lubridate and now over in the Packages pane, I have a check mark next to lubridate. I can also just click the box, and that would load a package as well. So that loaded the lifecycle package, and I can click it again to unload it or detach it.



These packages do not stay the same for very long. R is continually updated, and other packages that some of these packages depend on are also updated. If you want to update your packages, you can click on the Update button in the Package pane, and it will give you a list of all the different packages that need to be updated. And you could simply select the ones you want or select all of them and then install the updates. Now, why wouldn't you ever want to keep them all updated?



Well, sometimes, when a package is updated, it may change how it works with other functions. And sometimes unintentionally, it prevents certain functions from working. Or sometimes, it stops using certain functions and replaces it with better functions. So whenever you update your package, you need to make sure that all of your scripts that use that package are also updated. Or you need to test them to make sure that they work as you expect them to. That's especially important if you're putting a script into a process that happens automatically. Another question is, why don't you just load all of the packages for use? Well, it takes a while to load packages, and that's just a lot more that has to be retained in the memory of the machine that you're using. So you really only want to load the package is that you currently need. Before I conclude there's one more thing about packages that I want to talk to you about and that is name space. And to illustrate this, I'm going to go back to rdocumentation.org website and I will search for a function, arrange.



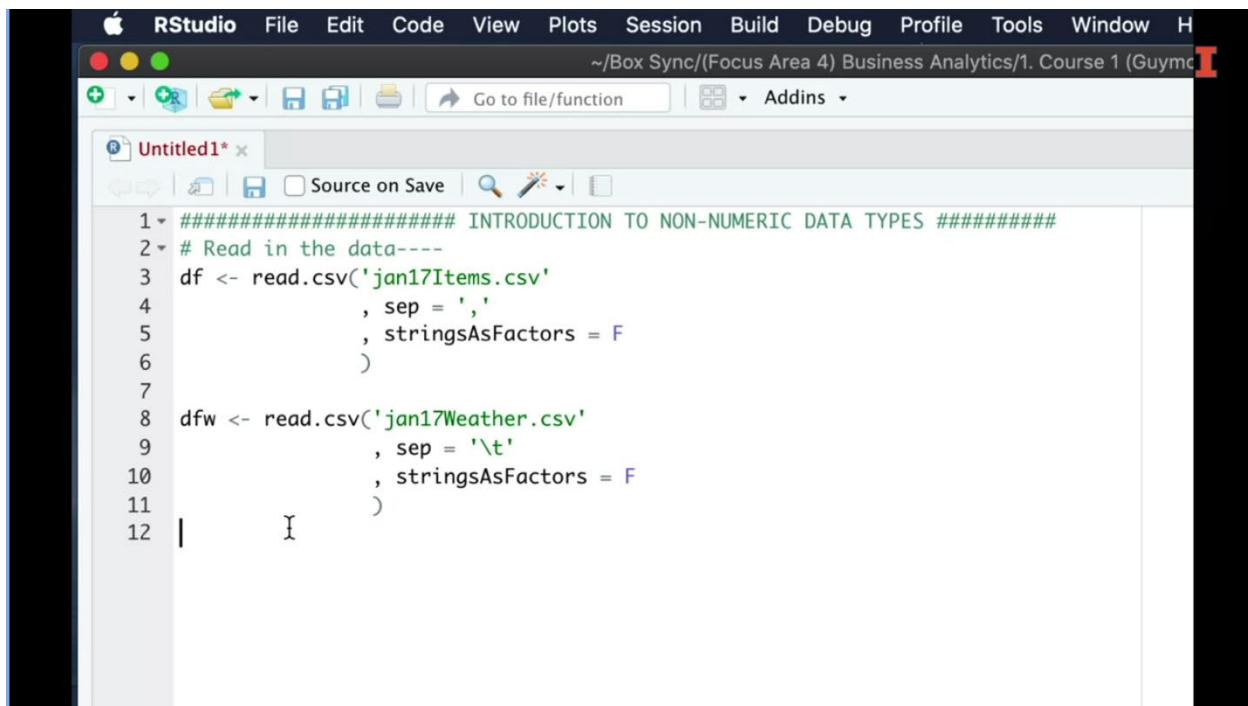
The screenshot shows the RStudio interface with a dark theme. A red 'I' icon is visible in the top right corner. The code editor window has a tab labeled 'Untitled1*' containing the following R code:

```
1 library(lubridate)
2 library(dplyr)
3 library(plyr)
4 dplyr::arrange
```

And notice that there are several `arrange` functions and that makes sense, it's a common name. And with over 19,000 packages, it's bound to be used in more than one package. And let's say I've got two packages `dplyr` and `plyr`, and they're both installed and loaded on my machine. If I want to use the `arrange` function, I need to tell R which version of `arrange` I want to use. The way I do that is, I type the name of the package and then two colons. And so this is telling R to use the `arrange` function from the `dplyr` package instead of the `plyr` package. So in conclusion, packages are really helpful. They allow us to focus on data analytic tasks and not worry as much about the low level programming.

Lesson 3-5 Introduction to Other Data Types

Lesson 3-5.1 Introduction to Other Data Types



The screenshot shows the RStudio interface with a script file named 'Untitled1'. The code in the script reads:

```
1 ##### INTRODUCTION TO NON-NUMERIC DATA TYPES #####
2 # Read in the data---
3 df <- read.csv('jan17Items.csv'
4   , sep = ','
5   , stringsAsFactors = F
6 )
7
8 dfw <- read.csv('jan17Weather.csv'
9   , sep = '\t'
10  , stringsAsFactors = F
11 )
12 |     |
```

To this point, we've talked a lot about numeric objects, vectors and columns in the data frame. In this lesson, I want to give you a brief introduction to other data types that you see in a data frame. So to start out with, let's open a new script. And give it a title in subheading and read in some data. Okay, I've given the title of introduction to non numeric data types and I've got a subheading for reading in the data and you can see that the data that I'm going to read in is the jan17Items.csv and jan17Weather.csv files and I'll store those as df and dfw data frame objects, respectively. So I'll go ahead and run that.

The screenshot shows the RStudio interface. On the left is the code editor with the following R code:

```

10      , stringsAsFactors = F
11  )
12 str(df)
13 # Character strings-----
14 x <- 'foXK'
15 |

```

The code is run in the console, resulting in:

```

15:1  Character strings : 
Console Terminal Jobs 
~/Box Sync/(Focus Area 4) Business Analytics/1.Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts & Slides_MBA561 MOOC/M1/RScripts_Module1/ 
$ Category      : chr "Glass Bottle" "Lamb Chops" "Salmon and Wheat Bran Salad" "Fountain" ...
$ CardholderName : chr NA NA NA NA ...
$ RegisterName  : chr "RT149" "RT151" "RT151" "RT151" ...
$ StoreNumber   : chr "AZ23501411" "AZ23501251" "AZ23501305" "AZ23501289" ...
$ TransactionNumber: chr "002670578157" "00XT6G2179417" "00XT6G2179417" "00XT6G2179417" ...
$ CustomerCode  : chr "CWM11331L80" "CXV10742CJW" "CXV10742CJW" "CXV10742CJW" ...
$ Cost          : num 0.11 0.11 0.11 0.11 0.11 0.11 0.11 0.11 0.11 ...
$ Price         : num 3.73 13.38 13.9 2.88 4.83 ...
$ Quantity      : int 1 1 1 1 1 1 1 1 ...
$ Modifiers     : num 0.01 0.01 0.01 0.01 2.36 2.36 1.08 1.08 0.01 ...
$ Subtotal      : num 3.74 13.39 13.91 2.89 4.84 ...
$ Discounts     : num 3 -0.03 -0.03 -0.03 -0.03 -0.03 -0.03 -0.03 3.63 ...
$ NetTotal      : num 0.74 13.42 13.94 2.92 4.87 ...
$ Tax           : num NaN 1.06 1.09 0.23 0.38 1.13 1.13 1.23 1.04 NaN ...
$ TotalDue      : num NaN 14.48 15.03 3.15 5.25 ...
> # Character strings-----
> x <- 'Fox'
> |

```

To the right of the code editor is the file browser, showing a directory structure under 'elwal'.

And a quick overview in the environment. Pain shows me that those were read in and it looks like they were read in correctly. So perfect. Now, let's go ahead and look at the column types of the df data frame and I will use the structure command str and df. And let's focus on this column here that tells us the type so we can see that for all of the first 12 or so columns. They are character string types chr, whereas the last miner. So columns are numeric types, so num for numerical, int for integer. So those numeric columns are the types of vectors that we've been focusing on. At this point, I want to talk to you just briefly about these other data types so that you have an idea of what to do with them and what to expect. And then we'll spend other lessons focusing specifically on each data type and how you can manipulate data in those types of vectors. So let's start by talking about character strings.

The screenshot shows the RStudio interface. In the top-left pane, there is R code:

```

1: # INTRODUCTION TO NON-NUMERIC DATA TYPES ----
2: # Read in the data ----
3: df <- read.csv("jan17Items.csv")
4:      , sep = ','
5:      , stringsAsFactors = F
6: )
7:
8: dfw <- read.csv("jan17Weather.csv")
9:      , sep = ','
10:     , stringsAsFactors = F
11: )
12: str(df)
13: # Character strings ----
14: x <- 'fox'
15:

```

In the top-right pane, the "Environment" tab is selected, showing the global environment:

- df: 8899 obs. of 21 variables
- dfw: 31 obs. of 5 variables
- Values: x "fox"

A tooltip over the "x" entry in the Values list says "x (character, 112 bytes)".

The bottom-left pane is the "Console" tab, showing the same R code and its output.

The bottom-right pane is the "Files" tab, showing a list of files:

- IntroductionToNonNumericDataTypes.R: 6.4 KB, Feb 18, 2020, 3:22 PM
- jan17Items.csv: 1.7 MB, Feb 15, 2020, 9:26 PM
- jan17Weather.csv: 1.5 KB, Feb 15, 2020, 9:30 PM
- readingAndWritingData.R: 4.1 KB, Feb 6, 2020, 10:03 PM
- RScripts_Module1.Rproj: 205 B, Feb 18, 2020, 2:15 PM

And let's start by creating a simple character string object. We'll create the object `x` and assign it the value of fox, the word fox. And you can see that I put in quotation marks to indicate that I'm not assigning it the value from another object, but just this character string of fox. Now, if you look over in the environment, you see that I've got this X. If I hover over a little, it says it's a character and tells me how many bites. It is. I can see it's fox. I can also find out the type of an object by using the `class` function. So I do.

The screenshot shows the RStudio interface. In the top-left pane, there is R code:

```

12: str(x)
13: # Character strings----
14: x <- 'fox'
15: class(x)
16: class(df$Price)
17: x <- g('fox', 'hound')

```

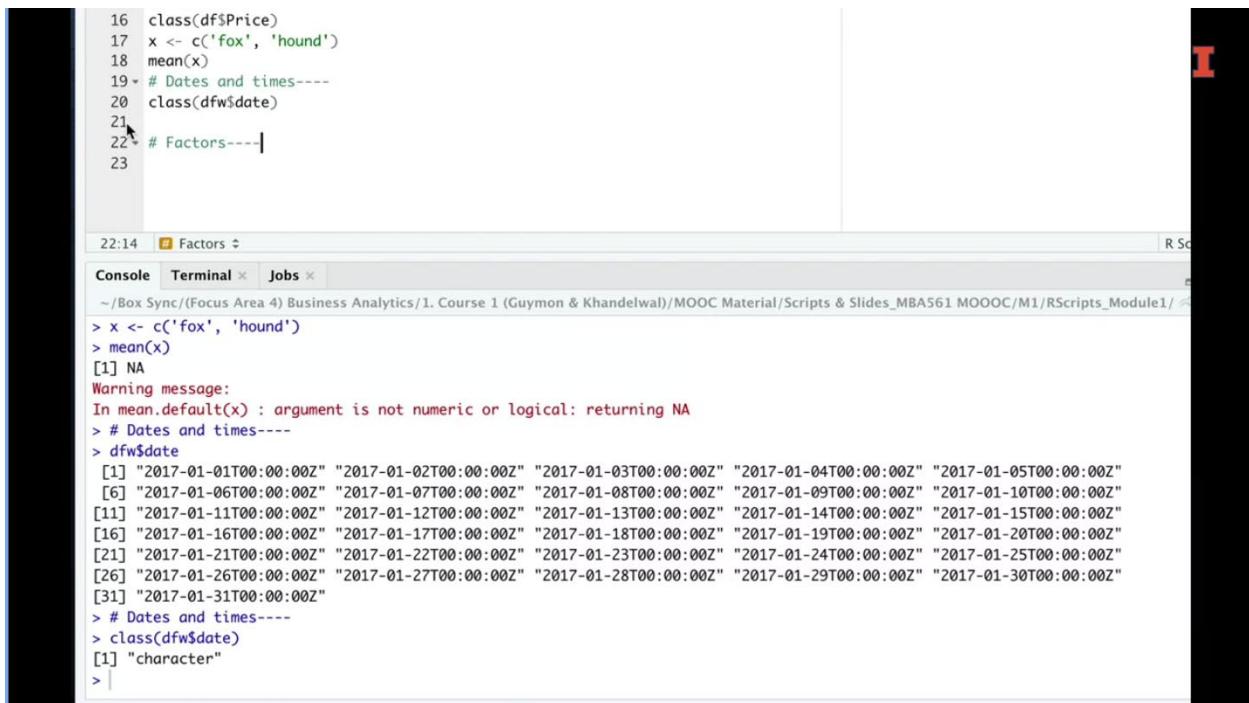
In the top-right pane, the "Environment" tab is selected, showing the global environment:

- Cost: num 0.11 0.11 0.11 0.11 0.11 0.11 0.11 0.11 0.11 ...
- Price: num 3.73 13.38 13.9 2.88 4.83 ...
- Quantity: int 1 1 1 1 1 1 1 1 1 ...
- Modifiers: num 0.01 0.01 0.01 0.01 0.01 2.36 2.36 1.08 1.08 0.01 ...
- Subtotal: num 3.74 13.39 13.91 2.89 4.84 ...
- Discounts: num 3 -0.03 -0.03 -0.03 -0.03 -0.03 -0.03 -0.03 3.63 ...
- NetTotal: num 0.74 13.42 13.94 2.92 4.87 ...
- Tax: num NaN 1.06 1.09 0.23 0.38 1.13 1.13 1.23 1.04 NaN ...
- TotalDue: num NaN 14.48 15.03 3.15 5.25 ...

The bottom-left pane is the "Console" tab, showing the same R code and its output.

The bottom-right pane is the "Files" tab, showing a list of files, identical to the one in the previous screenshot.

Class x down here in the consulate tells me it's a character. I can do that on a column of data as well. So if I do df and I'll just do time. That's also a character. If I do it on a different column, say price tells me it's numeric. Alright, I can create another character string vector by using the combined function, and I can enter two character strings. I'll do fox and hound, run that. And now you can see that x is a character. It does tell me now what type of object is. It's a character vector, and it has two values and fox and hound.



The screenshot shows an RStudio environment. The top pane displays R code:

```

16 class(df$Price)
17 x <- c('fox', 'hound')
18 mean(x)
19 # Dates and times----
20 class(dfw$date)
21
22 # Factors---|
23

```

The bottom pane shows the R console output:

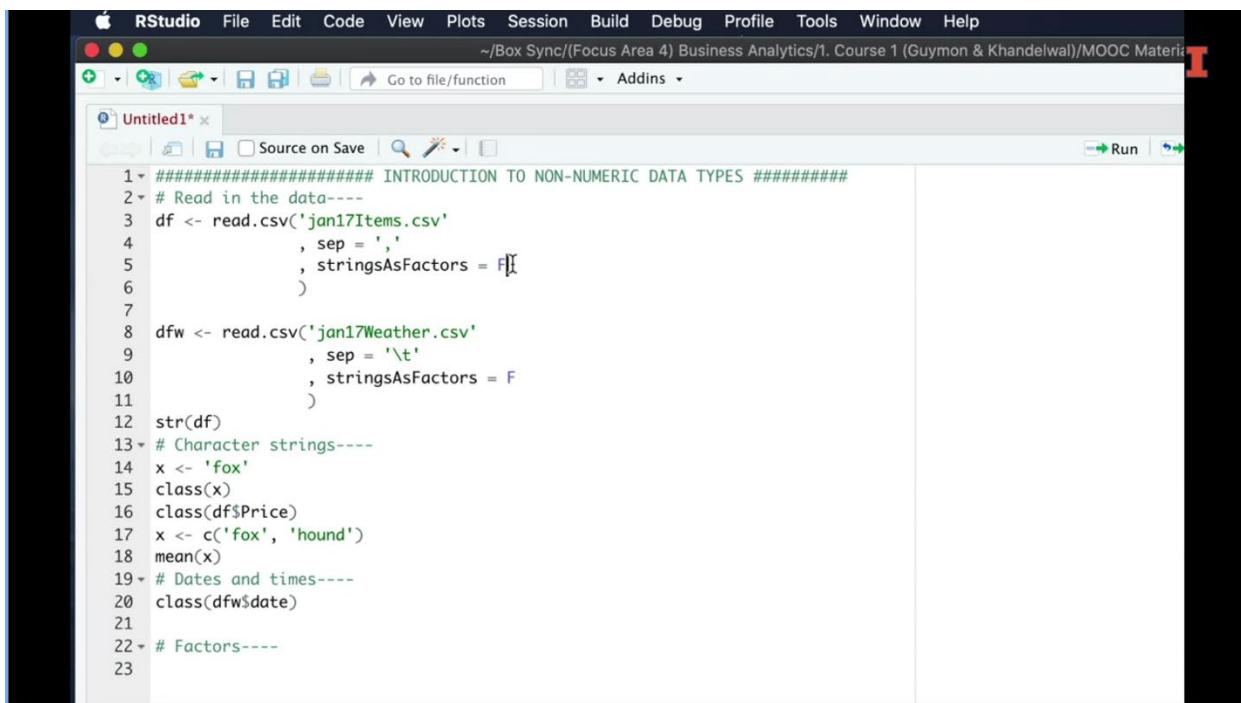
```

22:14  Factors ✘

Console Terminal ✘ Jobs ✘
~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts & Slides_MBA561 MOOOC/M1/RScripts_Module1/ <-->
> x <- c('fox', 'hound')
> mean(x)
[1] NA
Warning message:
In mean.default(x) : argument is not numeric or logical: returning NA
> # Dates and times---
> dfw$date
[1] "2017-01-01T00:00:00Z" "2017-01-02T00:00:00Z" "2017-01-03T00:00:00Z" "2017-01-04T00:00:00Z" "2017-01-05T00:00:00Z"
[6] "2017-01-06T00:00:00Z" "2017-01-07T00:00:00Z" "2017-01-08T00:00:00Z" "2017-01-09T00:00:00Z" "2017-01-10T00:00:00Z"
[11] "2017-01-11T00:00:00Z" "2017-01-12T00:00:00Z" "2017-01-13T00:00:00Z" "2017-01-14T00:00:00Z" "2017-01-15T00:00:00Z"
[16] "2017-01-16T00:00:00Z" "2017-01-17T00:00:00Z" "2017-01-18T00:00:00Z" "2017-01-19T00:00:00Z" "2017-01-20T00:00:00Z"
[21] "2017-01-21T00:00:00Z" "2017-01-22T00:00:00Z" "2017-01-23T00:00:00Z" "2017-01-24T00:00:00Z" "2017-01-25T00:00:00Z"
[26] "2017-01-26T00:00:00Z" "2017-01-27T00:00:00Z" "2017-01-28T00:00:00Z" "2017-01-29T00:00:00Z" "2017-01-30T00:00:00Z"
[31] "2017-01-31T00:00:00Z"
> # Dates and times---
> class(dfw$date)
[1] "character"
>

```

All right, so those are character strings. The important thing to recognize is that you cannot perform mathematical calculations on character strings. For example, if I tried to find out the mean of x, it would give me an error, because that's not a logical computation you can do with that. So that's important because sometimes you'll see that columns are read in as a character type instead of a numeric type. Now let's talk about dates and times briefly. [SOUND] If we look at the dfw, the weather data frame and look at the date column, we can see that it's obviously a date, and it also has a time stamp in there. So it obviously has a date and time format, but it's not a date and time type. In fact, if I look at the class of this column tells me it's a character, all right, so just keep that in mind, will spend another lesson working on dates and times. But at this point, just recognize that because it's a character vector, we cannot calculate, for instance, how far apart two different dates are or what the date will be 362 days in advance.



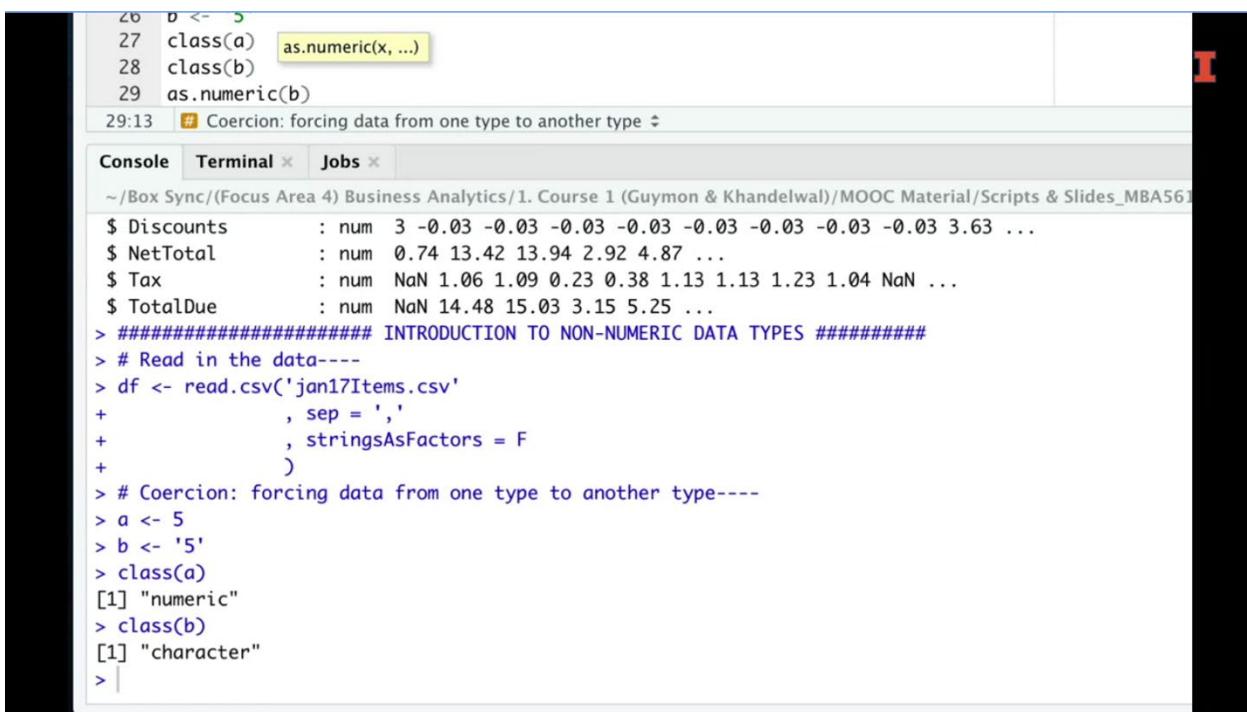
The screenshot shows the RStudio interface with the following R code in the script editor:

```

1 # ##### INTRODUCTION TO NON-NUMERIC DATA TYPES #####
2 # Read in the data---
3 df <- read.csv('jan17Items.csv'
4   , sep = ','
5   , stringsAsFactors = F)
6 )
7
8 dfw <- read.csv('jan17Weather.csv'
9   , sep = '\t'
10  , stringsAsFactors = F
11  )
12 str(df)
13 # Character strings---
14 x <- 'fox'
15 class(x)
16 class(df$Price)
17 x <- c('fox', 'hound')
18 mean(x)
19 # Dates and times---
20 class(dfw$date)
21
22 # Factors---
23

```

Now let's talk about factors. Remember that when we read in csv data, we have this argument where we indicate manually that strings as factors are false. What happens if we convert that to true and then run that and then look at the structure of df. Notice what happens to the data type for these first 12 or so columns. They change from character strings, to factors. So at this point, just recognize that our assumes that you want to make a conversion from character strings to numeric data types and factor is a special numeric data type, and we will talk about that later. So I'll leave that as strings as factors as false for now and read in the data.



The screenshot shows an RStudio interface with a code editor and a console window. The code editor has a few lines of R code, and the console window displays the output of that code, which includes some data frames and a demonstration of coercing a character string into a numeric type.

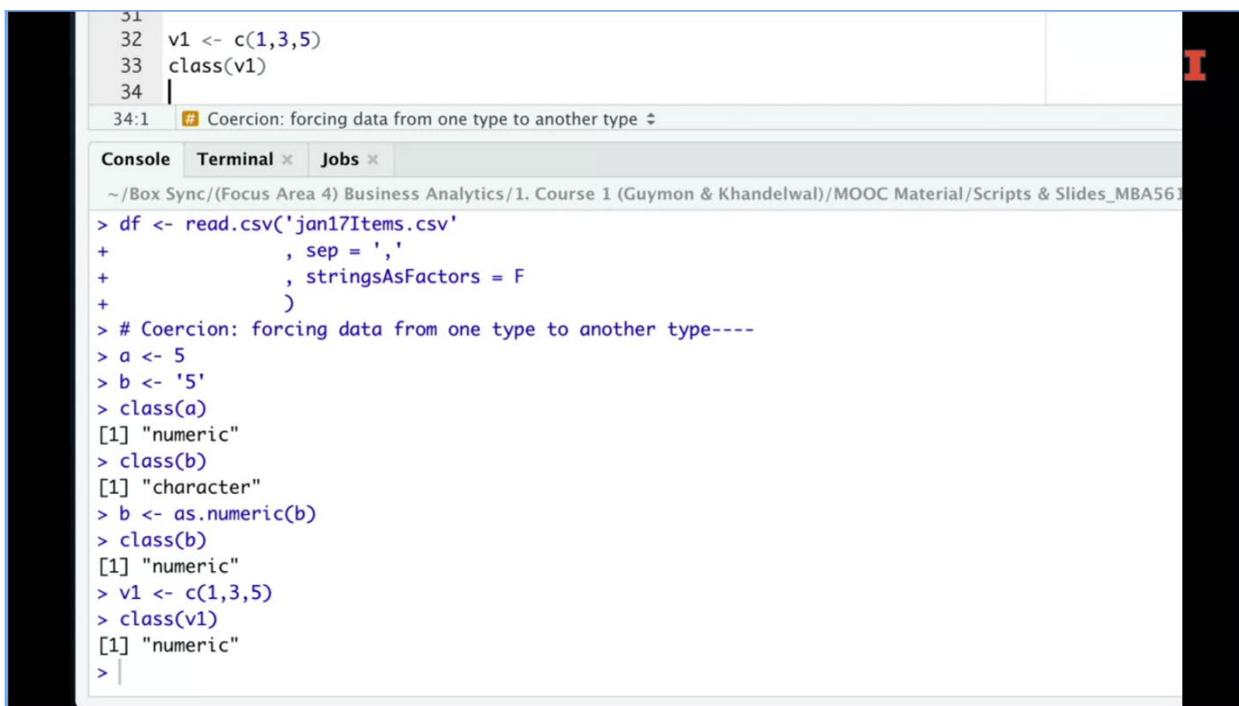
```

26 b <- 5
27 class(a) as.numeric(x, ...)
28 class(b)
29 as.numeric(b)
29:13 # Coercion: forcing data from one type to another type

Console Terminal × Jobs ×
~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts & Slides_MBA561
$ Discounts      : num  3 -0.03 -0.03 -0.03 -0.03 -0.03 -0.03 -0.03 3.63 ...
$ NetTotal       : num  0.74 13.42 13.94 2.92 4.87 ...
$ Tax            : num  NaN 1.06 1.09 0.23 0.38 1.13 1.13 1.23 1.04 NaN ...
$ TotalDue       : num  NaN 14.48 15.03 3.15 5.25 ...
> ##### INTRODUCTION TO NON-NUMERIC DATA TYPES #####
> # Read in the data----
> df <- read.csv('jan17Items.csv'
+                 , sep = ','
+                 , stringsAsFactors = F
+                 )
> # Coercion: forcing data from one type to another type----
> a <- 5
> b <- '5'
> class(a)
[1] "numeric"
> class(b)
[1] "character"
>

```

Okay, now let's talk about coercion, and what I mean by coercion is forcing data from one type to another type. [SOUND] This is important because, as I mentioned, sometimes data is read in and a column of numeric data is read in as a character string, and so you want to convert it to a numeric type. So let's start this discussion by talking about two simple objects. A will be numeric value of 5, b will be a character string of 5. All right, if I look at the type of A it is numeric and of B, it is character. So if I want to convert B to numeric type, I can use this as dot numeric function, and it will make b it will coerce b to be a numeric type so I'll assign that to itself again.



```

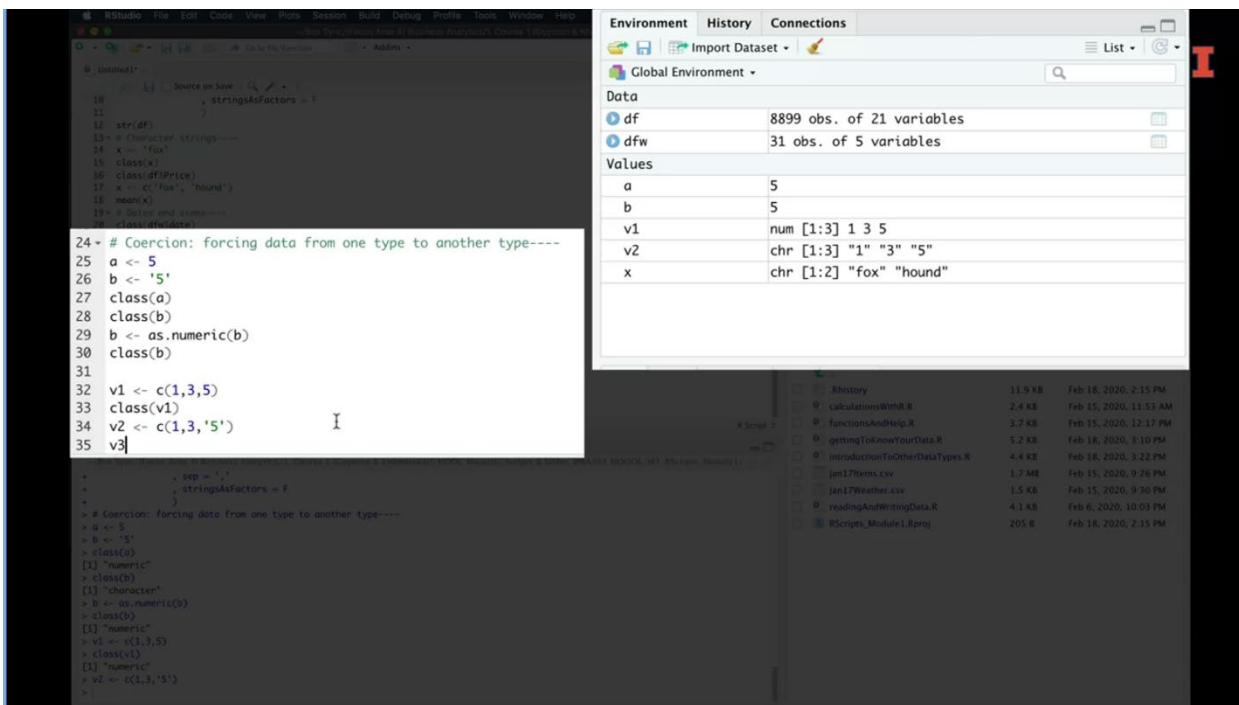
31
32 v1 <- c(1,3,5)
33 class(v1)
34 |
34:1 # Coercion: forcing data from one type to another type

Console Terminal × Jobs ×
~ /Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts & Slides_MBA561

> df <- read.csv('jan17Items.csv'
+                 , sep = ','
+                 , stringsAsFactors = F
+                 )
> # Coercion: forcing data from one type to another type----
> a <- 5
> b <- '5'
> class(a)
[1] "numeric"
> class(b)
[1] "character"
> b <- as.numeric(b)
> class(b)
[1] "numeric"
> v1 <- c(1,3,5)
> class(v1)
[1] "numeric"
>

```

And now if I look at the class of B, it is numeric, so you can explicitly force objects to be a certain data type. So why are numeric vector sometimes read in as character vectors? Well, let's look at what happens if we create a numeric vector. Let's say of 1, 3, 5 and we look at the class, it is numeric.



```

10 str(df)
11 # Coercion: strings----
12 x <- "fox"
13 class(x)
14 (df$dfPPrice)
15 x <- c("fox", "hound")
16 mean(x)
17 # Data and class----
18 class(dfw)

24 # Coercion: forcing data from one type to another type----
25 a <- 5
26 b <- '5'
27 class(a)
28 class(b)
29 b <- as.numeric(b)
30 class(b)
31
32 v1 <- c(1,3,5)
33 class(v1)
34 v2 <- c(1,3,'5')
35 v3 |  I

+                 , sep = ','
+                 , stringsAsFactors = F
+                 )
> # Coercion: forcing data from one type to another type----
> a <- 5
> b <- '5'
> class(a)
[1] "numeric"
> class(b)
[1] "character"
> b <- as.numeric(b)
> class(b)
[1] "numeric"
> v1 <- c(1,3,5)
> class(v1)
[1] "numeric"
> v2 <- c(1,3,'5')
>

Environment History Connections
Import Dataset List
Global Environment
Data
df 8899 obs. of 21 variables
dfw 31 obs. of 5 variables
Values
a 5
b 5
v1 num [1:3] 1 3 5
v2 chr [1:3] "1" "3" "5"
x chr [1:2] "fox" "hound"

File History 11.9 KB Feb 18, 2020, 2:15 PM
calculationsWithR.R 2.4 KB Feb 15, 2020, 11:53 AM
functionsAndHelp.R 3.7 KB Feb 15, 2020, 12:17 PM
gettingToKnowYourData.R 5.2 KB Feb 18, 2020, 1:10 PM
introductionToOtherDataTypes.R 4.4 KB Feb 18, 2020, 3:22 PM
jan17Items.csv 1.7 MB Feb 15, 2020, 9:26 PM
jan17Weather.csv 1.5 KB Feb 15, 2020, 9:30 PM
readingAndWritingData.R 4.1 KB Feb 6, 2020, 10:03 PM
RScripts_Module1.Rproj 205 KB Feb 18, 2020, 2:15 PM

```

And now let's compare that to a similar vector that has mixed types in there. So I'll do 1, 3 and then in quotation marks 5. And if I look at the data type for that, it is character.

So whenever you mix together numeric and character data types, it will coerce all of the data to be a character data type. And that's typically the case for all data types.

The screenshot shows an RStudio interface. The left pane displays R code in an R script window:

```

1 # Generated by RStudio
2 str(df)
3 ## # Coercing strings ----
4 x <- 'fox'
5 class(x)
6 class(df$Price)
7 x <- c('fox', 'hound')
8 mean(x)
9 ## # Coercing and times ----
10 class(df$date)
11
12 ## # Coercing
13 b <- '5'
14 class(a)
15 class(b)
16 b <- as.numeric(b)
17 class(b)
18
19 v1 <- c(1,3,5)
20 class(v1)
21 v2 <- c(1,3,'5')
22 v3 <- as.numeric(v2)
23 v4 <- as.integer(v2)
24
25
26
27
28
29
30
31
32
33
34
35
36
37

```

The right pane shows the Environment tab of the Global Environment panel, listing objects df (8899 obs. of 21 variables) and dfw (31 obs. of 5 variables), along with their values:

Object	Type	Value
a	5	
b	5	
v1	num [1:3] 1 3 5	
v2	chr [1:3] "1" "3" "5"	
v3	num [1:3] 1 3 5	
v4	int [1:3] 1 3 5	
x	chr [1:2] "fox" "hound"	

Below the environment pane, the History tab shows a list of recent scripts and files:

- Rhistory 11.9 KB Feb 18, 2020, 2:15 PM
- calculationsWithR.R 2.4 KB Feb 15, 2020, 11:53 AM
- functionsAndHelp.R 3.7 KB Feb 15, 2020, 12:17 PM
- gettingToKnowYourData.R 5.2 KB Feb 18, 2020, 1:10 PM
- introductionToOtherDataTypes.R 4.4 KB Feb 18, 2020, 3:22 PM
- jan17Items.csv 1.7 MB Feb 15, 2020, 9:26 PM
- jan17Weather.csv 1.5 KB Feb 15, 2020, 9:30 PM
- readingAndWritingData.R 4.1 KB Feb 6, 2020, 10:01 PM
- RScripts_Module1.Rproj 205 B Feb 18, 2020, 2:15 PM

Characters like the default data type. We could coerce v to be a numeric data type by using this as.numeric function for v2. And now if we look at v3, it has the same values in it. But there are numeric values instead of character values, all right, we can also coerce those to be integer, and if we look at v4, we can see that those are integers, now. The difference between integer and numeric really isn't important at this point, so we won't worry about that right now. So one other thing that I want to mention with respect to coercion is what happens if you have a vector that includes numeric values and strings that can't be converted into numeric types.

The screenshot shows an RStudio interface. On the left, the 'Console' tab displays R code and its output. The code involves coercing data types, specifically converting character vectors to numeric vectors and vice versa. It includes operations like `as.numeric` and `as.integer`. A warning message is visible at the bottom of the console regarding NA values.

```

14 x <- "cos"
15 class(x)
16 class(as.numeric)
17 x <- c("fox", "hound")
18 mean(x)
19 # Data and trees
20 class(df$date)
21
22 # Factors
23
24 # coercion: forcing data from one type to another type
25 a <- 5
26 b <- '5'
27 class(a)
28 class(b)
29 b <- as.numeric(b)
30 class(b)
31
32 v1 <- c(1,3,5)
33 class(v1)
34
35 v4 <- as.integer(v2)
36
37 v5 <- c(1,3,'five')
38
39

```

The 'Environment' tab on the right shows the global environment with two objects:

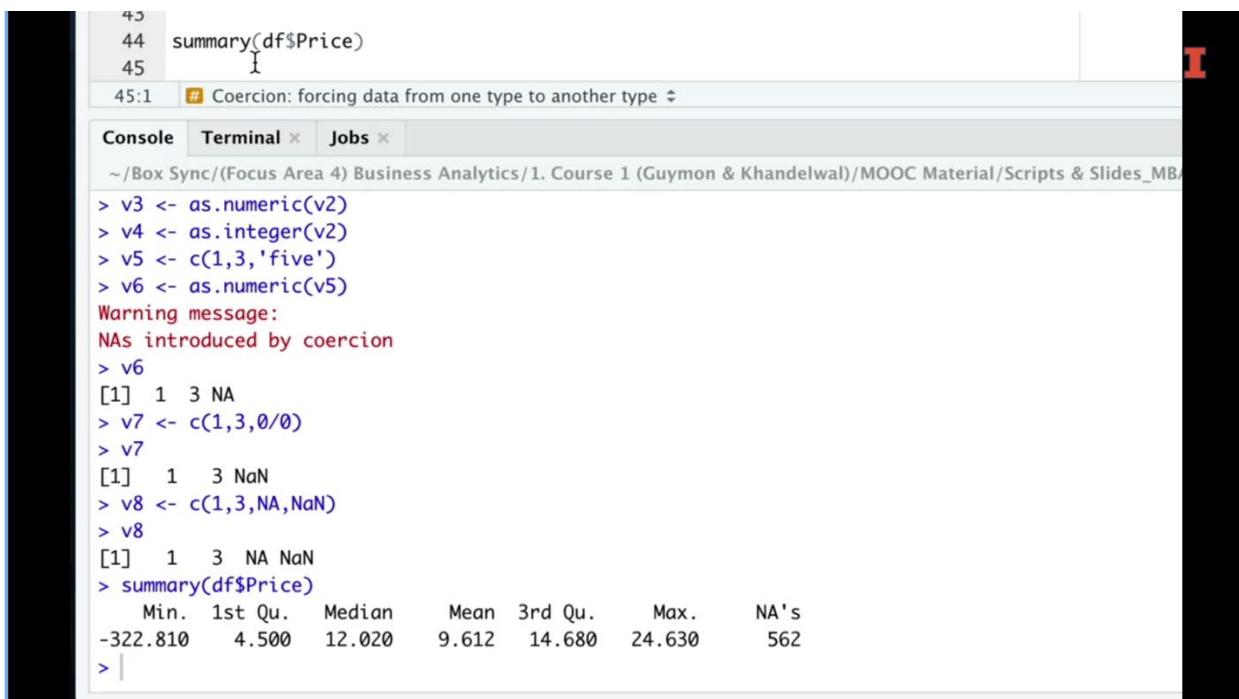
- df**: 8899 obs. of 21 variables
- dfw**: 31 obs. of 5 variables

Under 'Values', the following variables are listed with their types and values:

Variable	Type	Value
a	int	5
b	chr	5
v1	num	[1:3] 1 3 5
v2	chr	[1:3] "1" "3" "5"
v3	num	[1:3] 1 3 5
v4	int	[1:3] 1 3 5
v5	chr	[1:3] "1" "3" "five"
x	chr	[1:2] "fox" "hound"

The 'File' menu at the top includes options like 'File', 'Edit', 'Code', 'View', 'Files', 'Dataset', 'Build', 'Group', and 'Profile'.

So let's say, for instance, I've got vector 5 and that is equal to 1, 3. And then the word 5, you can look and see that, that is a character vector. If I try to coerce that to be numeric, I get this warning at the bottom that, says NAs are introduced by coercion, and I can look over at v6 or print out v6, and I can see that it is 1, 3 and then that word 5 was converted to NA. Now NA stands for not applicable, meaning it couldn't really coerce the word 5 into a numeric value. So it just puts in NA, now. Sometimes you'll also see something similar to NA, which is NaN, and that happens if you have a value that is not a numeric value.

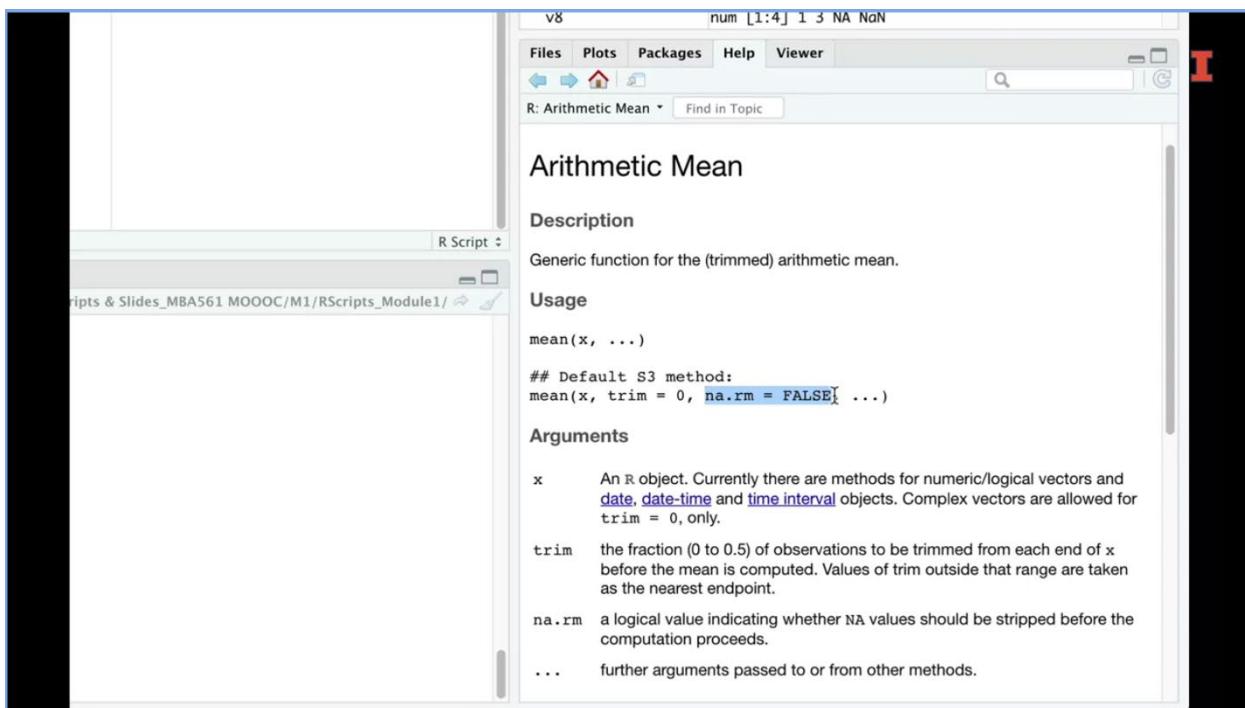


```

43
44 summary(df$Price)
45
45:1 # Coercion: forcing data from one type to another type
Console Terminal Jobs
~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts & Slides_MBA.R
> v3 <- as.numeric(v2)
> v4 <- as.integer(v2)
> v5 <- c(1,3,'five')
> v6 <- as.numeric(v5)
Warning message:
NAs introduced by coercion
> v6
[1] 1 3 NA
> v7 <- c(1,3,0/0)
> v7
[1] 1 3 NaN
> v8 <- c(1,3,NA,NaN)
> v8
[1] 1 3 NA NaN
> summary(df$Price)
   Min. 1st Qu. Median   Mean 3rd Qu. Max. NA's
-322.810    4.500   12.020  9.612  14.680  24.630    562
>

```

If, for instance, you have 1, 3 and let's say 0 divided by 0. All right. And then we print out v7. You can see that it changes that 0/0, which would be undefined to NaN. For not a number, you can even manually, you can create vectors that have NAs and then ends in them just by using NA and NaN, and you can see that they turn blue because they are special words, reserved words that have a special meaning to them. And anyway, so those are important because when you see those in a character or sorry in a numeric vector, it's important that you know what to do with them. Let's say, for instance, I want to calculate the summary statistics for the price column of the items data frame. I can do that, and it tells me that I've got 562 NAs in there, now by default.



If you try to perform a numeric calculation on this vector, this price column, it will try to include NAs and NaNs in that mean calculation, and it can't do it. And so it returns NaN because it doesn't know what to do with NAs. So if you look at the help for these calculations, you can see that one of the arguments is `na.rm`, and it is defaulted to false. Now what is `na.rm` stand for? It says it's a logical value, indicating whether NA values should be stripped before the computation proceeds. So if it's false, they will not be stripped. If it's true, then they will be stripped. So we need to make sure that we include explicitly this attribute `na.rm`, and set it equal to true meaning. We want to remove the NA values.

```

42 v7 <- c(1,3,NA,NaN)
43
44 summary(df$Price)
45 mean(df$Price,na.rm = T)
46 ?mean
47 min(df$Price, na.rm = T)
48 |
48:1 # Coercion: forcing data from one type to another type

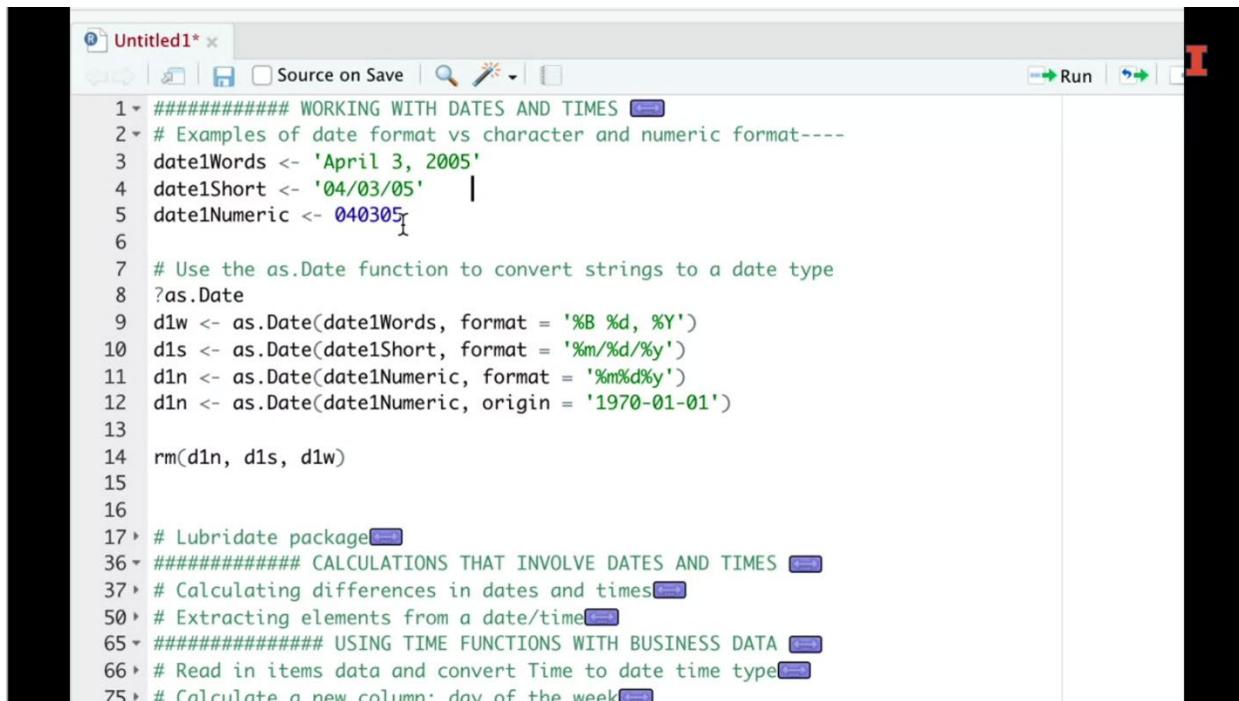
Console Terminal Jobs
~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts & Slides_MBA561 MOOOC/M1/RSc
> v7
[1] 1 3 NA NaN
> v8 <- c(1,3,NA,NaN)
> v8
[1] 1 3 NA NaN
> summary(df$Price)
   Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
-322.810 4.500 12.020 9.612 14.680 24.630 562
> mean(df$Price)
[1] NaN
> ?mean
> mean(df$Price,na.rm = T)
[1] 9.611895
> min(df$Price)
[1] NaN
> min(df$Price, na.rm = T)
[1] -322.81
>

```

And now, if I run that it will tell me the value. The mean value of that price column in the df data frame. And that is the case with many other numeric functions. So, for instance, if I tried the minimum of the price column, it will return NaN unless I explicitly indicate na.rm is equal to true. And then it gives me the minimum value, so there's an introduction to a few other non numeric data types as well as what to do when you have NA and NaN in numeric vectors.

Lesson 3-6 Creating Date Types

Lesson 3-6.1 Creating Date Types



The screenshot shows an RStudio interface with a dark theme. The code editor window contains R code for working with dates and times. The code includes examples of date format conversion from words ('April 3, 2005') to short ('04/03/05') and numeric ('040305') formats. It also demonstrates the use of the `as.Date` function to convert strings to date types and the Lubridate package. The code is organized into sections: WORKING WITH DATES AND TIMES, CALCULATIONS THAT INVOLVE DATES AND TIMES, and USING TIME FUNCTIONS WITH BUSINESS DATA.

```
1 ###### WORKING WITH DATES AND TIMES
2 # Examples of date format vs character and numeric format---
3 date1Words <- 'April 3, 2005'
4 date1Short <- '04/03/05' |
5 date1Numeric <- 040305
6
7 # Use the as.Date function to convert strings to a date type
8 ?as.Date
9 d1w <- as.Date(date1Words, format = '%B %d, %Y')
10 d1s <- as.Date(date1Short, format = '%m/%d/%y')
11 d1n <- as.Date(date1Numeric, format = '%m%d%y')
12 d1n <- as.Date(date1Numeric, origin = '1970-01-01')
13
14 rm(d1n, d1s, d1w)
15
16
17 # Lubridate package
18 ###### CALCULATIONS THAT INVOLVE DATES AND TIMES
19 # Calculating differences in dates and times
20 # Extracting elements from a date/time
21 ###### USING TIME FUNCTIONS WITH BUSINESS DATA
22 # Read in items data and convert Time to date time type
23 # Calculate a new column: day of the week
```

In this lesson, we're going to focus on understanding the date format and how to convert strings and numeric values to a date format. Dates are a big topic, we won't be able to cover everything in this lesson or this course, even on dates, but hopefully, we'll give you enough to get you started so that you can accomplish many of the tasks that are required in business analytics. Now, business data often has dates in it. One of the problems with dates though is that they recorded in many different ways. For example, you can record the date of April 3rd, 2005 in this manner, or what is often the case where I live is it's abbreviated as a two-digit month forward slash two-digit day forward slash two-digit year.

The screenshot shows the RStudio interface. On the left, there's a script editor window titled 'Untitled1' containing R code for working with dates and times. The code includes examples of date format vs character and numeric format, and calculations involving dates and times. On the right, there's a 'Files' tab showing a list of packages in the System Library, with 'base' checked. The 'Environment' tab shows three objects: 'date1Numeric' (40305), 'date1Short' ("04/03/05"), and 'date1Words' ("April 3, 2005"). The 'Plots' tab is active.

```

1 - ##### WORKING WITH DATES AND TIMES #####
2 - # Examples of date format vs character and numeric format----
3 date1Words <- 'April 3, 2005'
4 date1Short <- '04/03/05'
5 date1Numeric <- 40305
6
7 # Use the as.Date function to convert strings to a date type
8 ?as.Date
9 d1w <- as.Date(date1Words, format = '%B %d, %Y')
10 d1s <- as.Date(date1Short, format = '%m/%d/%y')
11 d1n <- as.Date(date1Numeric, format = '%m/d/%y')
12 d1n <- as.Date(date1Numeric, origin = '1970-01-01')
13
14 rm(d1n, d1s, d1w)
15
16
17 # Lubridate package
18 ###### CALCULATIONS THAT INVOLVE DATES AND TIMES #####
19 # Calculating differences in dates and times
20 # Extracting elements from a date/time
21 ###### USING TIME FUNCTIONS WITH BUSINESS DATA #####
22 # Read in items data and convert Time to date time type
23 # Calculate a new column: day of the week
24 # Calculate a new column: hour of the day
25
26
27 # Examples of date format vs character and numeric format

```

Then sometimes that can be recorded just as a numeric sequence of numbers, like so. I'm going to create these three objects here. The key with dates is that we want to both be able to read what the date is, and also we want to be able to make calculations with date, such as calculating the difference between two dates, how much time has passed. Let's start by learning about some of the base functions in R for dealing with dates. The key function that I'd like to teach you about is the as date function.

The screenshot shows the RStudio interface with the 'Date' package documentation open. The top menu bar includes 'File', 'Plots', 'Packages', 'Help', and 'Viewer'. The main content area displays the 'Date Conversion Functions to and from Character' page. It includes several examples of R code for date conversion between character strings and Date objects, as well as S3 methods for different classes like 'character', 'numeric', 'POSIXct', and 'Date'. Below the examples, the 'Arguments' section lists parameters for the 'as.Date' function, each with a detailed description.

```
R: Date Conversion Functions to and from Character
Functions to convert between character representations and objects of class "Date" representing calendar dates.

Usage

as.Date(x, ...)
## S3 method for class 'character'
as.Date(x, format, tryFormats = c("%Y-%m-%d", "%Y/%m/%d"),
        optional = FALSE, ...)
## S3 method for class 'numeric'
as.Date(x, origin, ...)
## S3 method for class 'POSIXct'
as.Date(x, tz = "UTC", ...)

## S3 method for class 'Date'
format(x, ...)

## S3 method for class 'Date'
as.character(x, ...)

Arguments

  x          an object to be converted.
  format     character string. If not specified, it will try tryFormats one by one on the first non-NA element, and give an error if none works. Otherwise, the processing is via strptime.
  tryFormats character vector of format strings to try if format is not specified.
  optional   logical indicating to return NA (instead of signalling an error) if the format guessing does not
```

Let's look at the hope documentation for this function. If we look at this, there are not a lot of arguments associated with it, but one thing that we do have to specify when using this function is the format in which the string is coming in. If we click on that strip time and look for the help in that, it takes us to a list of the abbreviations for indicating what type of format the date is recorded in. When we use this function, the way we do it is we call the function and then we add our date object, and then we have to specify the format.

The screenshot shows the RStudio interface with a help page open for "Date-time Conversion Functions". The code editor at the top contains R code demonstrating various date-time operations. The help page below provides detailed descriptions for each function:

- %a**: Abbreviated weekday name in the current locale on this platform. (Also matches full name on input: in some locales there are no abbreviations of names.)
- %A**: Full weekday name in the current locale. (Also matches abbreviated name on input.)
- %b**: Abbreviated month name in the current locale on this platform. (Also matches full name on input: in some locales there are no abbreviations of names.)
- %B**: Full month name in the current locale. (Also matches abbreviated name on input.)
- %c**: Date and time. Locale-specific on output, "%a %b %e %H:%M:%S %Y" on input.
- %C**: Century (00–99); the integer part of the year divided by 100.
- %d**: Day of the month as decimal number (01–31).

This date one word is the April 3rd, 2005. I have to indicate with a percentage sign in a capital B that this is the full month's name. Then there's a space, and then with the percentage d, that indicates the day of the month as a decimal number, and there's a comma, and then percent capital Y, and that capital Y is the four-digit year.

The screenshot shows the RStudio interface. The top bar includes tabs for 'RStudio', 'File', 'Edit', 'Code', 'Tools', 'View', 'Help', and 'Addins'. The title bar says 'RStudio - RStudio: RStudio'.

The main area contains R code:

```
## Examples of date format vs character and numeric format ----  
# Examples of date format vs character and numeric format ----  
date1Words <- "April 3, 2005"  
date1Short <- "04/03/05"  
date1Numeric <- 040305  
  
# Use the as.Date function to convert strings to a date type  
# as.Date  
# as.Date(date1Words, format = "%B %d, %Y")  
date1 <- as.Date(date1Short, format = "%m/%d/%y")  
date1 <- as.Date(date1Numeric, format = "%m%d%y")  
date1 <- as.Date(date1Numeric, origin = "1970-01-01")  
  
rm(date1, date1Words)  
  
# Lubridate package  
# Date-time calculations that involve dates and times  
# Calculating differences in dates and times  
# Extracting elements from a date/time  
# Date-time conversion functions to and from character  
# Read in items data and convert Time to date-time type  
# Calculate a new column: day of the week  
# Calculate a new column: hour of the day
```

The bottom left shows the 'Console' tab is active, with the message 'This session uses RStudio's Business Analytics Course'.

The bottom right shows the 'Global Environment' viewer with a tooltip over the date '2005-04-03'.

The bottom center shows the 'Files' tab is active, with a file named 'Date-time Conversion Functions to and from Character.R' open.

Let's go ahead and run that, and it creates this new object, d1 w. You can see that the format is the four-digit year dash, two-digit month dash, two-digit day. This is a standardized format known as ISO 8601 format. This is the format that is most often used when programming.

The screenshot shows an RStudio session with the following code:

```
date1Short <- '04/03/05'  
d1s <- as.Date(date1Short, format = '%m/%d/%y')  
d1m <- as.Date(date1Numeric, origin = '1970-01-01')  
rm(d1n, d1s, d1m)  
# Lubridate package  
# Examples of date format vs character and numeric format:  
# WORKING WITH DATES AND TIMES #####  
# Examples of date format vs character and numeric format----  
date1Words <- 'April 3, 2005'  
date1Short <- '04/03/05'  
date1Numeric <- 40805  
# Use the as.Date function to convert strings to a date type  
# los.Date  
# d1w <- as.Date(date1Words, format = "%B %d, %Y")  
#
```

The code demonstrates various ways to represent dates (character strings, numeric values, and date objects) and how to convert them using the `as.Date` function. It also shows how to use the `lubridate` package for date calculations.

Now let's go ahead and illustrate what you would need to do to use the update function to convert the same date but in a different shorthand notation here. We'd have to use a lower percent, lowercase m forward slash, and so forth, but if I run that, it gives us this new object and it is the same date so it worked. What about this numeric version down here?

The screenshot shows an RStudio interface with the following details:

- Top Bar:** RStudio, File, edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, Window, Help.
- Project:** Project (None)
- Environment Tab:** Shows variables d1s, d1w, date1Numeric, date1Short, and date1Words.
- Code Editor:** Contains R code for date manipulation. The code includes:
 - Comments: `# Examples of date format vs character and numeric formats----`
 - Assignments: `date1Words <- 'April 3, 2005'`, `date1Short <- '04/03/05'`, `date1Numeric <- 040305`.
 - A call to `as.Date`: `d1n <- as.Date(date1Numeric, format = '%m%d%Y')`.
 - A warning message: `Warning message: Error in as.Date(date1Numeric, format = "%m%d%Y") : 'origin' must be supplied`.
- Console Tab:** Displays the R session history with the same commands and the error message.
- Terminal Tab:** Shows a note about the ISO calendar: `Warning message: Error in as.Date(date1Numeric, format = "%m%d%Y") : 'origin' must be supplied. Note that this will break. Note that this will be used with agreement of UTC. Values up to +14:00 are pliable. This may not be`.
- Jobs Tab:** Shows a note about locale: `Warning message: Error in as.Date(date1Numeric, format = "%m%d%Y") : 'origin' must be supplied. Names are matched by platform and the locale. If that is in the C locale they are likely used in English. On any two of Linux, you wish to use ita, ita or th`.

If I run that and I tried to indicate using the symbols what it means, I get an error. This is because it's expecting a string and instead I gave it a number. When I give it a number, it says, he must be telling me that you want me to calculate how many days it has been since the epic. I have to supply that epic.

The screenshot shows the RStudio interface with the following code in the console:

```
1 <- suppressWarnings(WORKING WITH DATES AND TIMES)
2 # Examples of date format vs character and numeric Format-----
3 dateWords <- 'April 3, 2005'
4 date1Numeric <- 040305
5
6 # Use the as.Date function to convert strings to a date type
7 on.Date
8 d1w as.Date(dateWords, format = "%B %d, %Y")
9 d1s as.Date(date1Numeric, format = "%m/%d/%y")
10 d1n as.Date(date1Numeric, origin = '1970-01-01')

d1n <- as.Date(date1Numeric, origin = '1970-01-01')

14 rm(d1n, d1s, d1w)
15
16
17 ## Load relevant packages
18 ## WORKING WITH CALCULATIONS THAT INVOLVE DATES AND TIMES
19 ## Calculating differences in dates and times
20 ## Extracting elements from a Date/Time object
21 ## Date-time Conversion Functions to and from Characters
22 ## Read in file as data and convert Time in date-time type
23 ## Calculate a new column: day of the week
24 ## Calculate a new column: hour of the day

54 Examples of date format vs character and numeric format
```

The data frame 'date1' is displayed in the environment pane:

	date1	date1s	date1n
1	2005-04-03	2005-04-03	2005-04-03

The status bar at the bottom right indicates 'Date 1 3 rows'.

The help browser shows the 'Date-time Conversion Functions to and from Characters' page, noting that the default century inferred from a 2-digit year will change in a future version.

The footer of the RStudio window displays the R version and build information.

This next version here, I supplied the origin of January 1st of 1970. If I run that, I don't get an error and I get this new object. It worked, however, it is not the right date. Apparently, 40,305 days after January 1st 1970 is May 8th, 2080. It didn't convert it to the right date, even though it is a date format. You can see that there are some important considerations that you need to be aware of when converting the way a date is recorded into a date type. Let's go ahead and remove those objects that I just created.

```

16
17 # Lubridate package----
18
19 install.packages('lubridate')
20 library(lubridate)
21 d1w <- mdy(date1Words)  I
22 d1s <- mdy(date1Short)
23 d1n <- mdy(date1Numeric)
24 ymd('2005-04-03')
25
26
27 # Let's try converting times to a date time format
28 t1w <- mdy_hms('April 3, 2005 10:20:45')
29 t1w2 <- ydm_hms('2005, 03 April 10:20:45')
30 t1s <- ymd_hms('05/04/03 10:20:45')
31 t1n <- dmy_hm('030405T10:20')
32
33
34
17:1 # Lubridate package : 

Console Terminal x Jobs x
~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts & Scripts/01.R
> ##### WORKING WITH DATES AND TIMES #####
> # Examples of date format vs character and numeric format----
```

Now let's talk about the Lubridate package. The Lubridate package makes dealing with dates a whole lot easier. If you haven't installed the Lubridate package, go ahead and install it by using the `install.packages` function, or you can go ahead and go over to the packages pane and click on the Install utility and type in Lubridate and install that. Now I've already installed it. The next thing I need to do is make sure it's loaded, so I'll go ahead and load the package. I'll run that library Lubridate.

The screenshot shows the RStudio interface with the Lubridate package loaded. The Global Environment pane displays several objects:

Name	Description	Version
d1w	2005-04-03	
date1Short	"04/03/05"	
date1Words	"April 3, 2005"	

The Packages pane shows the Lubridate package installed with version 1.6.0. Other packages listed include base, base64enc, BH, boot, broom, callr, caret, cellranger, class, cli, clipr, cluster, codetools, colorspace, compiler, crayon, curl, datasets, DBI, dplyr, digest, etc.

Now I can access the functions in the Lubridate package. Now I'd like to illustrate how we can make these conversions on these strings in numeric representations of date into the appropriate date type using Lubridate. Let's start with the date one words, which is this April 3rd, 2005. The function I'm going to use is MDY, and what that stands for is month, day, year. Basically I'm saying first is the month, then as the day, then it's the year. I'll run that, and I can see in the environment that this object was created and it definitely looks like the right date, so very good.

The screenshot shows the RStudio interface. In the top-left pane, there is a code editor with R script content. The script includes code to convert strings to dates using the `lubridate` package. In the top-right pane, the "Environment" tab is selected, showing a table titled "Values" with three entries: d1n, d1s, and d1w, all corresponding to the date "2005-04-03". In the bottom-right pane, the "Packages" tab is selected, displaying a list of available packages in the System Library, with the "base" package checked.

```

date1Words <- "April 3, 2005"
date1Numeric <- 040305

# Use the as.Date function to convert strings to a date
d1s <- as.Date(date1Words, format = "%B %d, %Y")
d1s <- as.Date(date1Words, format = "%m/%d/%y")
d1n <- as.Date(date1Numeric, format = "%m%d%y")
d1w <- as.Date(date1Numeric, origin = "1970-01-01")

rm(d1n, d1s, d1w)

# Lubridate package
install.packages("lubridate")
library(lubridate)
d1n <- mdy(date1Words)

d1n <- mdy(date1Numeric)

```

What about if I use the MDY on this date one short, go ahead and run that, and awesome. It converted it to the same date, exactly what we want. Even though it was a different format, since it was the month, day, year, Lubridate was able to interpret that 04 meant the month and converted it to April. What about the numeric version? We'll go ahead and run that. Wonderful. This converted it to the right data as well, even though it was a number and not a string. This is awesome because it's just one function and it's taking into consideration the different ways in which the date could have been recorded and making the appropriate transformation to the date that we want and the date format.

The screenshot shows the RStudio interface. The left pane displays an R script with code for date conversion using the lubridate package. The right pane shows the 'System Library' with many packages listed. The bottom pane has tabs for 'Console', 'Terminal', and 'Jobs', with the 'Console' tab active, showing the execution results of the R commands.

```
15  
16  
17 # Lubridate package----  
18  
19 install.packages('lubridate')  
20 library(lubridate)  
21 d1w <- mdy(date1Words)  
22 d1s <- mdy(date1Short)  
23 d1n <- mdy(date1Numeric)  
24 ymd('2005-04-03')  
25  
26  
27 # Let's try converting times to a date time format  
28 t1w <- mdy_hms('April 3, 2005 10:20:45')  
29 t1w2 <- ydm_hms('2005, 03 April 10:20:45')  
30 t1s <- ymd_hms('05/04/03 10:20:45')  
27:1 # Lubridate package
```

Console Terminal Jobs

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts

```
> d1w <- mdy(date1Words)  
> d1s <- mdy(date1Short)  
> d1n <- mdy(date1Numeric)  
> ymd('2005-04-03')  
[1] "2005-04-03"  
> |
```

Files Plots Package

Install Update

Name

System Library

- askpass
- assertthat
- backports
- base
- base64enc
- BH
- boot
- broom
- callr
- caret
- cellranger
- class
- cli
- clipr
- cluster

Now, what if the date is recorded in the ISO 8601 format? Well, I can use a different function. This is YMD, for year, month, day. Let's just run that and look in the console, and sure enough, it converted it to the right date. Hopefully this helps you appreciate the beauty of Lubridate and how much easier it is to work with dates using this Lubridate package.

The screenshot shows the RStudio interface with the following components:

- Script Editor:** Displays an R script with several commented-out sections and a few lines of code. One section is titled "# CALCULATIONS THAT INVOLVE DATES AND TIMES". Another section is titled "# USING TIME FUNCTIONS WITH BUSINESS DATA". The script ends with a comment "# Let's try converting times to a date time format".
- Console:** Shows the command-line interface with the same R script being run. It includes commands like `install.packages('lubridate')` and `t1w <- mdy_hms('April 3, 2005 10:20:45')`.
- Global Environment:** A table showing the current state of variables. It includes:

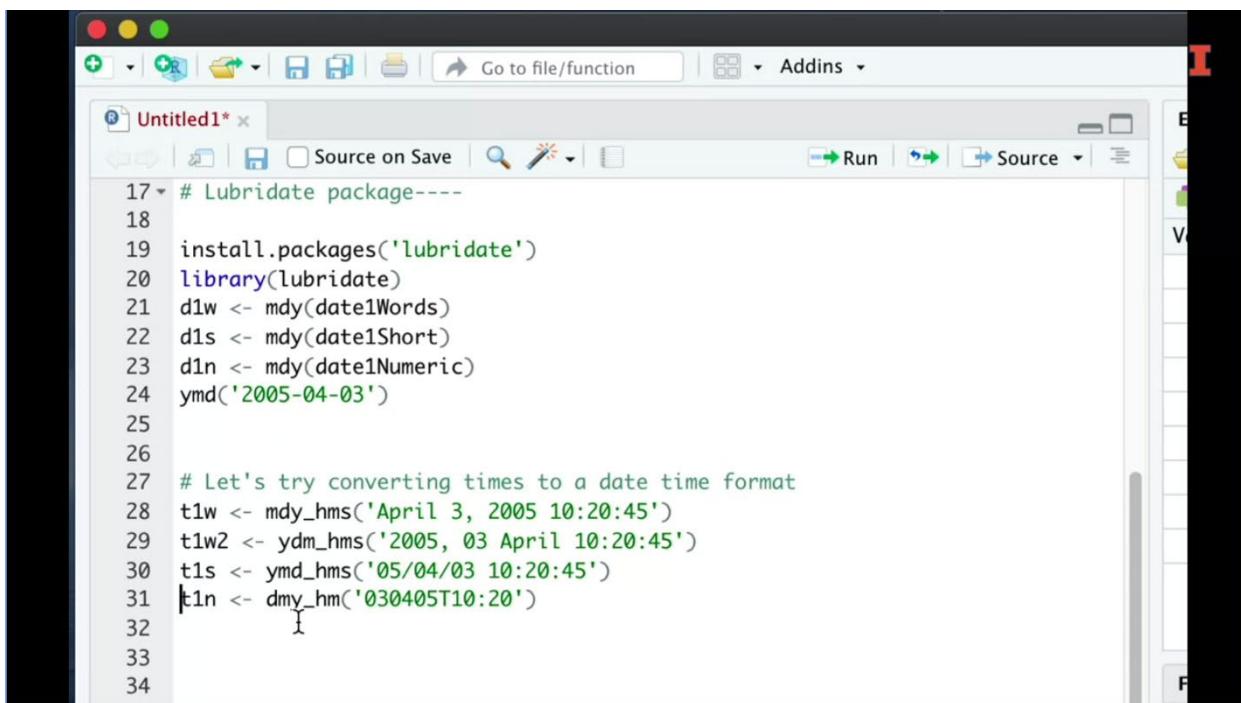
Name	Type	Value
dtn	POSIXt	2005-04-03
dls	POSIXt	2005-04-03
dlw	POSIXt	2005-04-03
date1Numeric	double	40385
date1Short	character	"04/03/05"
- File Explorer:** Shows the file structure with a file named "R Script 1" selected.
- Help:** A search bar at the top right.
- Project Navigator:** Shows a project named "Project (None)".

Now let's also talk about converting times to date time objects. Here is April 3rd, 2005, and then a random time of 10: 20: 45 seconds. There's a function in Lubridate that similar to the MDY function, but in this case we're going to put an underscore and add on HMS for hours, minutes, seconds. If I run that, I can look over in my environment and I can see that this object was created and it shows the ISO 8601 format of the date. Wonderful it worked.

The screenshot shows the RStudio interface with the following details:

- Top Bar:** Shows tabs for RStudio, File, Edit, Code, View, Editor, Plots, Data, Debug, Profile, Tools, Window, Help.
- Environment Tab:** Shows the global environment with variables like dIn, dIs, dIh, dateNumeric, and ymd.
- Console Tab:** Displays R code and its output. The code includes loading packages, setting the working directory, and various date-time conversion functions (lubridate, base, and dplyr). It also shows calculations involving dates and times, such as extracting elements from a data frame and calculating differences in dates and times.
- Plots Tab:** Shows a small preview of a plot titled "t1w".
- Data View Tab:** Displays a table with columns "Name", "Description", and "Version". The table lists numerous R packages, including base, BH, boot, broom, callr, caret, cellranger, class, cli, clipr, cluster, codetools, colorspace, compiler, crayon, curl, datasets, DBI, dplyr, digest, and many others.

What if I have the year, then the day, and then the month, and then the time, well, I use this YDM, easy to remember year, day, month and then underscore hour, minute, second. Let's look over and they Environment. Sure enough, it created the right tag, the right datatype. Very good.

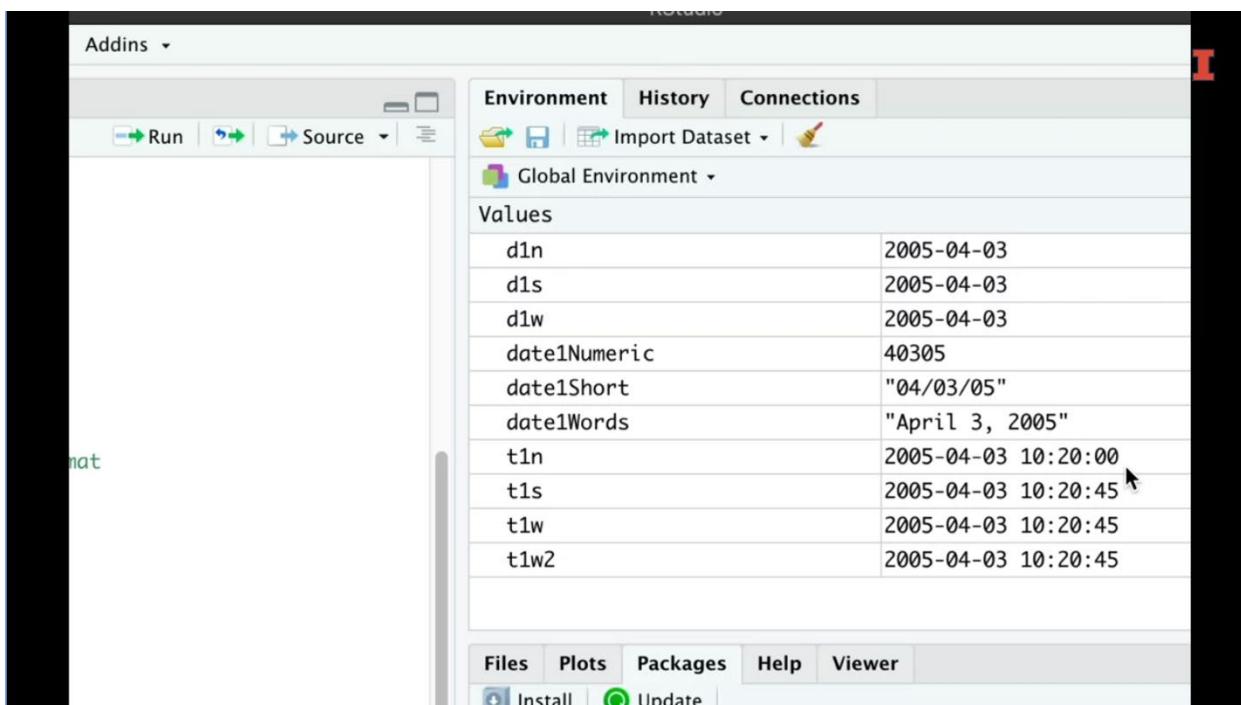


```

17 # Lubridate package----
18
19 install.packages('lubridate')
20 library(lubridate)
21 d1w <- mdy(date1Words)
22 d1s <- mdy(date1Short)
23 d1n <- mdy(date1Numeric)
24 ymd('2005-04-03')
25
26
27 # Let's try converting times to a date time format
28 t1w <- mdy_hms('April 3, 2005 10:20:45')
29 t1w2 <- ydm_hms('2005, 03 April 10:20:45')
30 t1s <- ymd_hms('05/04/03 10:20:45')
31 t1n <- dmy_hm('030405T10:20')
32
33
34

```

Let's see what happens if I have the year, month, day, and then hour, minute, second, I can use a different function, but it's easy to remember that run that. Once again, it converts it to the right data type. What if I have a situation where I have the day, the month, and the year, and then a t to indicate time, and then I just have the hours and the minutes and now seconds. You can probably guess at this point what the function will look like. There'll be DMY for day, month, year underscore, HM, for hour, minutes.

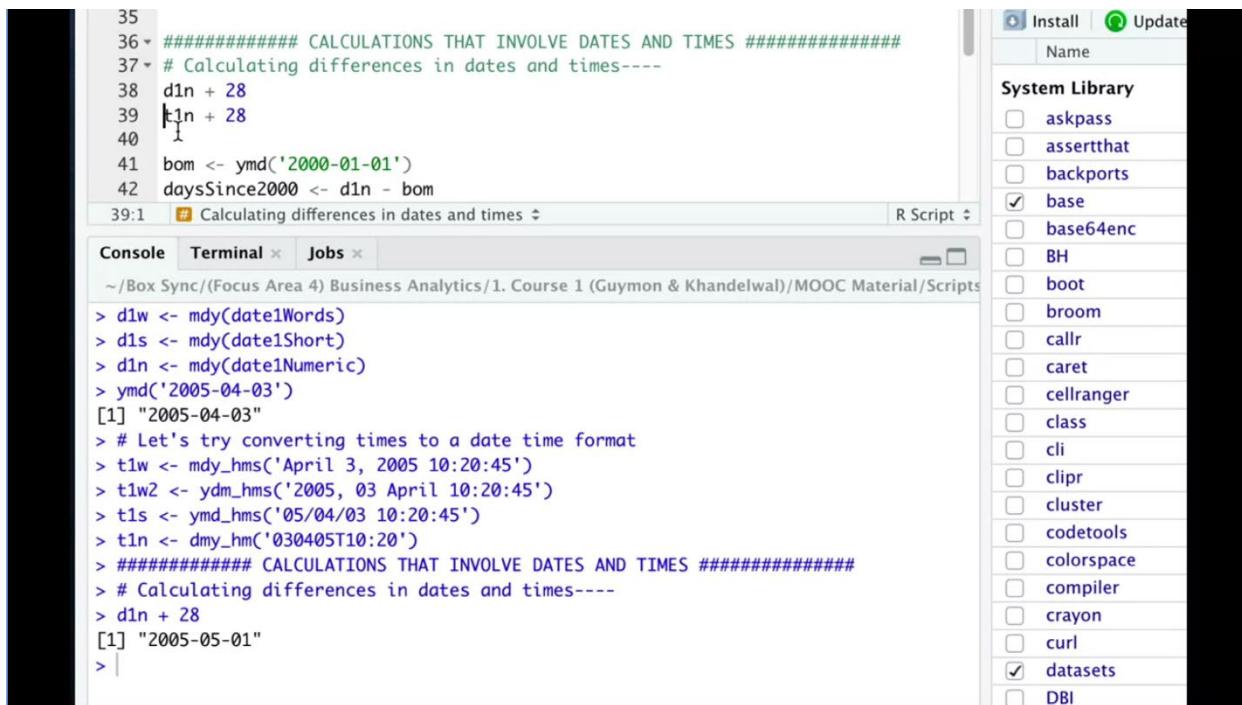


Values	
d1n	2005-04-03
d1s	2005-04-03
d1w	2005-04-03
date1Numeric	40305
date1Short	"04/03/05"
date1Words	"April 3, 2005"
t1n	2005-04-03 10:20:00
t1s	2005-04-03 10:20:45
t1w	2005-04-03 10:20:45
t1w2	2005-04-03 10:20:45

If I run that and look over in the environment, it creates a datetime object that has the date and the seconds are just rounded to the beginning of the minute. That's a brief introduction to converting strings and numeric values to date objects using the base package and the Lubridate package.

Lesson 3-7 Calculations with Dates

Lesson 3-7.1 Calculations with Dates



The screenshot shows an RStudio interface. On the left is the R Script pane containing R code. On the right is the Environment pane showing the 'System Library' with several packages listed. The code in the R Script pane is as follows:

```

35
36 # ##### CALCULATIONS THAT INVOLVE DATES AND TIMES #####
37 # Calculating differences in dates and times---
38 d1n + 28
39 | d1n + 28
40
41 bom <- ymd('2000-01-01')
42 daysSince2000 <- d1n - bom
39:1 # Calculating differences in dates and times

```

Below the R Script pane, the Console tab is active, showing the following R session history:

```

> d1w <- mdy(date1Words)
> d1s <- mdy(date1Short)
> d1n <- mdy(date1Numeric)
> ymd('2005-04-03')
[1] "2005-04-03"
> # Let's try converting times to a date time format
> t1w <- mdy_hms('April 3, 2005 10:20:45')
> t1w2 <- ydm_hms('2005, 03 April 10:20:45')
> t1s <- ymd_hms('05/04/03 10:20:45')
> t1n <- dmy_hm('030405T10:20')
> ##### CALCULATIONS THAT INVOLVE DATES AND TIMES #####
> # Calculating differences in dates and times---
> d1n + 28
[1] "2005-05-01"
>

```

The Environment pane on the right lists the 'System Library' with several packages checked (base, datasets) and others uncheckable.

In this lesson, we'll focus on making calculations that involve dates and times and then applying what we've learned about dates to business data. So, let's go ahead and take this date object, d1n which, if we look in the environment, this is a date object that represents April 3rd, 2005, and we just add a number to it. Add 28. You can see that in the console prints out a date, May 1st 2005 that is 28 days after that original date of April 3rd, 2005.

The screenshot shows an RStudio session with the following code and output:

```
install.packages("lubridate")
library(lubridate)
d1n <- mdy('04/03')
d1s <- mdy('04/03')
d1a <- mdy('04/03')
yndi <- ymd('2005-04-03')
d1n
d1s
d1a
dateNumeric
dateShort
# Let's try converting times to UTC
t1n <- mdy_hms('Apr 3, 2005 10:20:45')
t1s <- ymd_hms('05/04/03 10:20:45')
t1a <- ymd_hms('05/04/03 10:20:45')
t1n
t1s
t1a
```

Calculating differences in dates and times---

```
d1n + 28
t1n + 28
```

> # Calculating differences in dates and times---

```
> d1n + 28
[1] "2005-05-01"
> t1n + 28
[1] "2005-04-03 10:20:28 UTC"
```

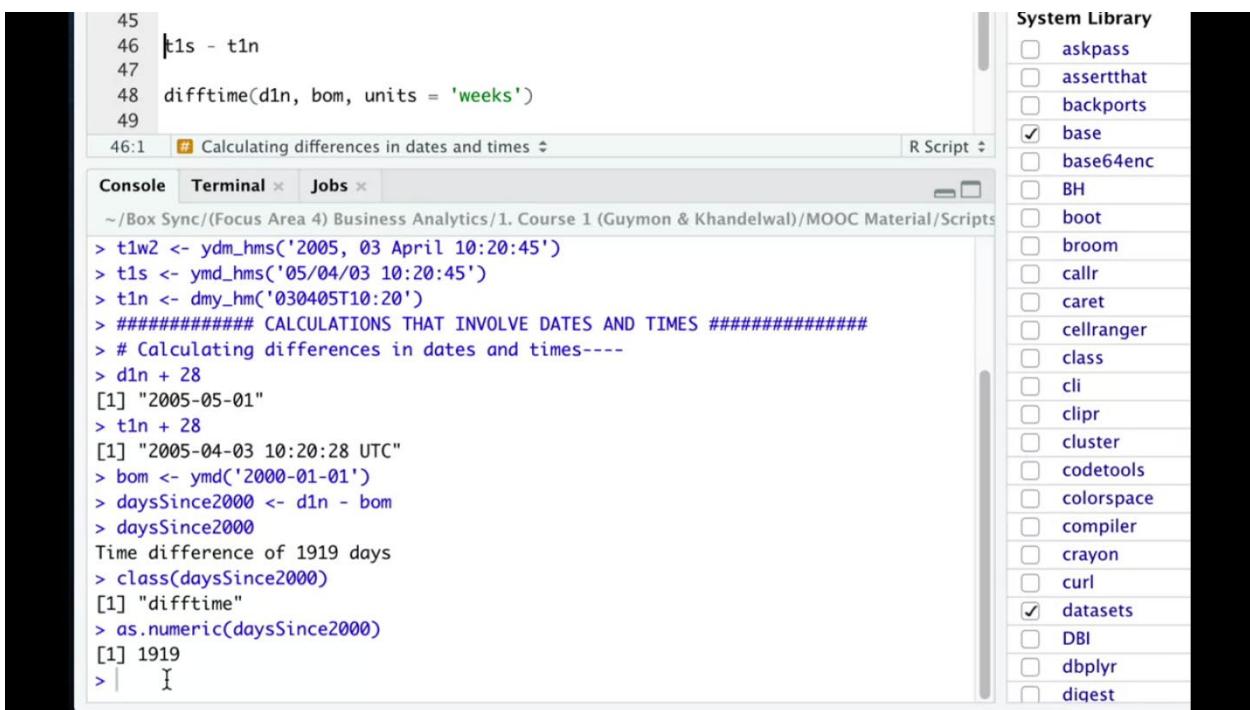
What if I take this `t1n` object, which is a time object, and I add 28 to it. Down on the console, that tells us some interesting things, and now it's the same day of the year, April 3rd 2005, but it has added 28 seconds to it, and so this is reflecting the difference in the way dates are stored versus times.

```
> d1s <- mdy(date1Short)
> d1n <- mdy(date1Numeric)
> ymd('2005-04-03')
[1] "2005-04-03"
> # Let's try converting times to a date time format
> t1w <- mdy_hms('April 3, 2005 10:20:45')
> t1w2 <- ydm_hms('2005, 03 April 10:20:45')
> t1s <- ymd_hms('05/04/03 10:20:45')
> t1n <- dmy_hm('030405T10:20')
> ##### CALCULATIONS THAT INVOLVE DATES AND TIMES #####
> # Calculating differences in dates and times----
> d1n + 28
[1] "2005-05-01"
> t1n + 28
[1] "2005-04-03 10:20:28 UTC" }
> |
```

Dates are stored as an integer, which indicates the number of days that have passed since the epic began, whereas time is recorded as the number of seconds that have passed. And so If you had 28 to it, then you're just adding 28 seconds versus days. Now, at times, we also have to worry about time zones, and so, UTC stands for coordinated universal time, and that represents what the time is at zero longitude. So, unless you specify time zones explicitly, it will assume that you're referring to zero longitude. It's beyond the scope of this lesson to talk about dealing with time zones, but that's something that you could investigate further if you're interested in doing so.

```
> t1s <- ymd_hms('05/04/03 10:20:45')
> t1n <- dmy_hm('030405T10:20')
> ##### CALCULATIONS THAT INVOLVE DATES AND
> # Calculating differences in dates and times----
> d1n + 28
[1] "2005-05-01"
> t1n + 28
[1] "2005-04-03 10:20:28 UTC"
> bom <- ymd('2000-01-01')
> daysSince2000 <- d1n - bom
> daysSince2000
Time difference of 1919 days
> |
```

Let's talk about other calculations that you might want to perform that involved dates. Let's create a new date object and will use the ymd function and convert the string here to January 1st, 2000, and we'll call that the beginning of the millennium. Then, let's make a calculation to find out the number of days that have passed since April 3rd, 2005 and January 1st, 2000. If I run that and then in the console look at daysSince 2000, it tells me a time difference of 1919 days, so that's pretty clear. If I look at the class. That is a difftime object.



```

45
46 t1s - t1n
47
48 difftime(d1n, bom, units = 'weeks')
49
46:1 # Calculating differences in dates and times

```

Console Terminal Jobs

```

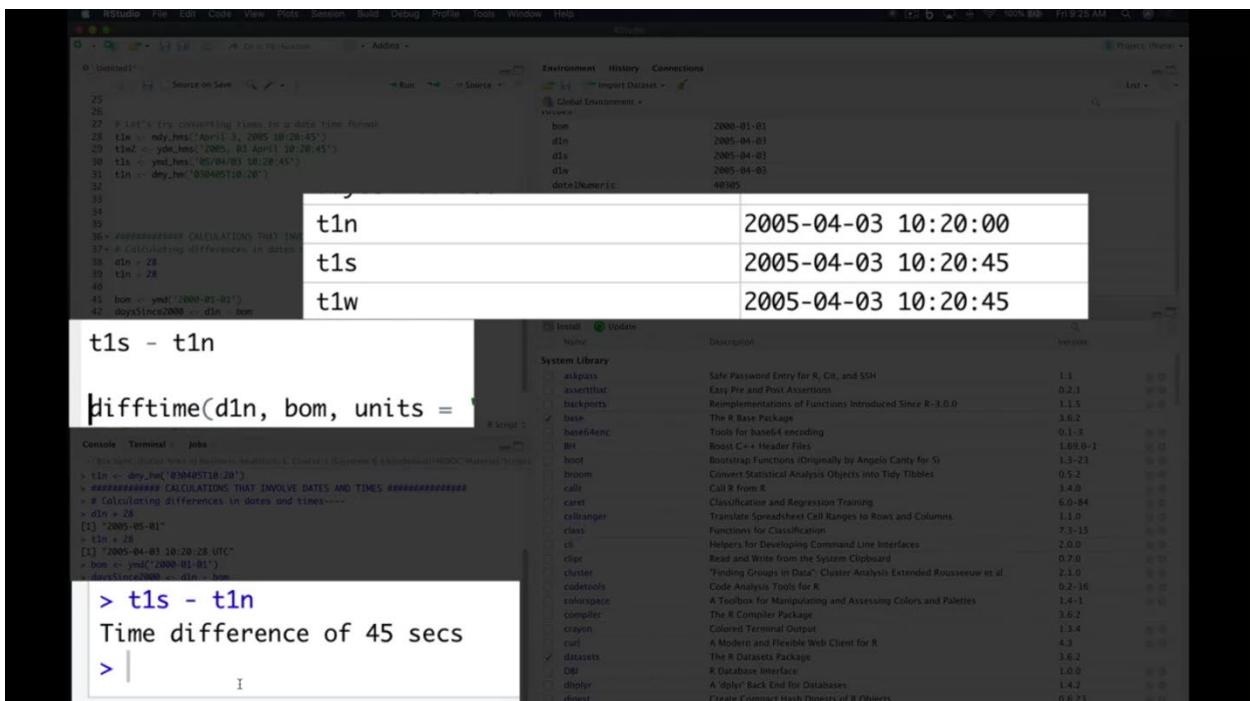
~ /Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts
> t1w2 <- ymd_hms('2005, 03 April 10:20:45')
> t1s <- ymd_hms('05/04/03 10:20:45')
> t1n <- dmy_hm('030405T10:20')
> ##### CALCULATIONS THAT INVOLVE DATES AND TIMES #####
> # Calculating differences in dates and times----
> d1n + 28
[1] "2005-05-01"
> t1n + 28
[1] "2005-04-03 10:20:28 UTC"
> bom <- ymd('2000-01-01')
> daysSince2000 <- d1n - bom
> daysSince2000
Time difference of 1919 days
> class(daysSince2000)
[1] "difftime"
> as.numeric(daysSince2000)
[1] 1919
> |

```

System Library

- askpass
- assertthat
- backports
- base
- base64enc
- BH
- boot
- broom
- callr
- caret
- cellranger
- class
- cli
- clipr
- cluster
- codetools
- colorspace
- compiler
- crayon
- curl
- datasets
- DBI
- dbplyr
- digest

So, it's not a numeric data type, and oftentimes will want to take the number of days that have passed and convert it into years or months or weeks. Or compare that to some other quantity of time. So, we can coerce that date, that difftime class to a numeric data type by using the as dot numeric functions. So if I do that, you can see in the console that it prints out just the integer of 1919.



```

25
26
27 R-Lint: try converting t1ns to a date-time format
28 t1w2 <- ymd_hms('April 1, 2005 10:20:45')
29 t1d <- ymd_hms('2005-04-01 10:20:45')
30 t1s <- ymd_hms('05/04/03 10:20:45')
31 t1n <- dmy_hm('030405T10:20')
32
33
34
35
36 ##### CALCULATIONS THAT INVOLVE DATES AND TIMES #####
37 # Calculating differences in dates
38 d1n + 28
39 t1n + 28
40
41 bom <- ymd('2000-01-01')
42 daysSince2000 <- d1n - bom

```

t1n	2005-04-03 10:20:00
t1s	2005-04-03 10:20:45
t1w	2005-04-03 10:20:45

t1s - t1n

```

difftime(d1n, bom, units =

```

Console Terminal Jobs

```

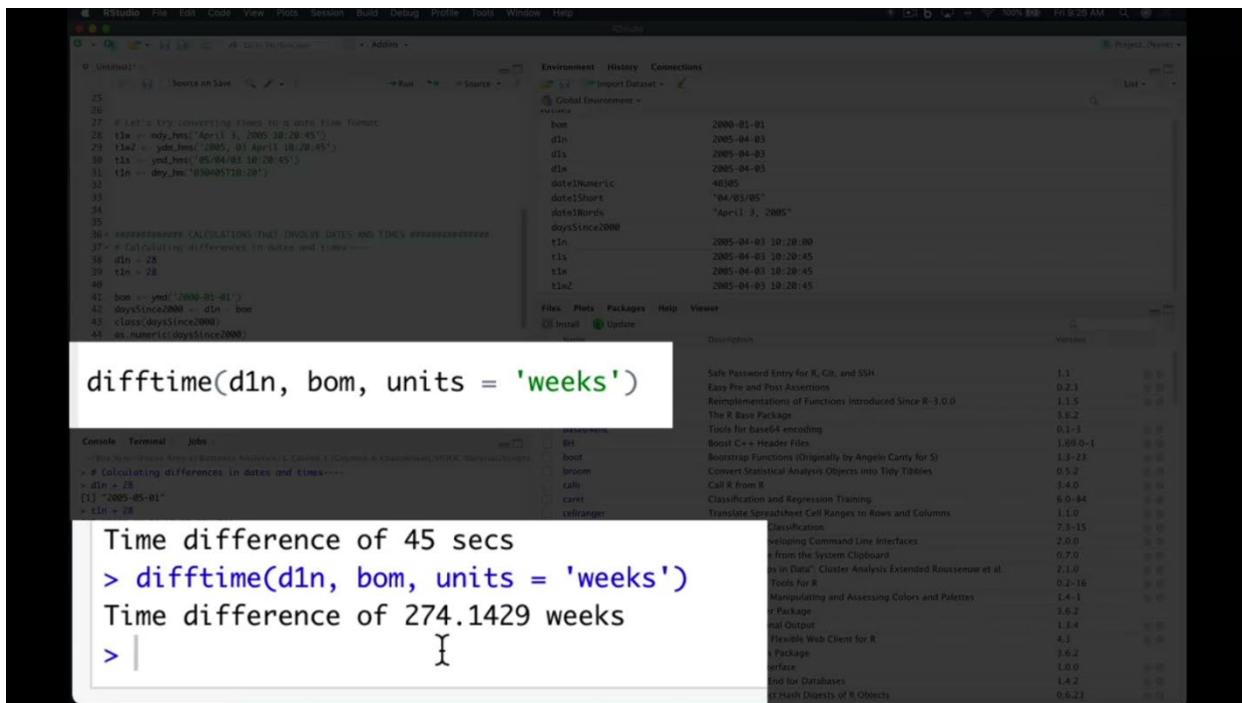
> t1s - t1n
Time difference of 45 secs
> |

```

System Library

Name	Description	Version
askpass	Safe Password Entry for R, Git, and SSH	1.1
assertthat	Easy Pre and Post Assertions	0.2.1
backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.5
base	The R Base Package	3.6.2
base64enc	Tools for base64 Encoding	0.1-3
BH	Boost C++ Header Files	1.69.0-1
boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-23
broom	Convert Statistical Analysis Objects into Tidy Tibbles	0.5.2
callr	Call R from R	3.4.0
caret	Classification and Regression Training	6.0-84
cellranger	Translate Spreadsheet Cell Ranges to Rows and Columns	1.1.0
class	Functions for Classification	7.3-15
cli	Helpers for Developing Command Line Interfaces	2.0.0
clipr	Read and Write from the System Clipboard	0.7.0
cluster	Finding Groups in Data: Cluster Analysis Extended Rousseeuw et al.	2.1.0
codetools	Code Analysis Tools for R	0.2-16
colorspace	A Toolbox for Manipulating and Assessing Colors and Palettes	1.4-1
compiler	The R Compiler Package	3.6.2
crayon	Colored Terminal Output	1.3.4
curl	A Modern and Flexible Web Client for R	4.2
datasets	The R Datasets Package	3.6.2
DBI	R Database Interface	1.0.0
dbplyr	A dplyr Back End for Databases	1.4.2
digest	Object Comparison Digits of R Objects	0.6.25

All right, let's go ahead and look at what happens if we take t1s, which is a time stamp of 10:20:45 seconds. And compare that to a time stamp of 10:20:00 seconds, both on April 3rd, 2005. If I run that, it tells me a difference of 45 seconds. So once again, this is reflecting the way that times are stored as the number of seconds that have passed, and so it automatically results to telling me the number of seconds.



The screenshot shows an RStudio interface with the following code in the console:

```

25
26 # Let's try converting times to a date-time format
27 t1s <- ymd_hms("April 3, 2005 10:20:45")
28 t1s2 <- ymd_hms("2005, 04/03 10:20:45")
29 t1s == ymd_hms("05/04/05 10:20:45")
30 t1s == dmy_hm("03/04/05 10:20:45")
31 t1s == dmy_hm("03/04/05T10:20")
32
33
34
35
36 # Calculations that involve dates and times
37 # Calculating differences in dates and times
38 d1n = 28
39 t1n = 28
40
41 bom <- ymd("2000-01-01")
42 daysSince2000 <- d1n - bom
43 class(daysSince2000)
44 as.numeric(daysSince2000)

```

The output shows the calculation of the difference between April 3, 2005, and January 1, 2000:

```

Time difference of 45 secs
> difftime(d1n, bom, units = 'weeks')
Time difference of 274.1429 weeks
>

```

The RStudio environment pane shows variables and their values:

Variable	Value
bom	2000-01-01
d1n	2005-04-03
t1s	2005-04-03
t1s2	2005-04-03
date1Numeric	40385
date1Short	"04/03/05"
date1Words	"April 3, 2005"
daysSince2000	2005-04-03 10:20:45
t1n	2005-04-03 10:20:45
t1s	2005-04-03 10:20:45
t1s2	2005-04-03 10:20:45

The packages pane lists available packages:

Description	Version
Safe Password Entry for R, C/C++, and SSH	1.1
Easy Pro and Post Assertions	0.2.1
Reimplementations of Functions Introduced Since R-3.0.0	1.1.5
The R Base Package	3.6.2
Tools for base64 encoding	0.1-3
Boost C++ Header Files	1.00.0-1
Bootstrap Functions (Originally by Angelino Camy for S)	1.3-23
Convert Statistical Analysis Objects into Tidy Tibbles	0.5.2
Call R from R	3.4.0
Classification and Regression Training	6.0-44
Translate Spreadsheet Cell Ranges to Rows and Columns	1.1.0
Classification	7.3-15
Developing Command-Line Interfaces	2.0.0
R from the System Clipboard	0.7.0
ts in Data: Cluster Analysis Extended Rousseeuw et al.	2.1.0
Tools for R	0.2-16
Manipulating and Assessing Colors and Palettes	1.4-1
R Package	3.6.2
mailOutput	1.3.4
Flexible Web Client for R	4.3
R Package	3.6.2
surface	1.0.0
Grid for Databases	1.4.2
externalDots of R Objects	0.6.23

Now, if I want to make a quick conversion, I can use a base function called `difftime`, and I can enter a beginning time and an end time, and it'll take this first time or date and subtract the second date from it. So in this case, I've got April 3rd, 2005 and January 1st, 2000, and we're going to calculate the difference in weeks. So, I'll go ahead and run that, and it tells me the difference is 274.1429 weeks. So, there are some calculations that involve dates and times that are often performed.

The screenshot shows the RStudio interface. On the left is the R Script pane containing R code for extracting date/time elements from a date/time object. On the right is the Environment pane showing objects t1s, t1w, and t1w2 with their creation times. Below the script pane is the Console pane showing the execution of the code. At the bottom is the Packages pane displaying the System Library.

```

55 year(d1n)
56 month(d1n)
57 month(d1n,label = T)
58 day(d1n)
59 wday(d1n)
60 wday(d1n,label = T)
61 hour(t1s)
62 minute(t1s)
63 second(t1s)
64
65 ##### USING TIME FUNCTIONS WITH BUSINESS DATA #####
66 # Read in items data and convert Time to date time type
67 # Calculate a new column: day of the week
68 # Calculate a new column: hour of the day
56:1  Extracting elements from a date/time : R Script

```

Console Terminal × Jobs ×

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts
[1] "2005-05-01"
> t1n + 28
[1] "2005-04-03 10:20:28 UTC"
> bom <- ymd('2000-01-01')
> daysSince2000 <- d1n - bom
> daysSince2000
Time difference of 1919 days
> class(daysSince2000)
[1] "difftime"
> as.numeric(daysSince2000)
[1] 1919
> t1s - t1n
Time difference of 45 secs
> difftime(d1n, bom, units = 'weeks')
Time difference of 274.1429 weeks
> year(d1n)
[1] 2005
>

```

Files Plots Packages Help Viewer

Name Description

System Library

- askpass
- assertthat
- backports
- base
- base64enc
- BH
- boot
- broom
- callr
- caret
- cellranger
- class
- cli
- clipr
- cluster
- codetools
- colorspace
- compiler
- crayon
- curl
- datasets
- DBI
- dbplyr
- digest

Now, another thing that is often done with dates and times is we'd like to extract certain elements from it. For instance, we've got this date of April 3, 2005 and perhaps we want to extract the year or the month through the day. Or if we have a time stamp, we may want to extract just the hour or the minute or the seconds. So, the lubridate package makes that very simple. We can simply call the year function and then enter a date object in there, and it returns the year 2005 or if I use the month function, it returns 4 for April.

The screenshot shows the RStudio interface. The R Script pane contains modified R code that includes the execution of the as.numeric(daysSince2000) command. The Environment pane shows the same objects t1s, t1w, and t1w2. The Packages pane displays the System Library.

```

55 year(d1n)
56 month(d1n)
57 month(d1n,label = T)
58 day(d1n)
59 wday(d1n)
60 wday(d1n,label = T)
61 hour(t1s)
62 minute(t1s)
63 second(t1s)
64
65 ##### USING TIME FUNCTIONS WITH BUSINESS DATA #####
66 # Read in items data and convert Time to date time type
67 # Calculate a new column: day of the week
68 # Calculate a new column: hour of the day
60:1  Extracting elements from a date/time : R Script

```

Console Terminal × Jobs ×

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts
> as.numeric(daysSince2000)
[1] 1919
> t1s - t1n
Time difference of 45 secs
> difftime(d1n, bom, units = 'weeks')
Time difference of 274.1429 weeks
> year(d1n)
[1] 2005
> month(d1n)
[1] 4
> month(d1n,label = T)
[1] Apr
12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
> day(d1n)
[1] 3
> wday(d1n)
[1] 1
>

```

Files Plots Packages Help Viewer

Name Description

System Library

- askpass
- assertthat
- backports
- base
- base64enc
- BH
- boot
- broom
- callr
- caret
- cellranger
- class
- cli
- clipr
- cluster
- codetools
- colorspace
- compiler
- crayon
- curl
- datasets
- DBI
- dbplyr
- digest

If I want to give it a label of April, I can use this label argument run that, and it returns an abbreviated month name there. If you look into that function more, you can tell it to return the full name if you want. And then it also tells us that this is a special data type now, which we'll talk about later. But it's an ordered factor data type. All right, I can extract the day by using the day function that tells me the day of the month. Three.

The screenshot shows the RStudio interface. On the left, the R Script pane displays R code for extracting date components from a data frame. The Console pane shows the results of running this code, including the month names (e.g., Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, ... < Dec) and their corresponding labels (e.g., 1, 2, 3, 4). On the right, the Environment pane shows four variables: t1n, t1s, t1w, and t1w2, each containing a single value: "2005-04-03 10:20:00". Below the environment is a package browser titled "System Library" with a list of available packages.

```

55 year(d1n)
56 month(d1n)
57 month(d1n,label = T)
58 day(d1n)
59 wday(d1n)
60 wday(d1n,label = T)
61 hour(t1s)
62 minute(t1s)
63 second(t1s)
64
65 ##### USING TIME FUNCTIONS WITH BUSINESS DATA #####
66 # Read in items data and convert Time to date time type
67 # Calculate a new column: day of the week
68 # Calculate a new column: hour of the day
69 ##### USING TIME FUNCTIONS WITH BUSINESS DATA #####

```

```

[1] 4
> month(d1n,label = T)
[1] Apr
12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
> day(d1n)
[1] 3
> wday(d1n)
[1] 1
> wday(d1n,label = T)
[1] Sun
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
> hour(t1s)
[1] 10
> minute(t1s)
[1] 20
> second(t1s)
[1] 45
>

```

If I want the day of the week, I can use the wday function, and that's one. And if I'm not sure if that's a Sunday or on Monday, I can add a label in there, and that tells me that it's a Sunday, and again there are more options here. You can go in and set a parameter that sets Monday as the first day of the week if you want, but the default is to use Sunday as the first day of the week. All right, what if I've got a time stamp and I want to extract the hour. Well there's an hour function from want to extract the minute, you can use the minute function, and second, to extract the seconds. So, lubricate makes it very intuitive for dealing with dates and extracting components of a time or date that would be useful.

The screenshot shows the RStudio interface. On the left, the R Script pane displays R code for reading and summarizing time data from a CSV file. The code includes functions for extracting hours, minutes, and seconds from a timestamp column, and then creating new columns for the day of the week and hour of the day. The R Console pane shows the results of these operations, including the levels of the day-of-week factor and the summary statistics for the new columns.

```

61 hour(t1s)
62 minute(t1s)
63 second(t1s)
64
65 * ##### USING TIME FUNCTIONS WITH BUSINESS DATA #####
66 # Read in items data and convert Time to date time type---
67 df <- read.csv('jan17Items.csv'
68           , stringsAsFactors = F
69           )
70 class(df$Time)
71 summary(df$Time)
72 df$Time <- ymd_hms(df$Time)
73 class(df$Time)
74 summary(df$Time)
75 * # Calculate a new column: day of the week
76 * # Calculate a new column: hour of the day
77:  Read in items data and convert Time to date time type

```

Console | Terminal | Jobs |

```

> wday(d1n,label = T)
[1] Sun
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
> hour(t1s)
[1] 10
> minute(t1s)
[1] 20
> second(t1s)
[1] 45
> ##### USING TIME FUNCTIONS WITH BUSINESS DATA #####
> # Read in items data and convert Time to date time type---
> df <- read.csv('jan17Items.csv'
+           , stringsAsFactors = F
+           )
> View(df)
> class(df$Time)
[1] "character"
>

```

The right side of the interface shows the RStudio file browser. It lists various R scripts and CSV files used in the course, such as 'conditionalStatements.R', 'dates.R', and 'feb17Items.csv'. The current working directory is 'Focus Area 4 Business Analytics / 1. Course 1 (Guymon & Khandelwal) / MOOC Material / Scripts'.

Let's take what we've learned about dates and illustrate how to apply it to business data. So, let's first read in the data. We're going to use the jan17Items data. We'll read that in as we normally do, and let's go ahead and investigate this data set. If I open it, I can see the first column is time, and it's obviously a time stamp, right? It's got a date and then a time stamp in there. But if I look at the type, it is currently a character type, and I like the way it's displayed. But as a character type, I wouldn't be able to make any calculations or extract the day of the week or the year and so forth. Also, if I try to perform a summary calculation to get some additional insight, it doesn't really tell me much detail, just tells me how many observations, there are basically 8899 and then tells me it's all character.

The screenshot shows an RStudio interface. On the left is the R Script pane with the following code:

```

61 hour(t1s)
62 minute(t1s)
63 second(t1s)
64
65 ##### USING TIME FUNCTIONS WITH BUSINESS DATA #####
66 # Read in items data and convert Time to date time type----
67 df <- read.csv('jan17Items.csv'
68 , stringsAsFactors = F
69 )
70 class(df$Time)

df$Time <- ymd_hms(df$Time)
class(df$Time)
summary(df$Time)
76 df$wday <- wday(df$Time, label = T)
75:1 # Calculate a new column: day of the week

Console Terminal Jobs

```

The right side shows the RStudio file browser with a tree view of files and a status bar indicating the current focus area.

The R Console pane displays the output of the summary command:

```

8899 character character
> df$Time <- ymd_hms(df$Time)
> class(df$Time)
[1] "POSIXct" "POSIXt"
> summary(df$Time)
      Min.   1st Qu.   Median   Max. 
"2017-01-03 05:06:00" "2017-01-11 11:12:00" "2017-01-18 18:45:00" 
I       Mean           3rd Qu.          Max. 
"2017-01-18 00:16:19" "2017-01-25 18:06:00" "2017-01-31 20:12:00" 
>

```

So, let's go ahead and convert that time column to time type. And so, I'll use the `ymd_hms` function from the `lubricate` package and refer to that column, and I'll sign it back to itself. So, if I run that and now look at the class of that column or the type, it's a `POSIXct` which basically means a time type. And now, if I look at the summary, it tells me the minimum time, which in this case is January 3, 2017 at 5:06am. And I can see different parts of the distribution. But the max time is January 31st, 2017 at 10:12 pm. So, it uses that 24 hour time convention.

The screenshot shows an RStudio interface. On the left is the R Script pane with the following code:

```

62 minute(t1s)
63 second(t1s)
64
65 ##### USING TIME FUNCTIONS WITH BUSINESS DATA #####
66 # Read in items data and convert Time to date time type----
67 df <- read.csv('jan17Items.csv'
68 , stringsAsFactors = F
69 )
70 class(df$Time)
71 summary(df$Time)
72 df$Time <- ymd_hms(df$Time)
73 class(df$Time)
74 summary(df$Time)
75 # Calculate a new column: day of the week----
76 df$wday <- wday(df$Time, label = T)
77 class(df$wday)
78 summary(df$wday)
79
80 # Calculate a new column: hour of the day
75:1 # Calculate a new column: day of the week

```

All right, so again as a date, time object. In this case, we can do a lot more interesting calculations. All right, well, now, let's say maybe we want to investigate business fluctuations and trends based on day of the week. Let's calculate a new column day of the week and we can do it based on that Time column. So, I'm going to use the wday function from the lubridate package, and I'll tell it to perform this function on every observation in the Time column of the df dataset.

```
Length     Class      Mode
 8899 character character
> df$Time <- ymd_hms(df$Time)
> class(df$Time)
[1] "POSIXct" "POSIXt"
> summary(df$Time)
   Min.           1st Qu.          Median
"2017-01-03 05:06:00" "2017-01-11 11:12:00" "2017-01-18 18:45:00"
   Mean           3rd Qu.          Max.
"2017-01-18 00:16:19" "2017-01-25 18:06:00" "2017-01-31 20:12:00"
> # Calculate a new column: day of the week----
> df$wday <- wday(df$Time, label = T)
> class(df$wday)
[1] "ordered" "factor"
> summary(df$wday)
  Sun Mon Tue Wed Thu Fri Sat
  0 1022 1311 1465 1691 1671 1739
> |
```

Let's go ahead and add the label to it, so I'll run that. And now let's look at the type so we use the class function tells me it's an ordered factor type. Now, let's go ahead and look at a summary of this column so I'll run that. And wonderful, this is now telling me how many observations there are in this data set for each of these days of the week and notice that it orders those in a meaningful order. It's not alphabetic order, but it's as the days go, which is really helpful. All right, so again, that's a benefit of having a date time format in our Time column and using the lubridate package. It makes it really easy to extract this information.

The screenshot shows the RStudio interface. On the left is the R Script pane containing R code. On the right is the Environment pane showing the current workspace. A file browser pane is also visible.

```

79
80 # Calculate a new column: hour of the day---
81 df$hod <- hour(df$Time)
82 class(df$hod)
83 summary(df$hod)
84
84:1  Calculate a new column: hour of the day R Script
Console Terminal Jobs
~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts
"2017-01-03 05:06:00" "2017-01-11 11:12:00" "2017-01-18 18:45:00"
Mean            3rd Qu.        Max.
"2017-01-18 00:16:19" "2017-01-25 18:06:00" "2017-01-31 20:12:00"
> # Calculate a new column: day of the week---
> df$wday <- wday(df$Time, label = T)
> class(df$wday)
[1] "ordered" "factor"
> summary(df$wday)
Sun Mon Tue Wed Thu Fri Sat
 0 1022 1311 1465 1691 1671 1739
> # Calculate a new column: hour of the day---
> df$hod <- hour(df$Time)
> class(df$hod)
[1] "integer"
> summary(df$hod)
   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 2.00  12.00  15.00  15.01  18.00  21.00
>

```

How about let's calculate a new column? Maybe we're interested in looking at sales trends based on hour of the day. So, I can create a new column. I'll call it hod for hour of the day, and I'll use the hour function in the lubridate and I apply it to the Time column. And let's look at the data type here. This is an integer, and if I look at a summary of the hour of the day, it tells me the minimum hour is two. So, 2 a.m. The maximum hour is 21 or 11 p.m. So with that, you should be able to get started performing a lot of date time calculations using business data.

Lesson 3-8 Factors

Lesson 3-8.1 Factors

The screenshot shows the RStudio interface. The top-left pane displays an R script named "Untitled1" with the following code:

```

1 ##### WORKING WITH FACTORS IN R #####
2 # Coercing to factor---
3 x <- c('fox', 'hound', 'hound')
4 summary(x)
5
6
7
8
9
10 # You may have noticed the ordered factor type when working with dates---
11 # Read in the jan17Items data
12 df <- read.csv('jan17Items.csv',
13                 , stringsAsFactors = F
14                 )
15 df$Time <- lubridate::ymd_hms(df$Time) # Convert Time to a time type
10:1 You may have noticed the ordered factor type when working with dates

```

The bottom-left pane shows the R Console output:

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts & Slides
> ##### WORKING WITH FACTORS IN R #####
> # Coercing to factor---
> x <- c('fox', 'hound', 'hound')
> summary(x)
  Length   Class    Mode
      3 character character
>

```

The right side of the interface includes the Global Environment pane (showing 'x') and the Files pane, which lists files like ".Rhistory", "conditionalStatements.R", and "dates.R" under "Area 4 Business Analytics".

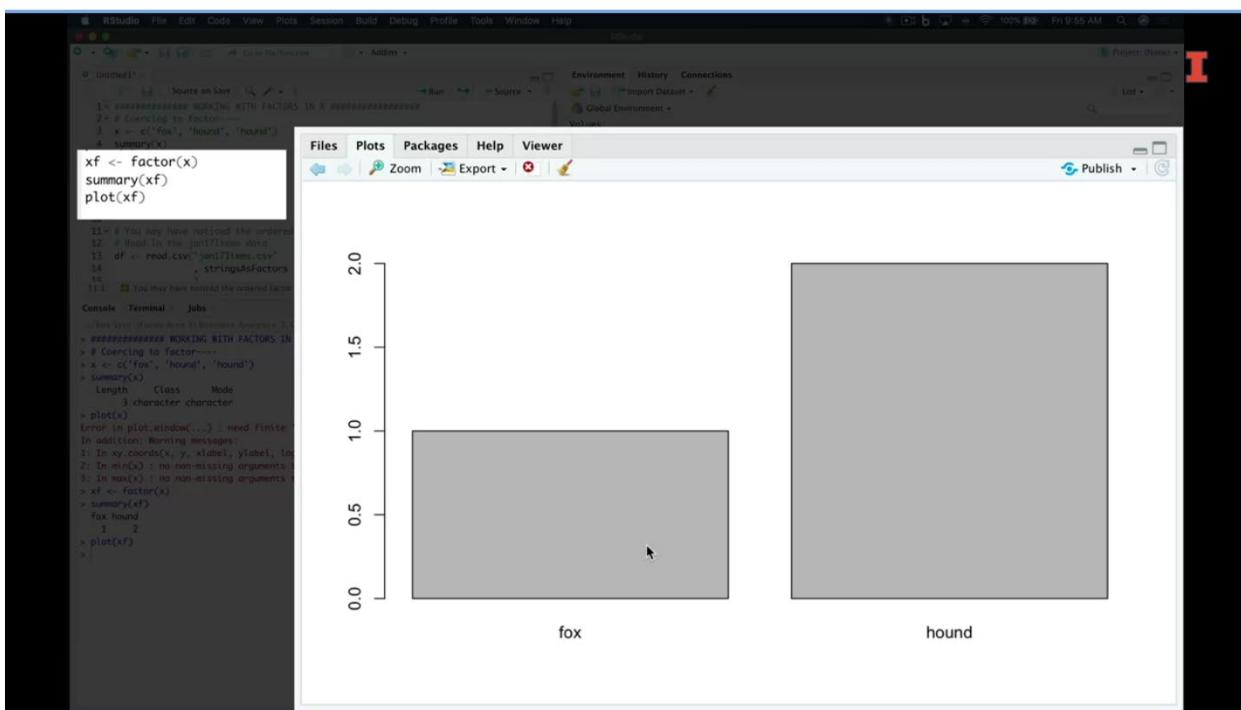
In this lesson, you're going to learn about the factor data type now before we talk about the factor data type using business data. Let's just look at it in terms of a simple vector. So let's say I have this vector here X, which is a character vector that has the word fox and hound in it, and Hound is in there twice. Now, if this were a much longer vector, I might be interested in knowing how many times each word appears. And I may also be interested in knowing what the different words are in this character strength. So it would be nice if I could use the summary function and if it would tell me that information.

The screenshot shows the RStudio interface. In the Global Environment pane, there is a table with two rows:

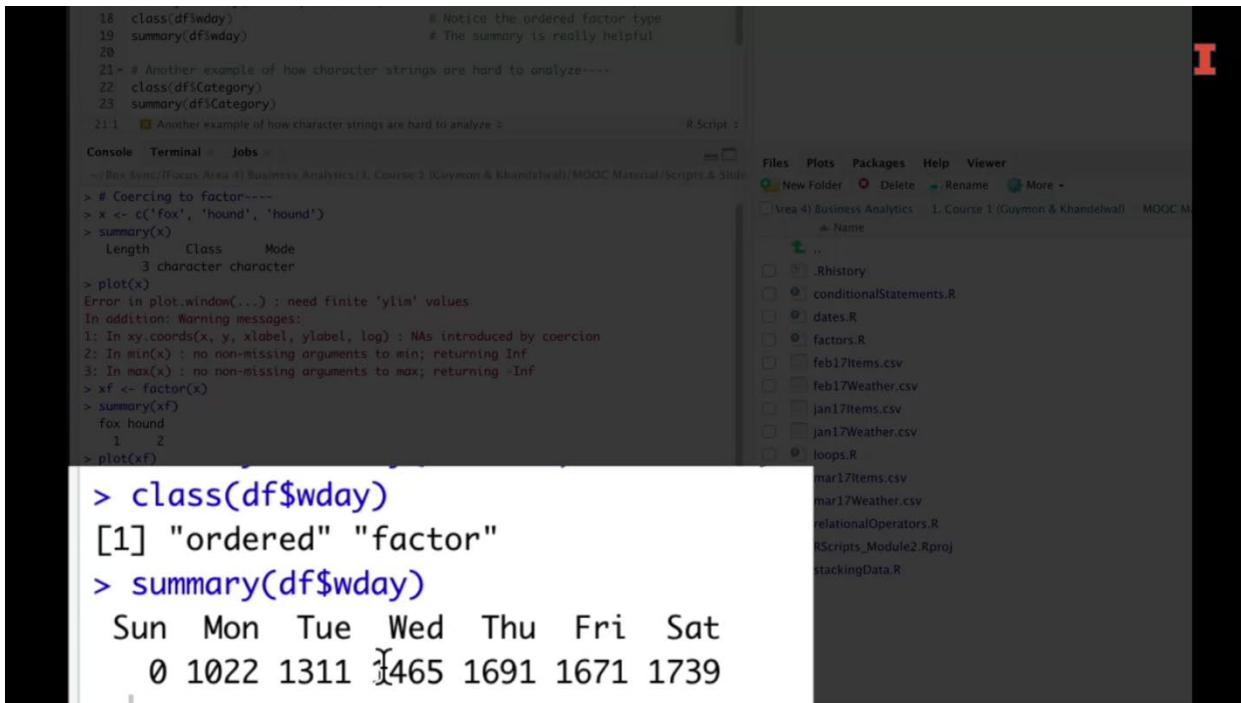
Values	
x	chr [1:3] "fox" "hound" "hound"
xf	Factor w/ 2 levels "fox", "hound": 1 2 2

A tooltip box is overlaid on the screen, containing the text: "A factor type displays strings, but those strings are internally represented as integers so that calculations can be performed with them." The RStudio menu bar at the top includes Files, Plots, Packages, Help, and Viewer, along with various icons for file operations.

But for a character string, it only tells me how many observations there are. Also, if I were to try and plot this vector, it couldn't do it because it is a character. String doesn't know what to plot. So let's go ahead and convert this to a factor, and we can coerce it to a factor data type by using the factor function. And then I'll put X in there, and now you can see I've got this XF object, and it tells me it is a factor. And it has two levels, fox and Hound. And then it also gives me integers indicating which level each of these words means. So that's what a factor is.



It's a combination of a character string for display purposes so that we can interpret it as humans as well as a numeric integer value, so that we can do some analysis with it. So now if I look at a summary of this ex FDA object, it tells me that foxes in their wants and hound is in there twice, and I can even plot it. And it gives me a history Graham that indicates the frequency with which each word or each level of the factor appears. All right, so now let's talk about factors in the context of business data. So let's go ahead and read.



In this January 17 items C S v file, and you may have already noticed in the lesson on dates that if we convert time the time column to a date and then create a new column W day for weekday, it will be an ordered factor. All right, so look at the class. It tells me it's an ordered factor, and if I do a summary of that, it tells me how many times each day of the week appears and it puts it in the right order for me. Now let's turn again to this January 17 items data set and let's look at the category column.

```

summary(df$Category)
Sun Mon Tue Wed Thu Fri Sat
 0 1022 1311 1465 1691 1671 1739
> # Another example of how character strings are hard to analyze----
> class(df$Category)
[1] "character"
> summary(df$Category)
  Length   Class    Mode
  8899 character character
> unique(df$Category)
[1] "Glass Bottle"           "Lamb Chops"
[3] "Salmon and Wheat Bran Salad" "Fountain"
[5] "Aubergine_and Chickpea Vindaloo" "Beef"
[7] "Chicken"                 "Beef and Apple Burgers"
[9] "Naan"                     "general"
[11] "Chutney"                  "Yogurt"
[13] "Rice"                      "Lamb"
[15] "Beef and Broccoli"        "Roll"
[17] "Beef Stew"                "Pork"
[19] "Curry"                     "Chips"
[21] "Non Food"
>
```

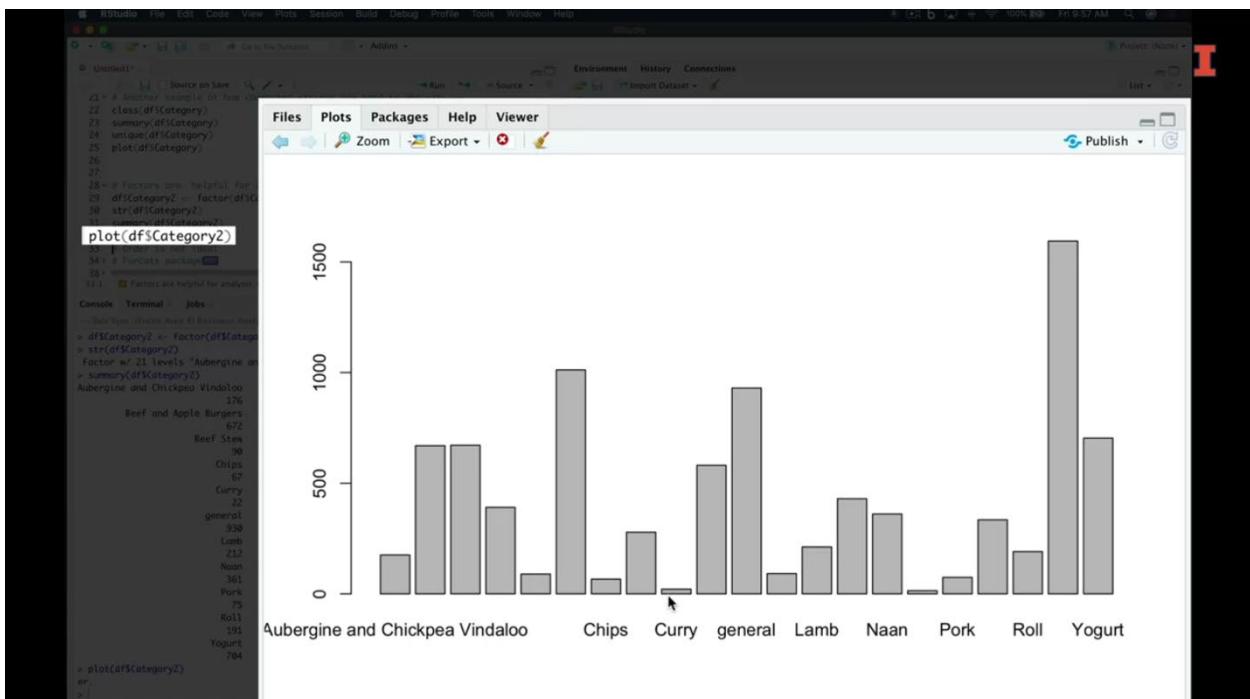
If we look at the class or the type of this column, it is a character string. And so if I try to look at the different categories in my data, it's not going to tell me anything. I can use this unique function, and it prints out a list of the different unique categories that show up in this data set. And so that's helpful, and I can see that there are 21 there, but if I try to plot it, nothing will happen.

```

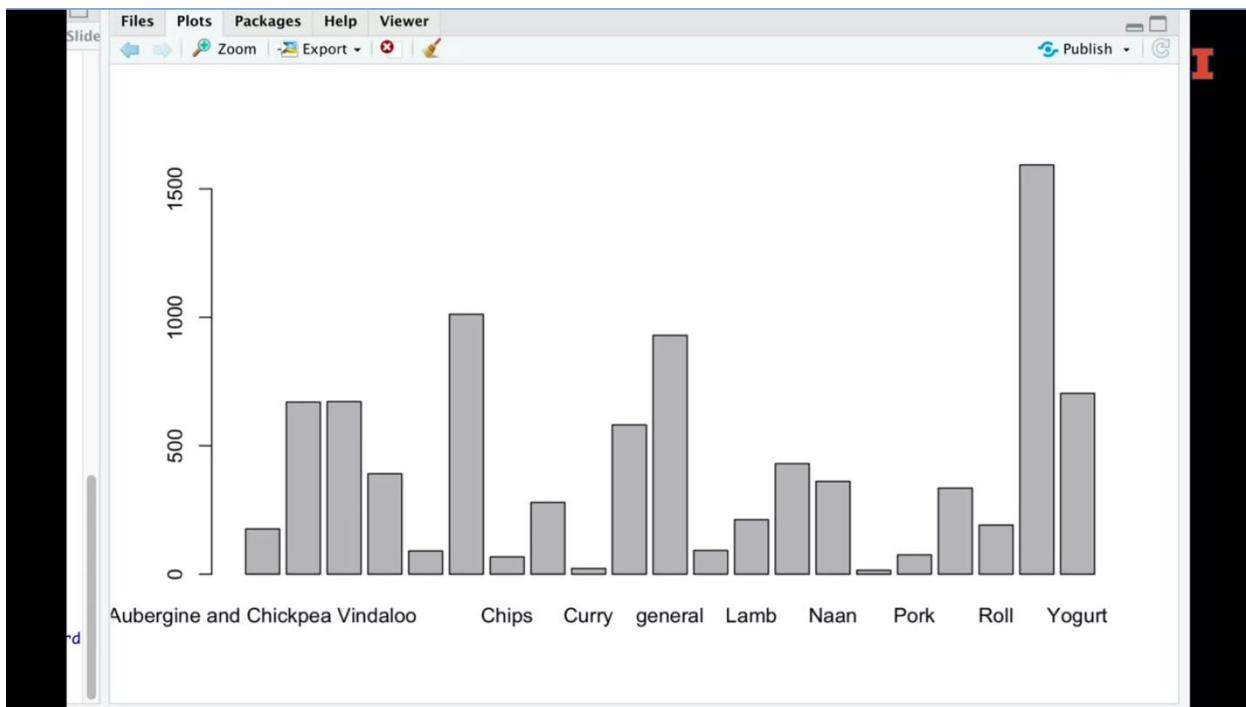
21> # Another example of how character strings are hard to analyze-----
22 class(df$Category)
23 summary(df$Category)
24 unique(df$Category)
25 plot(df$Category)
26
27 df$category2 <- factor(df$category)
28 str(df$Category2)
29 summary(df$Category2)
30 plot(df$Category2)
31 # Factors are helpful for analysis
32
33 > # Another example of how character strings are hard to analyze-----
34 > class(df$Category)
35 > summary(df$Category)
36 > unique(df$Category)
37 > plot(df$Category)
38 > length(df$Category)
39 > class(df$Category)
40 > mode(df$Category)
41 > nlevels(df$Category)
42 > levels(df$Category)
43 > str(df$Category)
44 > summary(df$Category)
45 > unique(df$Category)
46 > plot(df$Category)
47 > # Factors are helpful for analysis-----
48 > df$Category2 <- factor(df$Category)      # Create a new factor column for comparison
49 > str(df$Category2)                      # We can see the number of levels
50 Factor w/ 21 levels "Aubergine and Chickpea Vindaloo",...

```

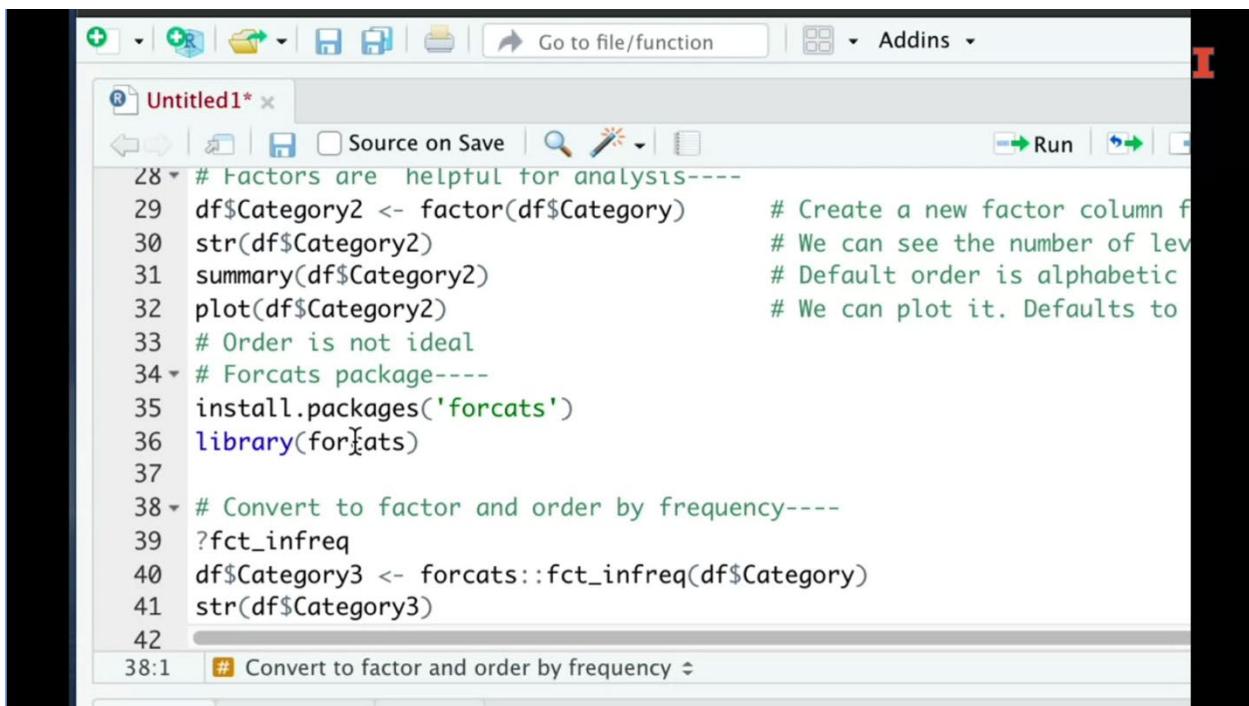
So let's go ahead and create a new column category two, and it will be a factored version of the original category column. Alright, so we're will create that column. And now if I look at the structure of that column, it tells me that it's a factor with 21 levels, and it tells me one of those levels and gives me some of the integers associated with those factor levels.



Now, if I look at the summary, it prints out a list of each level, and the order is alphabetical, so you can see it starts with a and ends with y. Now I can plot it, and once again, this is helpful. It's easy to see the most frequently occurring level of that factor, as well as what would be occurring least often. So while this is helpful, it's not ideal because the order could be improved. Typically, we want to order things in a meaningful way, ascending or descending.



So this is where the Four Cats package is helpful now. There are some base functions that you can use for dealing with actors and ordering them correctly, but it's a lot more verbose. And so this four Cats package has some functions that make it really easy to do some common tasks with factors. And this will be really helpful when you get more into visualization, because the order of the factors will influence the orders of the bars and the history. Graham, for instance.



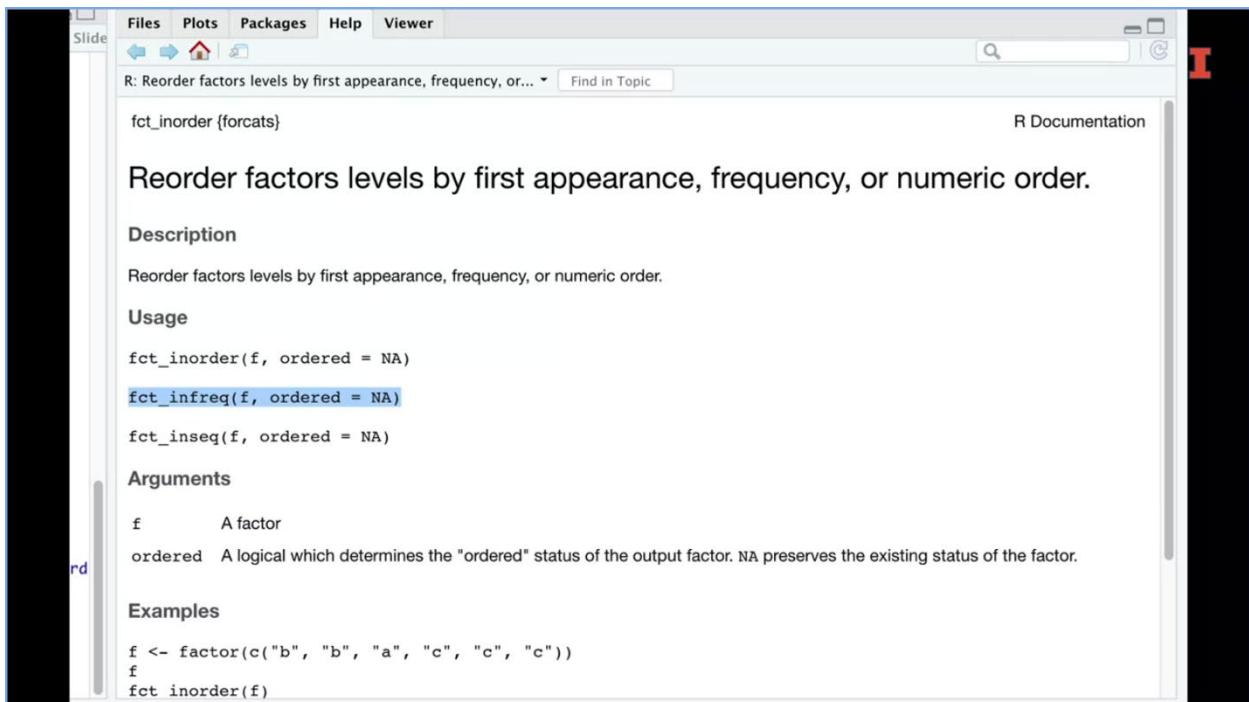
The screenshot shows an RStudio interface with a script editor window titled "Untitled1*". The code demonstrates how to convert a character vector into a factor and then reorder its levels based on frequency. The code is as follows:

```

28 # Factors are helpful for analysis---
29 df$Category2 <- factor(df$Category)          # Create a new factor column f
30 str(df$Category2)                            # We can see the number of lev
31 summary(df$Category2)                       # Default order is alphabetic
32 plot(df$Category2)                          # We can plot it. Defaults to
33 # Order is not ideal
34 # Forcats package---
35 install.packages('forcats')                 # Convert to factor and order by frequency---
36 library(forcats)
37
38 # Convert to factor and order by frequency---
39 ?fct_infreq
40 df$Category3 <- forcats::fct_infreq(df$Category)
41 str(df$Category3)
42
43:1 # Convert to factor and order by frequency

```

All right, so if you haven't installed the four Cats package. Go ahead and install that. You can do that using the installed packages or by going to the package pain and searching for four cats and installing that once you've installed it. Go ahead and load it by calling the library function and entering four cats in there and now you'll have access to all of the functions in that package. Let's start by talking about how to convert a character string vector into a factor vector and ordering it by frequency.



The screenshot shows the R Documentation page for the `fct_inorder` function. The page includes the following sections:

- Description**: Reorders factors levels by first appearance, frequency, or numeric order.
- Usage**: `fct_inorder(f, ordered = NA)`, `fct_infreq(f, ordered = NA)`, `fct_inseq(f, ordered = NA)`.
- Arguments**:
 - `f`: A factor.
 - `ordered`: A logical which determines the "ordered" status of the output factor. `NA` preserves the existing status of the factor.
- Examples**:


```
f <- factor(c("b", "b", "a", "c", "c", "c"))
f
fct_inorder(f)
```

So the function that you should read about if you want to read in more detail is the `fct_infreq`. And you can see that this is to reorder factor levels by first appearance, frequency or numeric order. So there are a variety of ways in which you could order the factors. We're going to use this in `freak` version of it, and so will simply create a new column `Category3` for comparison so we can compare it to `Category2` into the original category, and we'll call this function.

The screenshot shows the RStudio interface with the following details:

- Script Editor:** An R script named "Untitled1.R" is open, containing code related to factor reordering. A specific line of code, `?fct_infreq`, is highlighted with a yellow box.
- Console:** The output of the script execution is shown, including a summary of the `Category2` factor levels and their counts.
- Environment:** The sidebar shows the global environment with objects `df` and `xf`.
- Help:** A tooltip for the `fct_infreq` function is displayed, stating "R: Reorder factors levels by first app".
- Text:** A large text area at the bottom right contains the placeholder text "Reorder factors le".

```

30 str(df$Category2)
31 summary(df$Category2)
32 plot(df$Category2)
33 # Order is not ideal
34 # Forcats package----
35 install.packages('forcats')
36 library(forcats)
37
38 # Convert to factor and order by frequency
39 ?fct_infreq
40 df$Category3 <- forcats::fct_infreq(df$Category)
41 str(df$Category3)
42 summary(df$Category3)
43 plot(df$Category3)
44
45 Convert to factor and order by frequency

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts & Slides

```

> summary(df$Category2)      # Default order is alphabetic
Aubergine and Chickpea Vindaloo      Beef
                                         176      670
Beef and Apple Burgers      Beef and Broccoli
                                         672      391
Beef Stew      Chicken
                                         90      1012
Chips      Chutney

```

And just as a reminder if this library isn't loaded or if there happens to be a name space issue, meaning there's another package that has this function name in it. Then this is telling it that we want to use the function from the `FourCats` package. So we're simply going to call that function and then enter the column name that we're converting to a factor, which is just that category column. So I'll do that.

The screenshot shows the RStudio interface. On the left, the R Script pane contains R code for summarizing a dataset by category. The Console pane displays the resulting summary output, which is a frequency table of food items. The right side of the interface shows the Help pane for the `fct_inorder` function from the `forcats` package.

```

41 str(df$Category3)
42 summary(df$Category3)
43 plot(df$Category3)
44
45 # You can lump least frequent levels together
46
47 # Convert to factor and order by frequency
48 fct_infreq
49 df$Category3 <- forcats::fct_infreq(df$Category)
50 str(df$Category3)
Factor w/ 21 levels "Salmon and Wheat Bran Salad",...: 16 8 1 7 15 6 6 2 6 5 ...
51 summary(df$Category3)
  Salmon and Wheat Bran Salad      Chicken
  1594                           1012
  general                         Yogurt
  930                            704
  Beef and Apple Burgers          Beef
  672                            670
  Fountain                        Lamb Chops
  581                           430
  Beef and Broccoli               Naan
  391                            361
  Rice                            Chutney
  335                            279
  Lamb                            Roll
  212                            191
  Aubergine and Chickpea Vindaloo Glass Bottle
  176                           92
  Beef Stew                       Pork
  90                            75
  Chips                           Curry
  67                            22
  Non Food
  15

```

Help for fct_inorder:

- Description:** Reorder factors levels by first appearance, frequency, or numeric order.
- Usage:**

```
fct_inorder(f, ordered = NA)
fct_infreq(f, ordered = NA)
fct_inseq(f, ordered = NA)
```
- Arguments:**
 - `f`: A factor.
 - `ordered`: A logical which determines the "ordered" status of the output.
- Examples:**

```
f <- factor(c("b", "b", "a", "c", "c", "c"))
f
fct_inorder(f)
```

And now let's look at the structure of this category, and it tells me again it's a factor with 21 levels. If I summarize it now, I can see that it starts with salmon and ends with non food. And we've got the other categories in here, like Aberdeen and Chickpea Vindaloo, which is in the middle somewhere. So you can see what it's doing is it's ordering it in descending order based on the frequency of appearance. And so that's really helpful, especially if we want to plot it quick, create a quick histogram of it, and we can see that Sam and Wheat bran salad is the highest, and then something over here at the bottom. So this helps us see the relationship very quickly.

The screenshot shows an RStudio interface with a code editor on the left and a help viewer on the right.

Code Editor:

```
df$Category4 <- fct_lump(df$Category, n = 3)
```

Help Viewer (fct_lump):

Description: Lump together least/most common factor levels into "other".

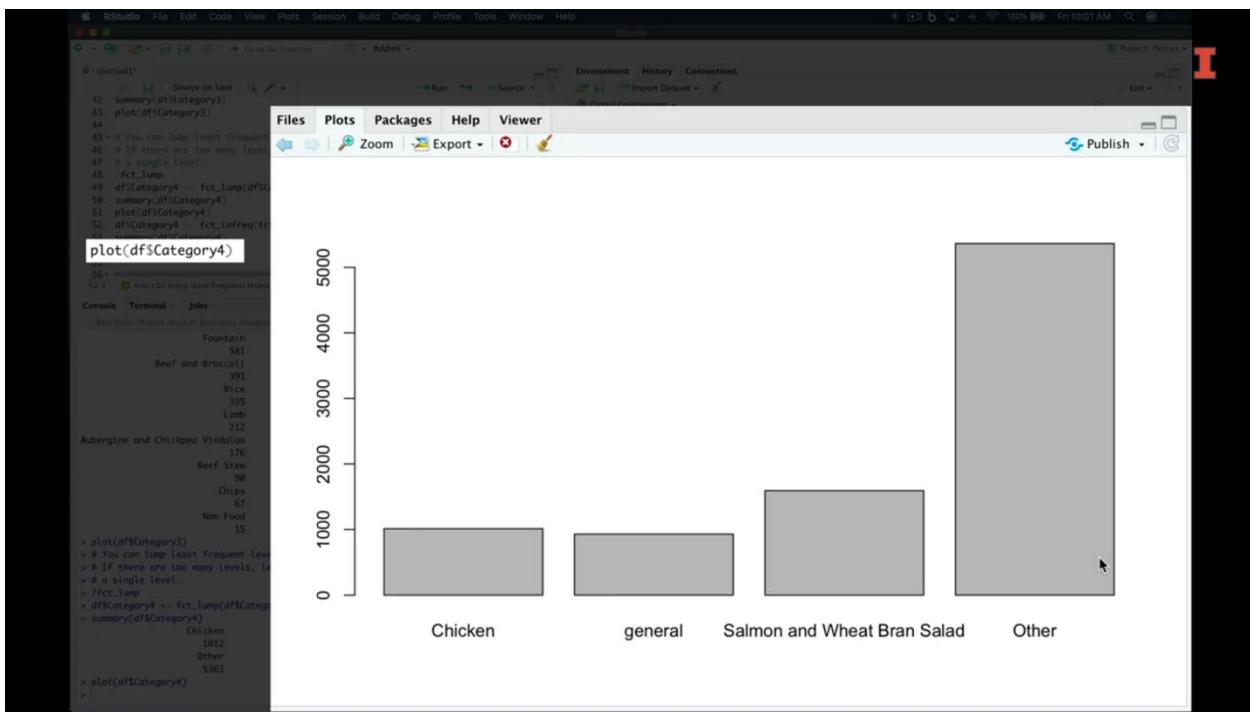
Usage:

```
fct_lump(n, prop, w = NULL, other_level = "Other", ties.method = c("min", "average", "first", "last", "random", "max"))  
fct_lump_min(f, min, w = NULL, other_level = "Other")
```

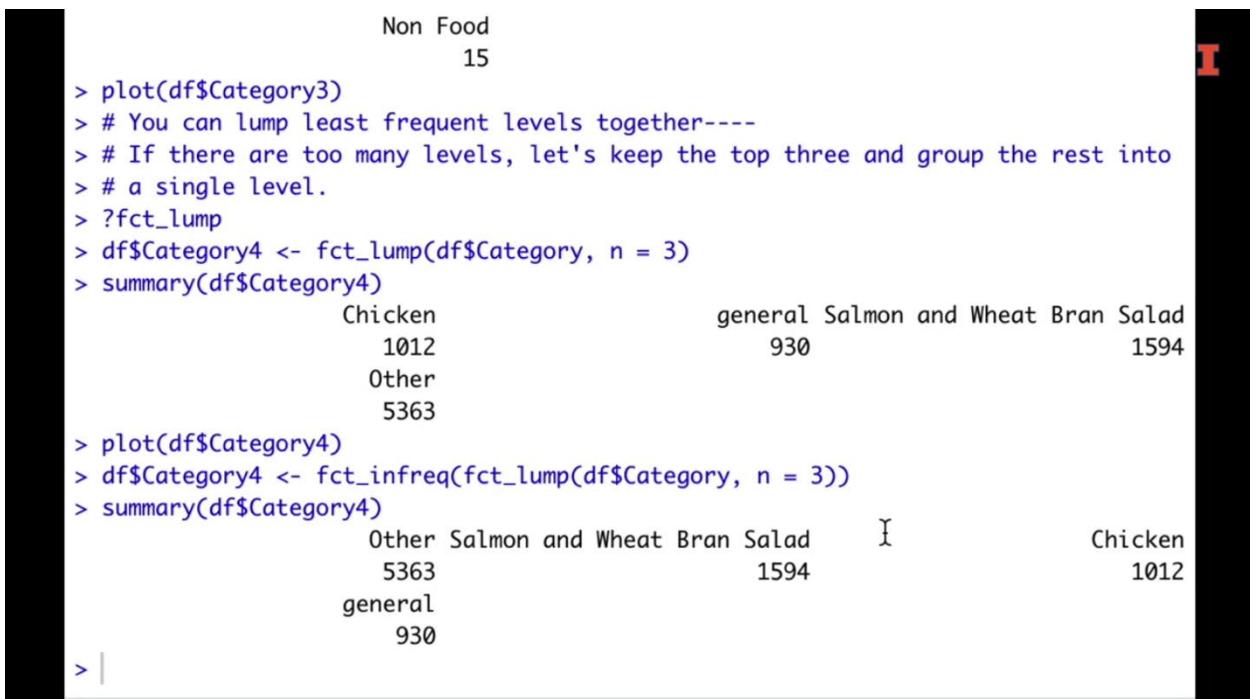
Arguments:

- f**: A factor (or character vector).
- n, prop**: If both n and prop are missing, `fct_lump` lumps together the least frequent levels into "other", while ensuring that "other" is still the smallest level. It's particularly useful in conjunction with `fct_inorder()`. Positive n preserves the most common n values. Negative n preserves the least common -n values. If there are ties, you will get at least abs(n) values. Positive prop preserves values that appear at least prop of the time. Negative prop preserves values that appear at most -prop of the time.
- w**: An optional numeric vector giving weights for frequency of each value (not level) in f.
- other_level**: Value of level used for "other" values. Always placed at end of levels.
- ties.method**: A character string specifying how ties are treated. See `rankAll` for details.

Now, another thing that you may want to do is recognize that Hey, there's a lot of different categories. I'm really not interested in all these other ones. I'm just interested in maybe the three most frequently occurring categories so we can use this other function in the four Cats package fact lump. And if you read the documentation on that, you can read all the details about this. But, what I'm going to do is I'm going to create another category column, Category four, and I'll use this fact lump function. And I'll indicate the category or sorry, the column that I'm using to create this new column from this lump factored column, and I'm going to indicate the number of unique levels in it. So I'm basically saying, I want to just see the top three levels, then lump everything else together in its own category, so I'll go ahead and run that.



Then I'll create the summary, and you can see that I've got chicken general salmon, wheat brand salad and then the other category. And if I plot it mhm, there's the chicken general sound, wheat bread, salad and other. And so that's nice, because it narrows it down to just my top three categories, but the order is not real helpful, so let's try ordering it in descending order.

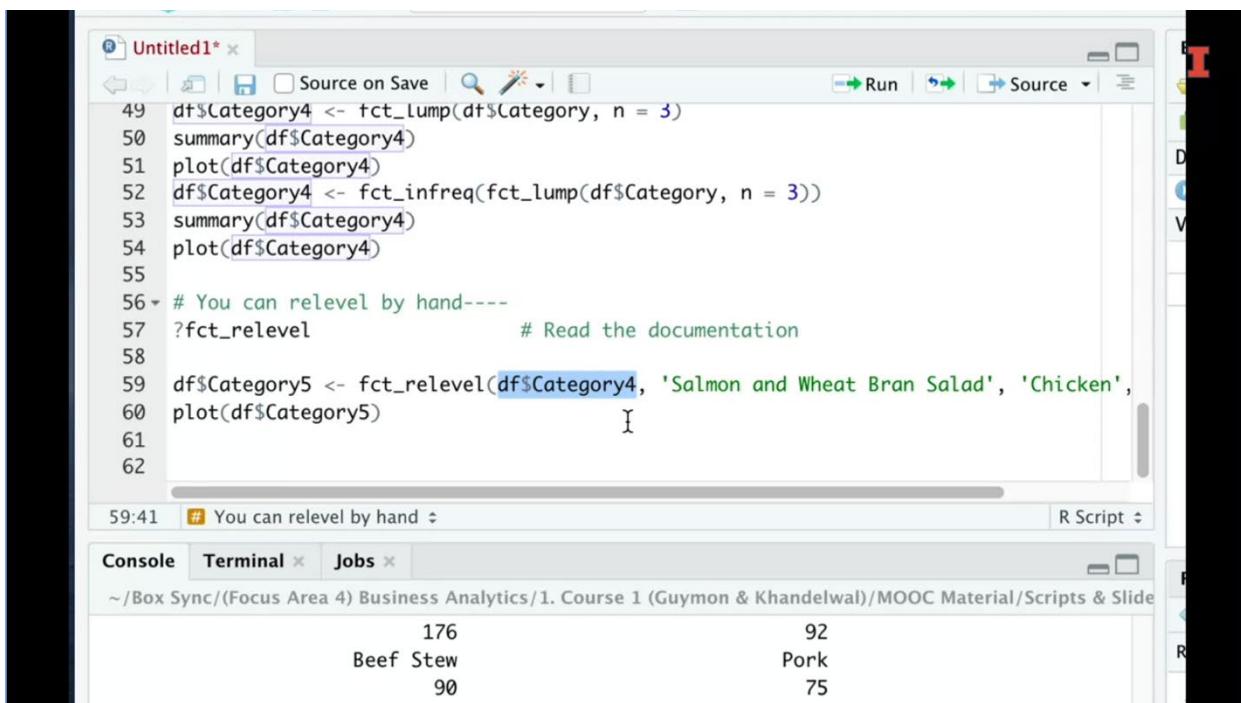


I'm going to run this fact lump again, but I'm going to wrap it in the fact and freak function. I'll run that and then look at the summary. And so now it's starting with other at the top, then salmon and white brand salad, then chicken and in general. And you can see that in the if we plot it in the history Graham that it's in descending order, which is often a great order.

The screenshot shows the RStudio interface with the following details:

- Console:** Displays R code for creating a dataset, lumping categories, summarizing, and plotting. It also shows the use of the `fct_relevel` function to reorder levels.
- Plots:** Shows a bar chart titled "Reorder factor levels by hand" with categories like Beef Steak, Chips, Non Food, etc., ordered from highest frequency (Beef Steak) to lowest (Non Food).
- Help:** The `fct_relevel` function is highlighted in the help documentation. The description notes it's a generalization of `stats::relevel()`. The usage section shows the syntax: `fct_relevel(.f, ..., after = 0L)`. The arguments section details `.f` (factor), `...` (function or character levels), and `after` (position). The examples section shows how to create a new factor with specific levels.

All right, so that's helpful. But this may not be exactly what I want, because other is kind of everything else. That's a miscellaneous category I'm not real interested in. I may actually want to start with the highest specific category salmon, wheat bran salad, then go down and then put other at the far right because it's not as important. So we can use this factory level function and you can read more about the details of that. But simply what I'm going to do is create a new column, Category five. Then I'll run this function.



The screenshot shows an RStudio interface. The top panel displays an R script named 'Untitled1' with the following code:

```

49 df$Category4 <- fct_lump(df$Category, n = 3)
50 summary(df$Category4)
51 plot(df$Category4)
52 df$Category4 <- fct_infreq(fct_lump(df$Category, n = 3))
53 summary(df$Category4)
54 plot(df$Category4)
55
56 # You can relevel by hand---
57 ?fct_relevel                         # Read the documentation
58
59 df$Category5 <- fct_relevel(df$Category4, 'Salmon and Wheat Bran Salad', 'Chicken',
60 plot(df$Category5)
61
62

```

The bottom panel shows the R Console output:

```

~ /Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts & Slides
      176          92
Beef Stew          Pork
      90           75

```

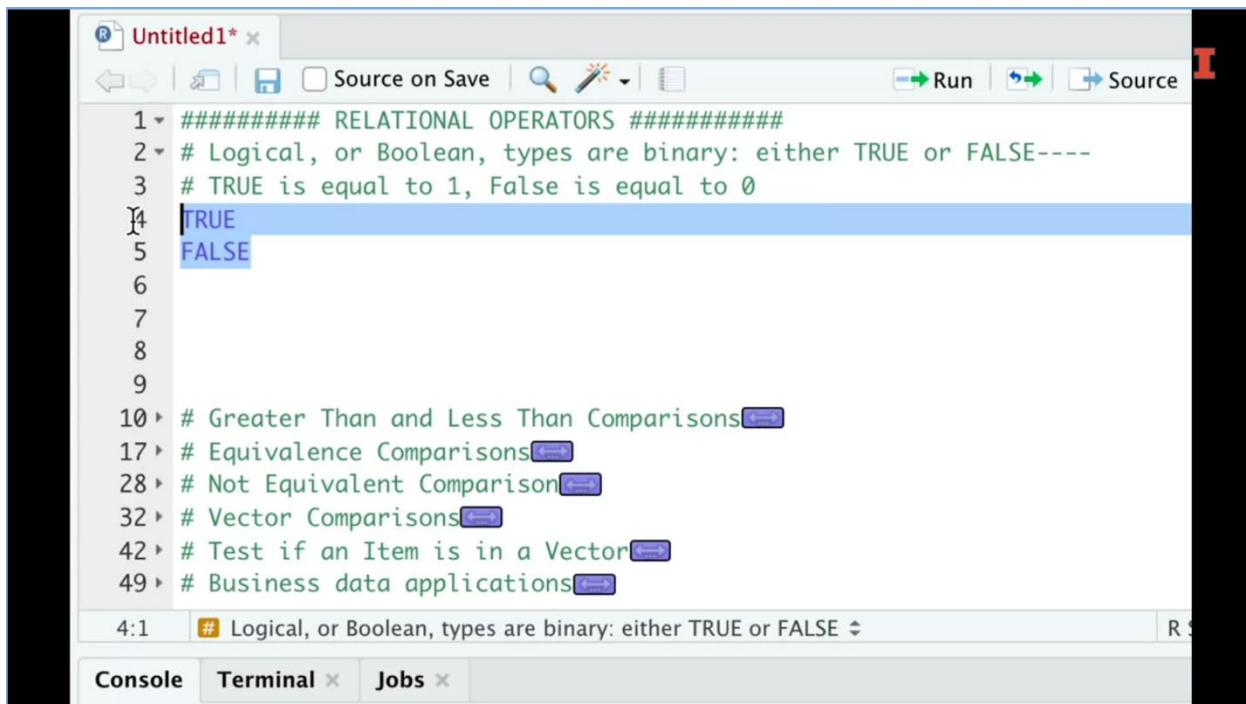
I'll base it off of Category four so a column that I've already created this lumped column and now I'm going to tell it the order. I'll manually indicate the order so you can see that I'm going to say, I want to start with salmon, wheat bran salad, then chicken and then general, I won't even put in other, and it will know what to do with it. All right, so I ran. That created Category five. Now let's plot it, and you can see that now I've got this meaningful, at least in perhaps for my context, it's a meaningful ordering of the factor levels.



It starts with the highest specific in the second highest third highest, and everything else is together. All right, so there's a quick overview of factors and how you can use the four Cats package to help you deal with factors

Lesson 3-9 Logical Type and Relational Operators

Lesson 3-9.1 Logical Type and Relational Operators



The screenshot shows an RStudio interface with a script editor window titled "Untitled1*". The code in the editor is as follows:

```
1: ##### RELATIONAL OPERATORS #####
2: # Logical, or Boolean, types are binary: either TRUE or FALSE----
3: # TRUE is equal to 1, False is equal to 0
4: TRUE
5: FALSE
6:
7:
8:
9:
10: # Greater Than and Less Than Comparisons
11: # Equivalence Comparisons
12: # Not Equivalent Comparison
13: # Vector Comparisons
14: # Test if an Item is in a Vector
15: # Business data applications
```

The code editor has tabs for "Console", "Terminal", and "Jobs". A red "I" icon is visible on the right side of the screen.

In this lesson we're going to talk about logical data types or Boolean data types. And then we'll talk about how those are used for returning the results of relational operators such as equivalent, or greater than, or less than. So let's start by identifying what is a logical or a Boolean data type. It's basically a data type that can have only two values, true or false. So if you type out TRUE or FALSE, in all caps, you'll notice that they turn blue in all because those are reserved words with special meaning. We can abbreviate those as TNF, and you'll see a capital T is blue. So is a capital F, and let's look at the class of just T. If we do that, it tells us it's a logical value.

The screenshot shows an RStudio interface. In the top-left pane, there is a code editor with the following R code:

```

x <- c(T,F,F,F,T)
summary(x)
plot(x)
sum(x)

```

In the top-right pane, the output window displays:

```

Values
x
logi [1:5] TRUE FALSE FALSE FALSE TRUE

```

Below the code editor, the console window shows:

```

> ###### RELATIONAL OPERATORS ######
> # Logical, or Boolean, types are binary: either TRUE or FALSE---
> # TRUE is equal to 1, False is equal to 0
> class(T)
[1] "logical"
> x <- c(T,F,F,F,T)
>

```

The bottom right pane shows a file browser with a list of files in a directory named "Focus Area 4 Business Analytics / 1. Course 1 (Guymon & Khandelwal) / MOOC Material / Scripts".

Let's create a vector of logical values. Do true, false, false, false, true. So we'll create that vector. And let's look at how vectors are somewhat like factors in the sense that you can plot them and you can get a summary of them.

The screenshot shows an RStudio interface. The code editor contains the same R code as the previous screenshot:

```

x <- c(T,F,F,F,T)
summary(x)
plot(x)
sum(x)

```

The console window shows the same output as before:

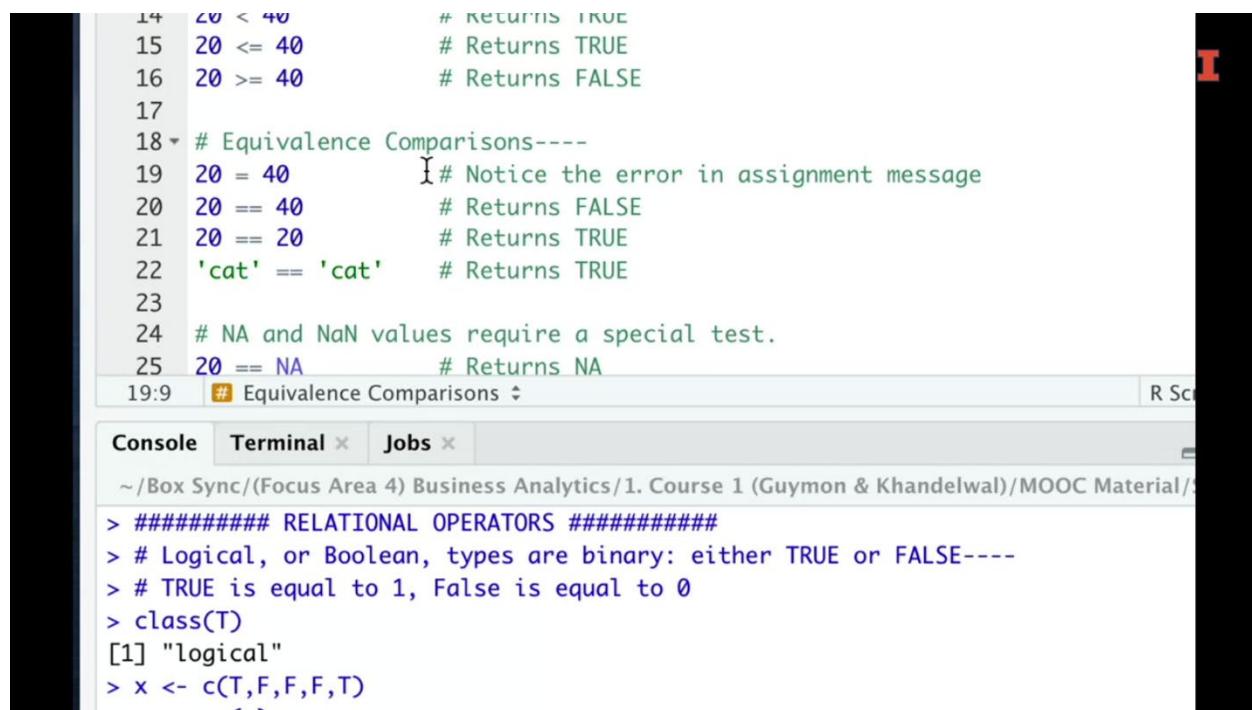
```

> ###### RELATIONAL OPERATORS ######
> # Logical, or Boolean, types are binary: either TRUE or FALSE---
> # TRUE is equal to 1, False is equal to 0
> class(T)
[1] "logical"
> x <- c(T,F,F,F,T)
> summary(x)
Mode   FALSE    TRUE
logical 3       2
> plot(x)
> sum(x)
[1] 2
>

```

The bottom right pane displays a scatter plot with the x-axis labeled "x" and the y-axis ranging from 0.4 to 1.0. There is one point plotted at approximately (0.5, 1.0).

So if we use the summary function on this vector x that we created, it tells us that there are 3 FALSE and 2 TRUE and that it's a logical vector. And if we plot it, it returns a plot that shows us really the way logical values are treated. TRUEs are ones, and FALSEs are our zeros. So we can see 10001. We can also see that they're treated like zero or ones if we use a sum function on x, and it will return 2 because there are two TRUEs. So there's a quick summary of how logical data types work and are. Now that we know how logical data types work, let's perform some comparison operations.



The screenshot shows an RStudio environment. On the left is a code editor with R code, and on the right is a console window.

```

14  20 < 40      # Returns TRUE
15  20 <= 40     # Returns TRUE
16  20 >= 40     # Returns FALSE
17
18 # Equivalence Comparisons---
19  20 = 40       # Notice the error in assignment message
20  20 == 40      # Returns FALSE
21  20 == 20      # Returns TRUE
22  'cat' == 'cat' # Returns TRUE
23
24 # NA and NaN values require a special test.
25  20 == NA      # Returns NA
19.9 # Equivalence Comparisons ▾

```

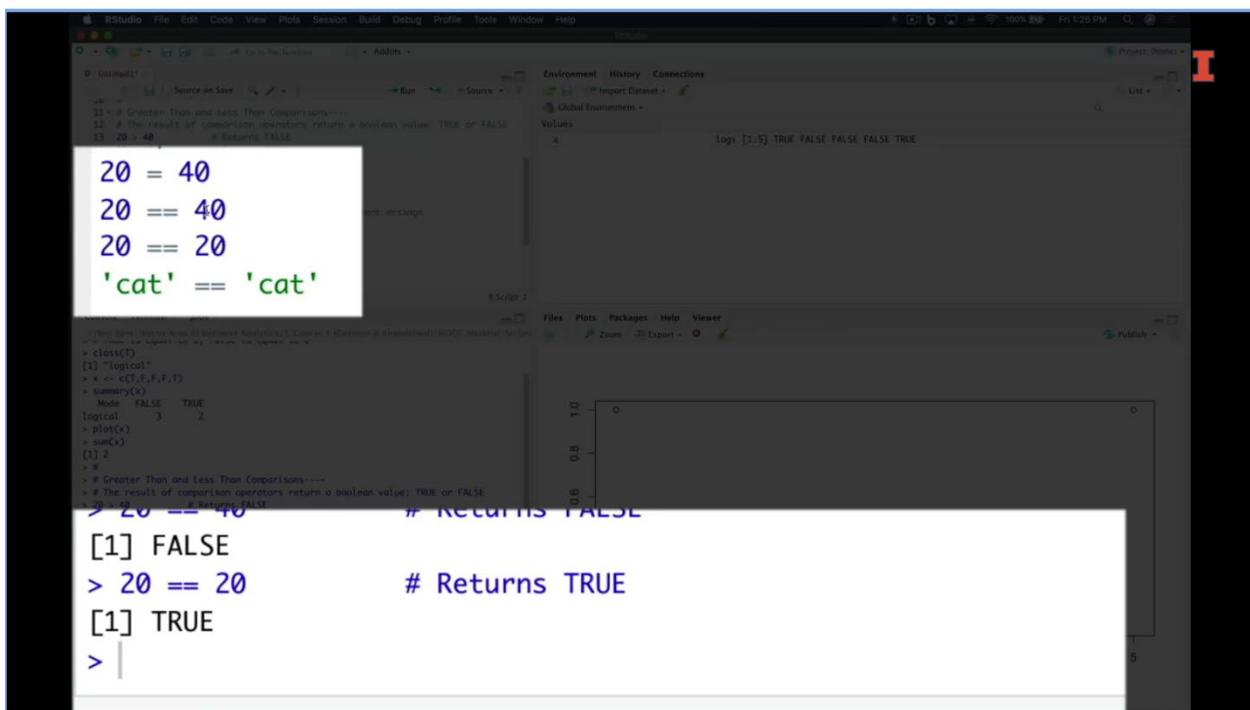
Console output:

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/
> ##### RELATIONAL OPERATORS #####
> # Logical, or Boolean, types are binary: either TRUE or FALSE---
> # TRUE is equal to 1, False is equal to 0
> class(T)
[1] "logical"
> x <- c(T,F,F,F,T)

```

Let's perform some simple comparisons. Is $20 > 40$? That's what this will be evaluated as it will return FALSE. A logical value FALSE. $20 < 40$ returns TRUE, $20 \leq 40$ so you can use the less than or equal to sign, that is also TRUE or greater than or equal to sign. You can use as well, and that returns FALSE. So those are pretty straightforward, and I just wanted to highlight that they returned those logical values. What if you want to test if some object is equal to another object. Well, we can't use just a single equal sign because that is interpreted as assignment.



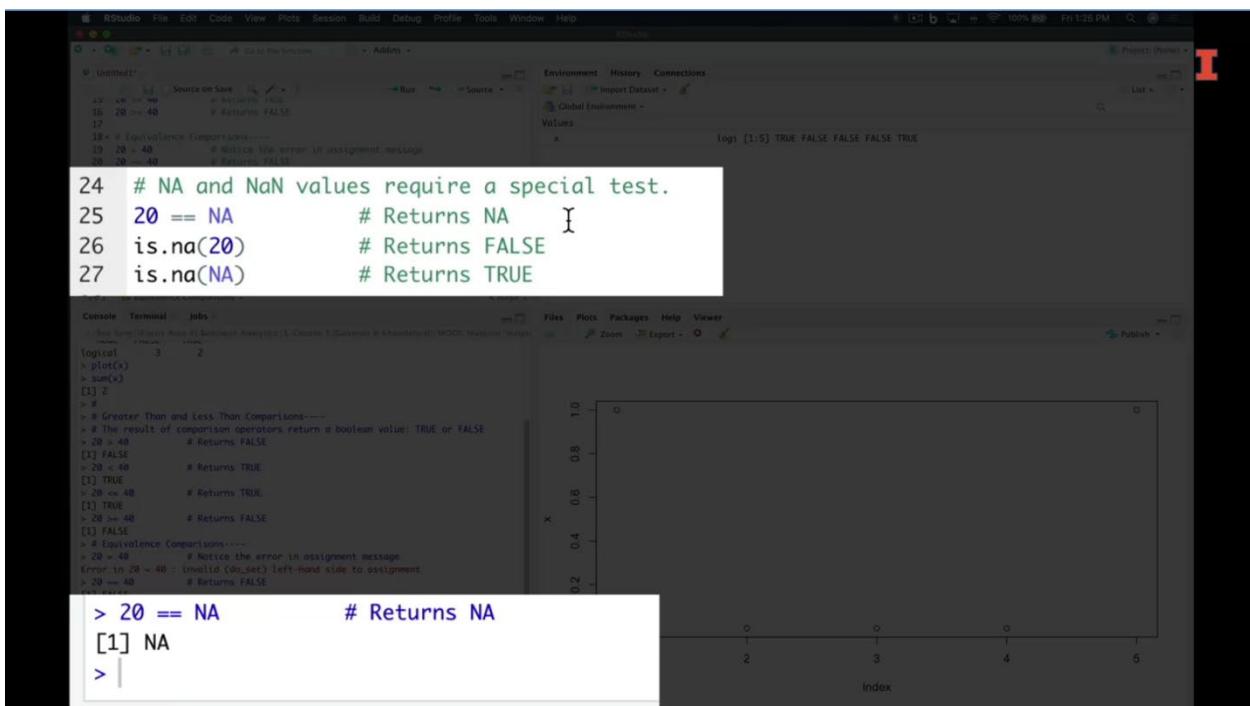
```

RStudio File Edit Code View Plots Session Build Debug Profile Tools Window Help
Console History Connections
Untitled1 Source on Save Run Source Environment History Connections
11 <-- Greater Than and Less Than Comparisons---#
12 # The result of comparison operators return a boolean value: TRUE or FALSE
13 20 <- 40 # Returns FALSE
20 = 40
20 == 40
20 == 20
'cat' == 'cat'

R Script 1 Files Plots Packages Help Viewer
14 <-- Logical Values---#
15 logical()
16 [1] "logical"
17 > c(F,F,T)
18 > is.logical(x)
19 > Mode(x)
20 > logical 3 TRUE
21 > plot(x)
22 > sum(x)
23 [1] 2
24 > # Greater Than and Less Than Comparisons---#
25 > 20 == 40 # Returns FALSE
[1] FALSE
26 > 20 == 20 # Returns TRUE
[1] TRUE
27 >

```

And we can't assign 40 to the value of 20, that will give us an error. So if we want to test, if a value is equal to another value, we use a double equal sign. So that's the equivalence comparison there. So it's $20 == 40$? It returns FALSE. How about $20 == 20$? That returns TRUE. And we can perform this with strings as well, as the character string ' $'cat' == 'cat'$ '. TRUE, that is all, right?



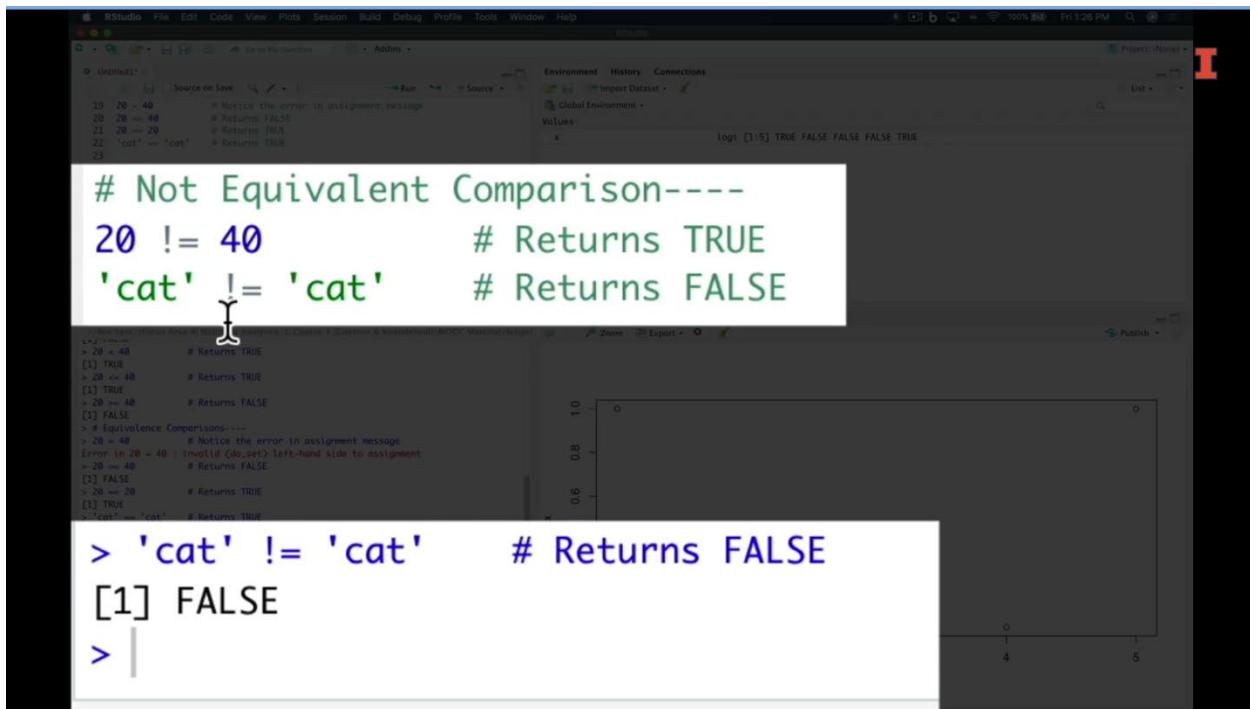
```

RStudio File Edit Code View Plots Session Build Debug Profile Tools Window Help
Console History Connections
Untitled1 Source on Save Run Source Environment History Connections
14 <-- Logical Values---#
15 logical()
16 [1] "logical"
17 > plot(x)
18 > sum(x)
19 [1] 2
20 > # Greater Than and Less Than Comparisons---#
21 > 20 <- 40 # Returns FALSE
22 > 20 = 40 # Returns FALSE
23 > 20 < 40 # Returns TRUE
24 > 20 == 40 # Returns TRUE
25 > 20 == 20 # Returns FALSE
26 > is.na(20) # Returns FALSE
27 > is.na(NA) # Returns TRUE

R Script 1 Files Plots Packages Help Viewer
28 <-- NA and NaN values require a special test.
29 > 20 == NA # Returns NA
30 > is.na(20) # Returns FALSE
31 > is.na(NA) # Returns TRUE
32 >
33 > # NA and NaN values require a special test.
34 > 20 == NA # Returns NA
[1] NA
35 >

```

What if we want to compare something with NA or NaN? Those null values. We can't use the same equivalence comparison. The double equal sign, because it is basically saying, is 20 equivalent to an unknown value and it'll say, well, I don't know. So instead, we need to use the special equivalent operator for NAs, and that is simply `is.na`. So `is.na 20` meaning is 20 not applicable and a non-existent value? FALSE, because it has a value. Is na an NA? And that returns TRUE. All right, so those are equivalent operators.



The screenshot shows an RStudio session with the following code and output:

```

# Not Equivalent Comparison---
20 != 40          # Returns TRUE
'cat' != 'cat'    # Returns FALSE

# Equivalence Comparisons---
> 20 == 40        # Returns TRUE
[1] TRUE
> 20 == 40        # Returns FALSE
[2] FALSE
> 20 === 40       # Returns FALSE
[3] FALSE
> 'cat' == 'cat'  # Returns TRUE
[1] TRUE
> 'cat' == 'cat'  # Returns TRUE
[1] TRUE

```

The code examples demonstrate the use of `!=` for non-equivalence and `==` and `===` for equivalence. The output shows the results for each comparison.

What about if we want to see if something is not equivalent to something else? Then we use an exclamation point and an equal sign. So it's `20 != 40`? That is TRUE. And that's a little backwards, I realized that can be easy to be confused about that. But there definitely times when you want to see if something is not equal to something. So that returns TRUE. And if we compare things that are equivalent to each other and ask if they are not equivalent to each other, then it will return FALSE.

The screenshot shows an RStudio interface. On the left, the code editor displays R code demonstrating vector comparisons:

```

# Vector Comparisons---
v1 <- c(1,5)
v2 <- c(1,3)
v3 <- c(1,3,5)
v4 <- c(2,NA)
v1 == v2      # Returns a vector with values of TRUE FALSE
v1 > v2      # Returns a vector with values of FALSE TRUE
v1 == v3    I # Returns an error because the vectors are different lengths.
v1 > v4      # Returns a vector with values of FALSE NA

> is.na(NA)      # Returns TRUE
[1] TRUE

> # Not Equivalent Comparison---
> 20 != 40      # Returns TRUE
[1] TRUE
> 'cat' != 'cat'  # Returns FALSE
[1] FALSE

> # Vector Comparisons---
> v1 <- c(1,5)
> v2 <- c(1,3)
> v3 <- c(1,3,5)
> v4 <- c(2,NA)
> v1 == v2      # Returns a vector with values of TRUE FALSE
[1] TRUE FALSE
> v1 > v2      # Returns a vector with values of FALSE TRUE
[1] FALSE TRUE
>

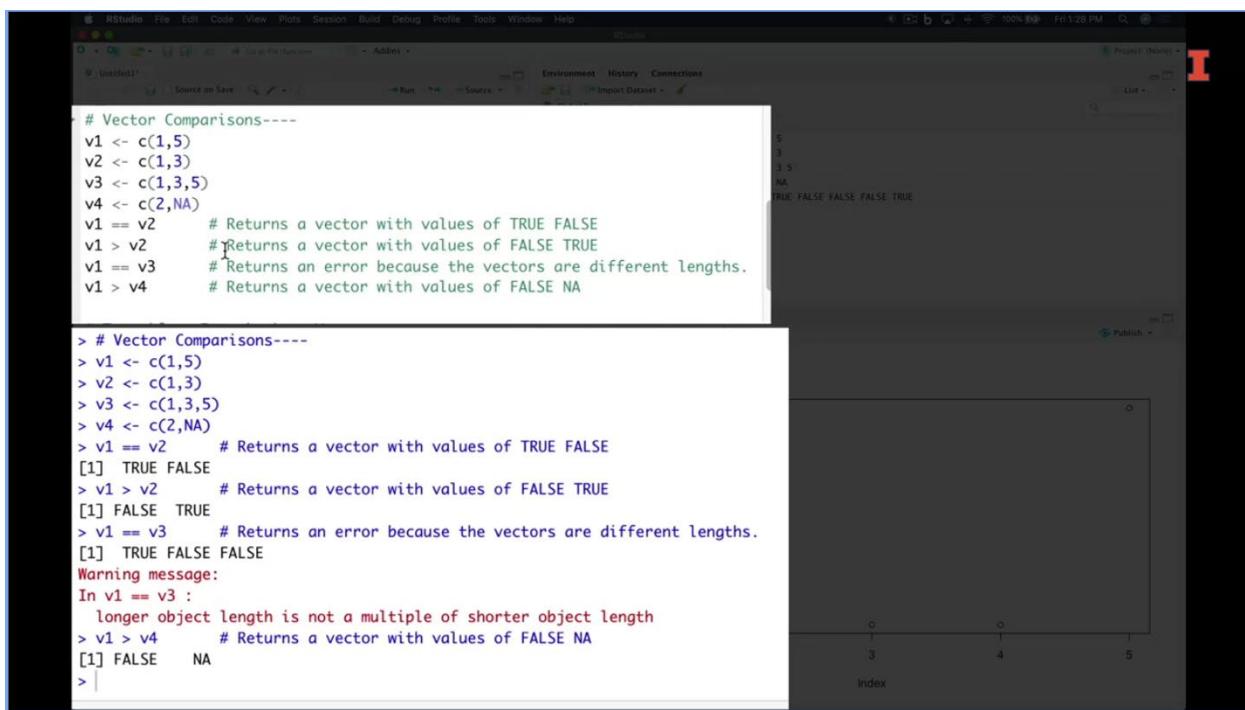
```

To the right, the environment pane shows the vectors v1 through v4:

	v1	v2	v3	v4
\$	1	3	1	2
NA	5		3	NA
TRUE FALSE FALSE FALSE TRUE				

Below the environment pane, a histogram is displayed with the x-axis labeled "Index" and values 0, 1, 3, 4, 5.

All right, and we can perform these comparisons on vectors as well. So let's create a vector1, that is v1 has the values of 1 and 5. v2 has the values 1 and 3. v3 has values 1, 3, 5 and v4 has values 2 and NA. So what happens if we compare vector1 to vector2 and see if they are equivalent to each other? Notice what happens is that it returns a vector of logical values TRUE and FALSE. So it does an element by element comparison in there. So in comparing vector1 to vector2 to each other, the first element is equivalent, but the second elements are not equivalent. What if we use the greater than and perform that comparison? Is v1 > v2? Well, again, it returns a vector of logical values FALSE, because 1 is equal to 1, it's not greater than 1. And then TRUE, because 5 is greater than 3.



The screenshot shows an RStudio interface with the following code and output:

```

# Vector Comparisons---
v1 <- c(1,5)
v2 <- c(1,3)
v3 <- c(1,3,5)
v4 <- c(2,NA)
v1 == v2      # Returns a vector with values of TRUE FALSE
v1 > v2      # Returns a vector with values of FALSE TRUE
v1 == v3      # Returns an error because the vectors are different lengths.
v1 > v4      # Returns a vector with values of FALSE NA

> # Vector Comparisons---
> v1 <- c(1,5)
> v2 <- c(1,3)
> v3 <- c(1,3,5)
> v4 <- c(2,NA)
> v1 == v2      # Returns a vector with values of TRUE FALSE
[1] TRUE FALSE
> v1 > v2      # Returns a vector with values of FALSE TRUE
[1] FALSE TRUE
> v1 == v3      # Returns an error because the vectors are different lengths.
[1] TRUE FALSE FALSE
Warning message:
In v1 == v3 :
  longer object length is not a multiple of shorter object length
> v1 > v4      # Returns a vector with values of FALSE NA
[1] FALSE NA
>

```

The output pane shows the results of the comparisons:

```

5
3
NA
TRUE FALSE FALSE FALSE TRUE

```

A small plot window is visible on the right side of the interface.

Now, just like when we perform Math with vectors, the vectors need to be the same length. If we try to compare vector1 to vector3, which are not the same length, then we'll get a warning message telling us that they're not the same length. And then finally, what if we compare values that have NaN in a vector? While just like an individual item, it will return values of FALSE and then NA. Because you can't really compare a number to a NA value. All right, so in other words, so the short of it is that you can use comparison operators on vectors, and it will do an element by element comparison.

The screenshot shows the RStudio interface with the following session history:

```

36 v3 <- c(1,3,5)
37 v4 <- c(2,NA)
38 v1 <- v2      # Returns a vector with values of 1
39 v1 <- v3      # Returns a vector with values of 1
40 v1 <- v4      # Returns an error because the vectors are different lengths.
41 v1 <- v1      # Returns a vector with values of 1

# %in%
1 %in% v1
2 %in% v1

> 1 %in% v1
[1] TRUE
> 2 %in% v1
[1] FALSE

```

The Environment pane shows a table titled "Values" with the following data:

v1	num [1:2]	1 5
v2	num [1:3]	1 3 5
v4	num [1:2]	2 NA
x	log[1:5]	TRUE FALSE FALSE FALSE TRUE

The Plots pane displays a scatter plot with points at (1, 1), (2, 2), (3, 3), (4, 4), and (5, 5).

One other operator that's really helpful that I want to present to you, is this in operator? So %in%. And what this will do is it will test if a value exists somewhere in as a vector or a list of value. So let's see if 1 is in the vector of 1 in 5. We know it is, but let's use this %in% and see if 1 is in v1. It returns a single value TRUE, because 1 is in v1. And if it were in their multiple times, it would still just return the single value of TRUE. In contrast, if we compared 2 in v1, then it returns FALSE because 2 is not in the vector of 1 and 5.

The screenshot shows the RStudio interface with the following session history:

```

36 v3 <- c(1,3,5)
37 v4 <- c(2,NA)
38 v1 <- v2      # Returns a vector with values of 1
39 v1 <- v3      # Returns a vector with values of 1
40 v1 <- v4      # Returns an error because the vectors are different lengths.
41 v1 <- v1      # Returns a vector with values of 1

!2 %in% v1
!1 %in% v1

> !2 %in% v1
[1] TRUE
> !1 %in% v1
[1] FALSE

```

The Environment pane shows a table titled "Values" with the following data:

v1	num [1:2]	1 5
v2	num [1:3]	1 3 5
v4	num [1:2]	2 NA
x	log[1:5]	TRUE FALSE FALSE FALSE TRUE

The Plots pane displays a scatter plot with points at (1, 1), (2, 2), (3, 3), (4, 4), and (5, 5).

So to evaluate if something is not in a vector or a group, it's not as straightforward. But it's somewhat similar to using the not equivalent comparison So let's start with `2 %in% v1`. And if we want to test if 2 is not in v1 instead of if it is in v1, then we need to put the exclamation point in front of the 2 and run that. That returns TRUE because 2 indeed is not in v1. So it's not super intuitive, but that's how you do. And just for completeness, let's test if `!1 %in% v1`. And that is FALSE because 1 is in v1.

The screenshot shows the RStudio interface. On the left, the 'Console' tab is active, displaying R code and its output. The code is as follows:

```

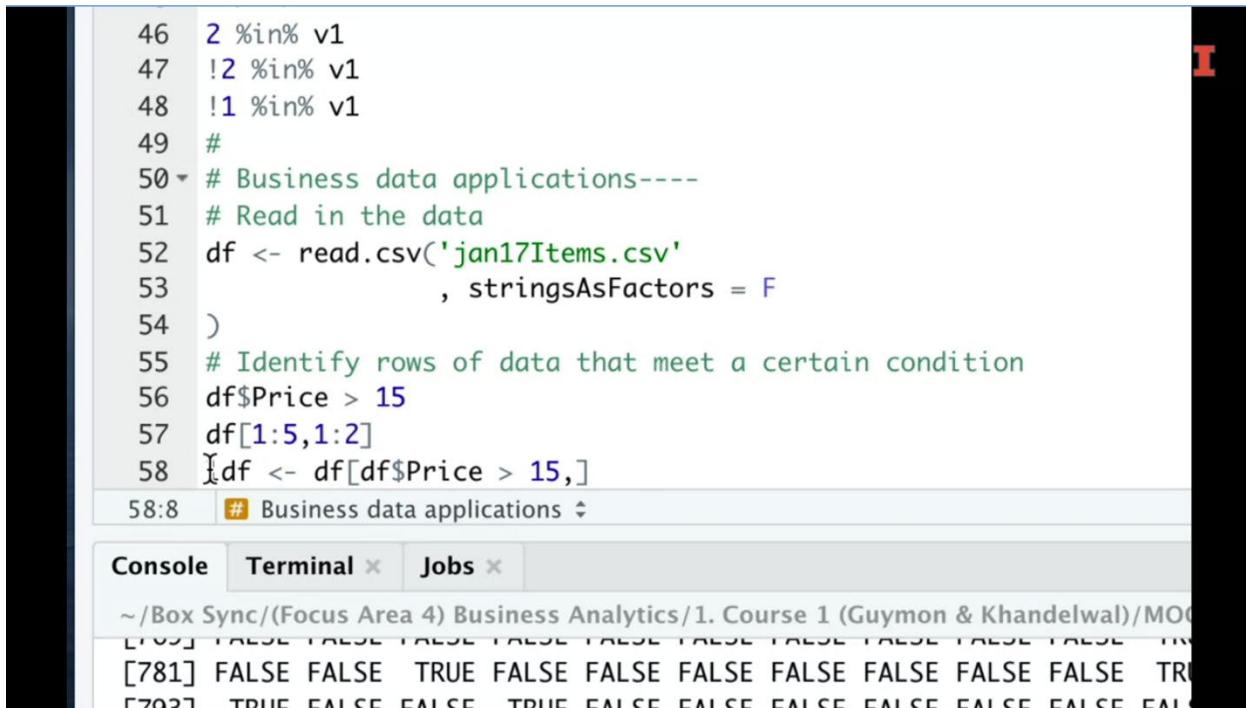
56 df$Price > 15
57
58
57:1  Business data applications  R Script
[Console] Terminal × Jobs ×
~/Box Sync/[Focus Area 4] Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Material/Scripts
[697] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE
[709] NA FALSE TRUE NA FALSE FALSE FALSE TRUE FALSE TRUE FALSE TRUE
[721] NA FALSE FALSE
[733] FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE NA
[745] FALSE FALSE
[757] FALSE FALSE FALSE FALSE FALSE TRUE FALSE NA FALSE FALSE FALSE FALSE FALSE
[769] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE FALSE
[781] FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE
[793] TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[805] FALSE TRUE
[817] FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[829] FALSE FALSE FALSE FALSE NA FALSE FALSE NA FALSE FALSE FALSE FALSE TRUE
[841] NA FALSE NA FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[853] FALSE FALSE NA TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[865] TRUE FALSE TRUE TRUE NA FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE TRUE
[877] NA FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE TRUE FALSE TRUE
[889] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE NA NA TRUE FALSE
[901] FALSE FALSE FALSE FALSE FALSE TRUE NA NA FALSE FALSE FALSE
[913] FALSE NA FALSE FALSE TRUE FALSE TRUE FALSE FALSE FALSE FALSE FALSE NA
[925] FALSE NA FALSE FALSE FALSE TRUE TRUE FALSE TRUE TRUE FALSE FALSE
[937] NA FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE
[949] FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
[961] FALSE FALSE
[973] FALSE TRUE FALSE FALSE FALSE NA FALSE TRUE FALSE FALSE TRUE FALSE
[985] FALSE FALSE
[997] FALSE FALSE
> [ reached getOption("max.print") -- omitted 7899 entries ]

```

On the right, the 'Files' tab of the file browser is selected, showing a list of files in the current directory:

- ..
- .Rhistory
- conditionalStatements.R
- dates.R
- factors.R
- feb17Items.csv
- feb17Weather.csv
- jan17Items.csv
- jan17Weather.csv
- loops.R
- mar17Items.csv
- mar17Weather.csv
- relationalOperators.R
- RScripts_Module2.Rproj
- stackingData.R

Let's now talk about how we can apply relational operators and logical values to a business data application. So I'm going to go ahead and read in the jan17items.csv file and convert it to a data frame. One way that we can use these relational operators is to filter the data down to certain rows based on a condition that we are interested in. Let's just start by looking at and making sure that if we compare this vector, let's say Price and we'll see if Price > 15. Think of Price just a vector and we're going to element by element go down and make that comparison. So what we get is a vector that is 8899 elements long. And it has logical values, TRUE, FALSE or NA, based on whether or not price in that row is greater than 15.



```

46  2 %in% v1
47  !2 %in% v1
48  !1 %in% v1
49  #
50  # Business data applications----
51  # Read in the data
52  df <- read.csv('jan17Items.csv'
53  , stringsAsFactors = F
54  )
55  # Identify rows of data that meet a certain condition
56  df$Price > 15
57  df[1:5,1:2]
58  ldf <- df[df$Price > 15,]
58:8 # Business data applications

```

Console Terminal × Jobs ×

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/Mod
L'00] FALSE FALSE
[781] FALSE FALSE TRUE FALSE FALSE
[782] TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

```

Now, that's not helpful in and of itself. But here's how it can be. So remember how we can identify certain rows of data based on location. Let's say we want to just return the first five rows of data and maybe just the first two columns of data from the DF data frame. We can do it like that. So let's take that idea and combine it with this idea, performing an element by element comparison. And let's return only the rows of data of df, for which the Price value > 15. All right, so it's going to return in this bracket here. It will basically say, return only those rows that have a value of TRUE in it or NA actually. So let's assign this to a new data frame, we'll call ldf.

The screenshot shows the RStudio interface. The top bar displays the title "Untitled1*" and various tool icons. The main area is a script editor containing R code. The code reads a CSV file named "jan17Items.csv", filters rows where Price > 15, and then further filters to include only items in the categories 'Beef' or 'Rice'. The bottom navigation bar includes tabs for "Console", "Terminal", and "Jobs". The "Console" tab is active, showing the command "# Business data applications" and its execution status.

```
45 1 %in% v1
46 2 %in% v1
47 !2 %in% v1
48 !1 %in% v1
49 #
50 # Business data applications----
51 # Read in the data
52 df <- read.csv('jan17Items.csv'
53             , stringsAsFactors = F
54 )
55 # Identify rows of data that meet a certain condition
56 df$Price > 15
57 df[1:5,1:2]
58 ldf <- df[df$Price > 15, ]
59 brdf <- df[df$Category %in% c('Beef', 'Rice')]
59:45 # Business data applications
```

And if we look at this data frame, it is a much smaller subset 1942 obs. And if we go to the price column, we can see that all the observations are either greater than 15 or RNA. Right, so that's not the most convenient way to filter data, but this is how it could be applied, at least under the hood in some instances. Now, another way in which we might want to filter the data is, let's see if we can narrow down this data frame to only the rows for which the category is beef or rice. So let's name this new data frame brdf.

The screenshot shows the RStudio interface. The top bar displays the title "Untitled1*" and various tool icons. The main area contains an R script with the following code:

```
46 2 %in% v1
47 !2 %in% v1
48 !1 %in% v1
49 #
50 # Business data applications----
51 # Read in the data
52 df <- read.csv('jan17Items.csv'
53 , stringsAsFactors = F
54 )
55 # Identify rows of data that meet a certain condition
56 df$Price > 15
57 df[1:5,1:2]
58 ldf <- df[df$Price > 15,]
59 brdf <- df[df$Category %in% c('Beef', 'Rice'),]
60 summary(factor(brdf$Category))

60:25 # Business data applications
```

The "Console" tab is selected at the bottom, showing the output of the script execution.

And we'll say, return the rows of df for which the df category value is in. And then we can create a vector as a character string vector, that contains Beef and Rice in it. And then we'll put a comma to indicate we want all the columns of data. And we'll run, that and we get this brdf data frame. It's 1005 observations long. And if we just visually explore that, we can look at the category and see that there's beef and rice in there. And we can even do a summary of the values in this be brdf data frame, and it currently has br category as a character string.

The screenshot shows an RStudio interface with the following details:

- Code Area:**

```

54 )
55 # Identify rows of data that meet a certain condition
56 df$Price > 15
57 df[1:5,1:2]
58 ldf <- df[df$Price > 15,]

summary(factor(brdf$Category))

```
- Output Area:**

	v1	v2	v3	v4	x
num [1:2]	1 5	num [1:2]	1 3	num [1:3]	1 3 5
logi [1:5]	TRUE FALSE FALSE FALSE TRUE				
- Environment Tab:**

	v1	v2	v3	v4	x
num [1:2]	1 5	num [1:2]	1 3	num [1:3]	1 3 5
logi [1:5]	TRUE FALSE FALSE FALSE TRUE				
- File Explorer:**
 - Focus Area 4 Business Analytics 1. Course 1 (Guymon & Khandelwal) MOOC Material
 - Names
 - Rhistory
 - conditionalStatements.R
 - dates.R
 - factors.R
 - feb17Items.csv
 - feb17Weather.csv
 - jan17Items.csv
 - jan17Weather.csv
 - loops.R
 - mar17Items.csv
 - mar17Weather.csv
 - relationalOperators.R
 - RScripts_Module2.Rproj
 - stackingData.R

So let's change it to a factor within the summary calculation so that we can see the different values were on that. And there we've got 670 rose for which the categories beef and 335 to which the category is rice. So that's an introduction to logical data types and how they are used in relational operators, as well as how they could be applied in a business data setting. We'll talk about simpler ways to do this filtering, but hopefully that's helpful to get an idea of why they're important.

Lesson 3-10 Character Strings

Lesson 3-10.1 Character Strings

The RStudio interface displays the following R code in the console:

```
1 #----- CHARACTER STRINGS -----
2 # install and load the stringr package
library(stringr)

7 testString <- 'This is so fun.'
8
9
10 # Read in data----
11 ji7i <- read.csv('jan17Items.csv')
12
13 # Length----
14 str_length(testString)
15
16 # Case----
17 str_to_title(testString)
18 str_to_lower(testString)
19 str_to_upper(testString)
20
21 # Detect----
22 str_detect(testString, 'his')
23
24
4:17 "Install and load the stringr package :"
```

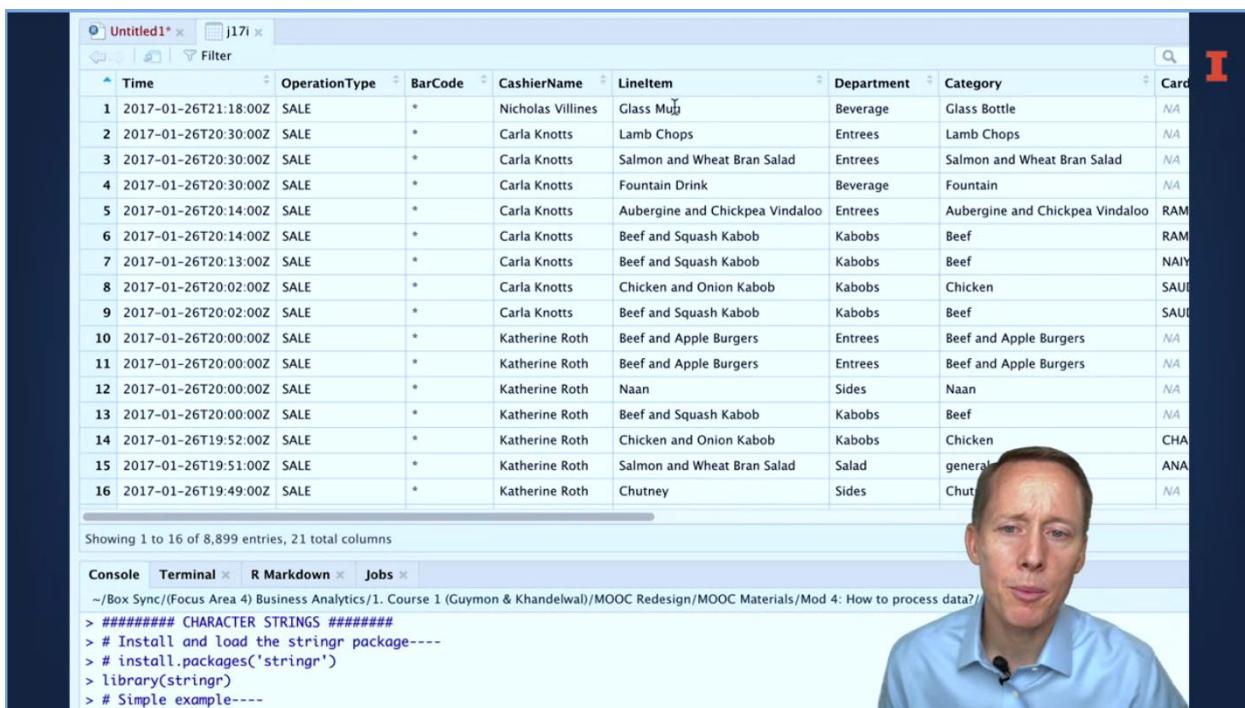
The video player shows a man speaking, likely the professor, with a red 'I' logo in the top right corner.

In business analytics, there are a variety of tasks that are often performed using character strings. In this lesson, I want to show you how to use functions from the stringer package to perform some of those tasks. The first thing is to make sure that you've already installed the stringer function on your machine. If not, then go ahead and run the installed packages function or install the package using the packages. Pain. Yeah, yeah. Once it's installed, make sure and load the functions from the Stringer library by using the library function.

```
1 # ##### CHARACTER STRINGS ↵
2 # Install and load the stringr package ↵
3 # Simple example---- ↵
4 testString <- 'This is so fun.' ↵
5 ↵
6 ↵
7 # Read in data---- ↵
8 j17i <- read.csv('jan17Items.csv') ↵
9 ↵
10 # Length ↵
11 # Case ↵
12 # Detect ↵
13 # Replace ↵
14 # Split ↵
```



Now, before we start using functions from the stringr package, let's go ahead and look at what some of those functions are, so you can see that there are many functions that start with str underscore. We are going to review some of those functions in this lesson. Let's go ahead and start with a simple character string, and it will be a sentence. This is so fun, and we'll store in an object test string so we don't have to keep typing out. This is so fun. I also want to show you some examples using business data, so I'm going to read in the Jan. 17 items dot c. S v file and store it as a data frame called J 17 I.



The screenshot shows a data analysis environment with a data frame viewer and an R console.

Data Frame View:

	Time	OperationType	BarCode	CashierName	LineItem	Department	Category	Card
1	2017-01-26T21:18:00Z	SALE	*	Nicholas Villines	Glass Mug	Beverage	Glass Bottle	NA
2	2017-01-26T20:30:00Z	SALE	*	Carla Knotts	Lamb Chops	Entrees	Lamb Chops	NA
3	2017-01-26T20:30:00Z	SALE	*	Carla Knotts	Salmon and Wheat Bran Salad	Entrees	Salmon and Wheat Bran Salad	NA
4	2017-01-26T20:30:00Z	SALE	*	Carla Knotts	Fountain Drink	Beverage	Fountain	NA
5	2017-01-26T20:14:00Z	SALE	*	Carla Knotts	Aubergine and Chickpea Vindaloo	Entrees	Aubergine and Chickpea Vindaloo	RAM
6	2017-01-26T20:14:00Z	SALE	*	Carla Knotts	Beef and Squash Kabob	Kabobs	Beef	RAM
7	2017-01-26T20:13:00Z	SALE	*	Carla Knotts	Beef and Squash Kabob	Kabobs	Beef	NA
8	2017-01-26T20:02:00Z	SALE	*	Carla Knotts	Chicken and Onion Kabob	Kabobs	Chicken	SAU
9	2017-01-26T20:02:00Z	SALE	*	Carla Knotts	Beef and Squash Kabob	Kabobs	Beef	SAU
10	2017-01-26T20:00:00Z	SALE	*	Katherine Roth	Beef and Apple Burgers	Entrees	Beef and Apple Burgers	NA
11	2017-01-26T20:00:00Z	SALE	*	Katherine Roth	Beef and Apple Burgers	Entrees	Beef and Apple Burgers	NA
12	2017-01-26T20:00:00Z	SALE	*	Katherine Roth	Naan	Sides	Naan	NA
13	2017-01-26T20:00:00Z	SALE	*	Katherine Roth	Beef and Squash Kabob	Kabobs	Beef	NA
14	2017-01-26T19:52:00Z	SALE	*	Katherine Roth	Chicken and Onion Kabob	Kabobs	Chicken	CHA
15	2017-01-26T19:51:00Z	SALE	*	Katherine Roth	Salmon and Wheat Bran Salad	Salad	general	ANA
16	2017-01-26T19:49:00Z	SALE	*	Katherine Roth	Chutney	Sides	Chutney	NA

Showing 1 to 16 of 8,899 entries, 21 total columns

R Console:

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Redesign/MOOC Materials/Mod 4: How to process data?/
> ##### CHARACTER STRINGS #####
> # Install and load the string package----
> # install.packages('stringr')
> library(stringr)
> # Simple example-----

```

Let's go ahead and quickly browse this data frame. Mhm. We can see that about the first 15 columns contain character string values. The remainder are numeric values. What are some of the business analytic functions that we'd want to perform on character string columns? Well, perhaps we'd want to separate out the date from the time and the Time Column. Perhaps we want to create some new categories based on certain key names in the menu. Items in the line Item column. Maybe we would want to analyze the cardholder names. There's a variety of other analytics that we may want to perform. That's just a sample, all right.

```
8 # Read in data
9
10 > j17i <- read.csv('jan17Items.csv')
11 > View(j17i)
12 > # Length----
13 > str_length(testString)
14 [1] 15
15 >
```



I

The first function that I want to show you is the string length function, and this will tell us the number of characters in a string. So I'll go ahead and run the string length function on the test string object and notice that it tells us that this character stream contains 15 characters in it. Now, all of the functions in the stringer package are vectorized, meaning it will perform that same operation on every observation of a vector. So if we want to calculate the number of characters in every line item for every observation, we can easily do that.

```
8
9
10 > # Read in data----
11 > j17i <- read.csv('jan17Items.csv')
12
13 > str_length(string)
14 > str_length(testString)
15 > str_length(j17i$LineItem)
16 > # Case----
17 > str_to_title(testString)
18 > str_to_lower(testString)
19 > str_to_upper(testString)
20
21 > # Detect➡
22 > # Replace➡
23 > # Split➡
```



I

And here's how we would do that. You can see that what has returned is a whole bunch of numbers. Now, this is really a vector. So if we want to create a new column so that we can analyze it, let's say we want to evaluate. If the length of the line item name influences the amount of sales, we could easily do that.

The screenshot shows the RStudio interface. On the left is a data frame with columns: Tax, TotalDue, and lineItemLength. The data is as follows:

Tax	TotalDue	lineItemLength
NaN	NaN	9
1.06	14.48	10
1.09	15.03	27
0.23	3.15	14
0.38	5.25	31
1.13	15.54	21
1.13	15.54	21
1.22	17.02	22

To the right of the data frame is a video player showing a man speaking. The video controls include a play button, a progress bar, and a volume icon. The video title is "j17i".

So let's create a new column and let's call it line item length and let's go ahead and look at our data frame here and scroll to the far right and we can see the line item length is created, and sure enough, it takes the length for the line item and stores it as an American value. Very good.

```
[901] 14 27 22 4 7 14 17 22 27 4 6 25 31 22 17
[946] 27 6 26 22 27 23 6 27 23 22 14 25 21 21 21
[991] 6 10 21 14 4 17 27 7 27 7
[ reached getOption("max.print") -- omitted 7899 e
> j17i$lineItemLength <- str_length(j17i$LineItem)
> # Case----
> str_to_title(testString)
[1] "This Is So Fun."
> str_to_lower(testString)
[1] "this is so fun."
> str_to_upper(testString)
[1] "THIS IS SO FUN."
> |
```



Let's go ahead and move on to case functions, so case refers to whether you're using upper case or lower case values for character strings. There are three case functions. There's title lower and upper. Here's the title. The strength to title function you can see what it did. Is it capitalize? The first letter of each word. The string to lower just converts every character to lowercase string to upper. It converts every character to uppercase. Case conversion is important In business analytics, for instance, let's say we want to create a new column in our data frame that tells us whether or not the line item name has the word kebab in it.



```

13 # Length---
14 str_length(testString)
15 j17i$lineItemLength <- str_length(j17i$LineItem)
16 # Case---
17 str_to_title(testString)
18 str_to_lower(testString)           str_to_lower(string, locale = "en")
19 str_to_upper(testString)
20 j17i$lineItemLower <- str_to_lower(j17i$LineItem)
21 # Detect---
22 str_detect(testString, 'his')
23
24
25 # Replace
26 # Split

```

21:1 # Detect

Console Terminal × R Markdown × Jobs ×

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon)

Well, the first thing we want to do is explore the line item names and find out if Kabob is spelled using uppercase, sometimes lower case sometimes or title case. If we think that might be the case, then we want to convert everything to either lower or upper case. So let's go ahead and do that.



TotalDue	lineItemLength	lineItemLower
15.54	21	beer and squash kabob
15.54	21	beef and squash kabob
17.02	23	chicken and onion kabob
14.17	21	beef and squash kabob
NaN	22	beef and apple burgers
28.42	22	beef and apple burgers
3.15	4	naan
14.17	21	beef and squash kabob
18.41	23	chicken and onion kabob
19.92	27	salmon and wheat bran salad
3.15	7	chutney
33.80	27	salmon and wheat bran salad
7.43	6	yogurt

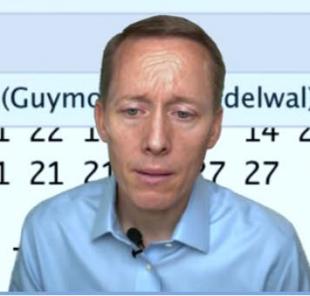
Import Data
Global Environment
Data
j17i
Values
testString

Let's convert everything in the line item column to a lower case value and will create a new column. We'll call it line item lower. We'll use a string to lower function on the line Item column. Let's go ahead and take a quick look at our data frame now, and sure enough, we can verify that the line items are all lower case.

```
[1] "This Is So Fun."  
> str_to_lower(testString)  
[1] "this is so fun."  
> str_to_upper(testString)  
[1] "THIS IS SO FUN."  
> j17i$lineItemLower <- str_to_lower(j17i$LineItem)  
> # Detect----  
> str_detect(testString, 'his')  
[1] TRUE  
> # Detect----  
> str_detect(testString, 'His')  
[1] FALSE  
>
```



Let's talk about the detect function. The detect function allows us to detect if a certain sub string exists. So a sub string is just a sequence of characters that may appear anywhere in a character string. In this case, we're going to detect if h i s appears in our testing object. Notice that it returns a value of true. The only other value will return is false. And why does it return true? Well, our testing object is a character string that says this is so fun and h i s is in there. Now let's see what happens if we run this with a capital H i s. It returns the value of false. That's important. So the detect function is case sensitive.



```

20 j17i$lineItemLower <- str_to_lower(j17i$LineItem)
21 # Detect---- str_detect(string, pattern, negate = FALSE)
22 str_detect(testString, 'H1S')
23 j17i$kabob <- str_detect(j17i$lineItemLower, 'kabob')
24
25 # Replace---- str_replace(testString, 'so', 'really really')
26 testString <- str_replace(testString, 'so', 'really real')
27
28
23:52 # Detect

```

Console Terminal × R Markdown × Jobs ×

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon)/

[901]	14	27	22	4	/	14	17	22	27	4	0	25	31	22	1	14	27	
[946]	27	6	26	22	27	23	6	27	23	22	14	25	21	21	21	27	27	6
[991]	6	10	21	14	4	17	27	7	27	7								

[reached getOption("max.print") -- omitted]

Let's go ahead and use this now. And let's create a new column in our data frame called kabob that indicates whether or not the line item menu contains kabob in its name. Somewhere I'm going to use the line item lower column and I'll go ahead and run this. Let's look at the results.



gth	lineItemLower	kabob
9	glass mug	FALSE
10	lamb chops	FALSE
27	salmon and wheat bran salad	FALSE
14	fountain drink	FALSE
31	aubergine and chickpea vindaloo	FALSE
21	beef and squash kabob	TRUE
21	beef and squash kabob	TRUE
23	chicken and onion kabob	TRUE
21	beef and squash kabob	TRUE

Environment Hist Import D

Global Environment

Data

j17i

Values

testString

Yeah, and as we look at this kabob column here, we can see false, false, false, true, and I think it's always a good idea to just do a quick visual inspection and we can see that for these true values here they do have kebab in it for the false values they do not. So it appears to be working.



```
24
25 # Replace----
str_replace(testString, 'so', 'really really')
27 testString <- str_replace(testString, 'so', 'really re
28 str_replace(testString, 'really', 'so')
29 str_replace_all(testString, 'really', 'so')
30
31 # Split
26:46 # Replace
```

Console Terminal R Markdown Jobs

```
~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guyr... andelw...
[991] 6 10 21 14 4 17 27 7 27 7
[ reached getOption("max.print") -- omitted 7890 entries ]
> j17i$lineItemLength <- str_length(j17i$Line...
> # Case----
```

The next function I want to show you is replaced. Replace is like the find and replace so it will detect a sub string and replace it with whatever characters during you identify. So let's go ahead and run string. Replace on the testing object and we will identify any instance of Esso and replace it with really, really so you can see it printed out. This is really, really fun. Now I'm going to go ahead and replace what's in the testing object with this updated values so that now that test ring does say this is really, really fun.



```

24
25 # Replace----
26 str_replace(testString, 'so', 'really really')
27 testString <- str_replace(testString, 'so', 'really really')
28 str_replace(testString, 'really', 'so')
str_replace_all(testString, 'really', 'so')
29
30 # Split
29:16 # Replace

```

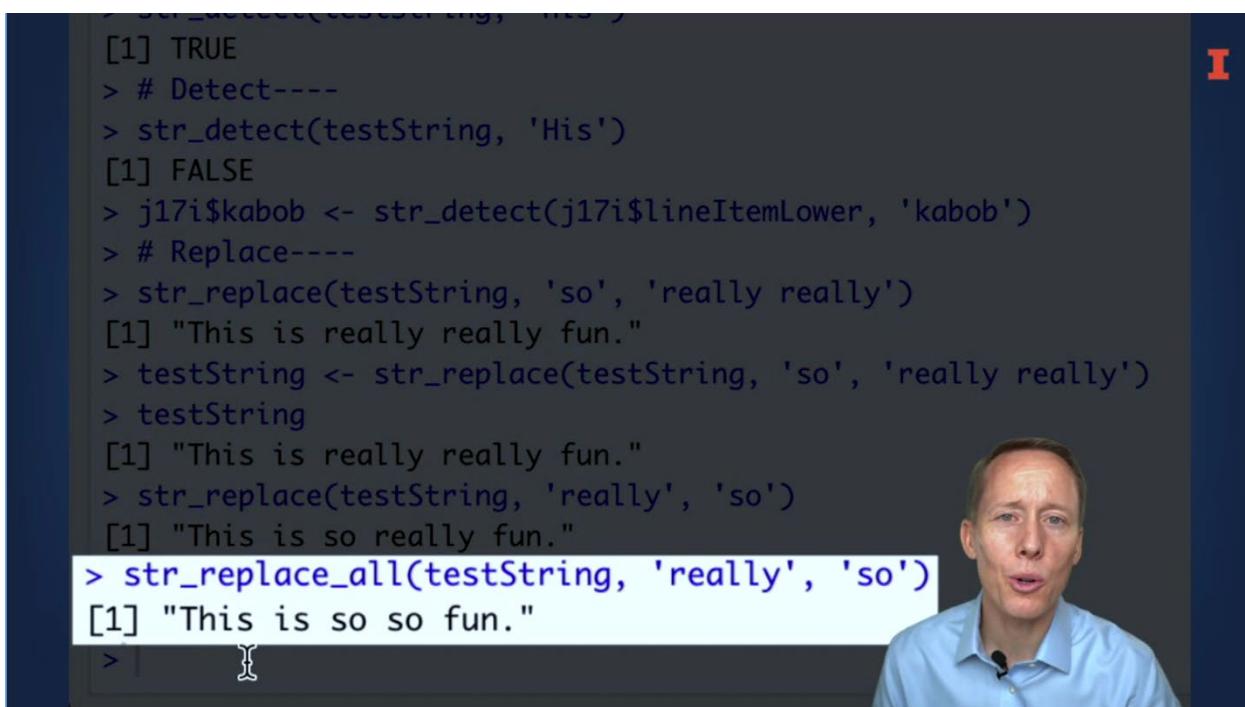
Console Terminal × R Markdown × Jobs ×

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/M
L1 THIS IS SO FUN.
> str_to_upper(testString)
[1] "THIS IS SO FUN."
> j17i$lineItemLower <- str_to_lower(j17i$LineItem)
> # Detect----
> str_detect(testString, 'his')

```

Now I want to go back and replace every instance of the word really with. So I'll run the string, replace function again on the test string object and will replace really with. So I notice what happens when I ran this. It only replace the first instance of really with. So if we want to replace every occurrence of a sub string, then we need to use the string, replace all function, and so I'll try this again.



```

> str_detect(testString, 'His')
[1] TRUE
> # Detect----
> str_detect(testString, 'His')
[1] FALSE
> j17i$kabob <- str_detect(j17i$lineItemLower, 'kabob')
> # Replace----
> str_replace(testString, 'so', 'really really')
[1] "This is really really fun."
> testString <- str_replace(testString, 'so', 'really really')
> testString
[1] "This is really really fun."
> str_replace(testString, 'really', 'so')
[1] "This is so really fun."
> str_replace_all(testString, 'really', 'so')
[1] "This is so so fun."
>

```

This time it worked. It says this is so, so fun. So let's talk about a business analytic application of this. I'm going to introduce a new function to that you may not have heard of.

The screenshot shows an RStudio interface. The code editor contains the following R code:

```
27 testString <- str_replace(testString, 'so', ' ')
28 str_replace(testString, 'really', 'so')
29 str_re[unique(x, incomparables = FALSE, ...)', 'so')
30
31 unique(j17i$LineItem)
32 # Split-----
33 str_split(testString, ' ')
34
```

Line 29 is highlighted with a yellow background. Below the code editor, the status bar shows "31:21 # Replace". The RStudio tabs at the top are "Console", "Terminal", "R Markdown", and "Jobs". In the "Console" tab, the output shows:

```
~/Box Sync/(Focus Area 4) Business Analytics/1. Council Guymon
LJ THIS IS SO FUN.
> j17i$lineItemLower <- str_to_lower(j17i$lineItem)
```

A video overlay of Professor Ronald Guymon is visible in the bottom right corner, looking slightly concerned or confused.

And that is the unique function. Unique function will take only the unique instances of certain values and report them. So I want to look at the unique values of the line item column. If we go down here in the council, we can see that there are 33, 34 unique line items. And let's pay attention to this.

The screenshot shows an RStudio interface. In the top right corner, there is a red 'I' logo. The code editor contains the following R script:

```

32 j17i$LineItem <- str_replace_all(j17i$LineItem, "-", " ")
33 # Split----
34 str_split(testString, ' ')
35
36
32:1 Replace ↴

```

The console output shows:

```

[1] "This is really really fun."
> testString <- str_replace(testString, 'so', 'really really')
[1] "This is really really fun."
> str_replace(testString, 'really', 'so')
[1] "This is so really fun."
> str_replace_all(testString, 'really', 'so')
[1] "This is so fun."
> unique(j17i$LineItem)
[1] "Glass Mar"          "Lamb Chops"        "Salmon and Wheat Bran Salad" "Fountain Drink"
[5] "Aubergine and Chickpea Vindaloo" "Beef and Squash Kabob"   "Chicken and Onion Kabob"    "Beef and Apple Burgers"
[9] "Naan"               "Chutney"           "Yogurt"                  "Rice"
[13] "Lamb and Veggie Kabob"       "Beef and Broccoli"     "Beef and Broccoli Stir Fry"  "Roll"
[17] "Beef Stew"           "Soya and Basil Salad" "Coconut and Beef Vindaloo"  "Pork"
[21] "Coconut Sauce Quart"      "Curry"              "Chips"                   "Coco"
[25] "Catering"            "Spring Greens"      "Larry's Commissary Shirt"  "Water"
[29] "Beef and Broccoli Side"    "Gift Cards"         "Grow-Your-Own-Mushroom Kit" "Chris"
[33] "Christmas Eve Vindaloo"    "Christmas Eve Roast Beef" "Salmon and Wheat Bran Salad" "Four"

```

A tooltip window titled "Simple, Convenient Wrappers for Common String Operations" lists various string manipulation functions like str_locate, str_replace, etc.

A blue box at the bottom of the screen contains the text: "Grow-Your-Own-Mushroom Kit"

Grow your own mushroom kit, which is kind of a funny item anyway. But notice that it has three in three different hyphens in it. Let's say we want to replace those for some reason, with just spaces. We can easily do that and let's go ahead and store it back into the line item column.

The screenshot shows the continuation of the RStudio session. The code has been run, and the console output now includes the modified line item:

```

[1] "Glass Mar"          "Lamb Chops"        "Salmon and Wheat Bran Salad" "Fountain Drink"
[5] "Aubergine and Chickpea Vindaloo" "Beef and Squash Kabob"   "Chicken and Onion Kabob"    "Beef and Apple Burgers"
[9] "Naan"               "Chutney"           "Yogurt"                  "Rice"
[13] "Lamb and Veggie Kabob"       "Beef and Broccoli"     "Beef and Broccoli Stir Fry"  "Roll"
[17] "Beef Stew"           "Soya and Basil Salad" "Coconut and Beef Vindaloo"  "Pork"
[21] "Coconut Sauce Quart"      "Curry"              "Chips"                   "Coco"
[25] "Catering"            "Spring Greens"      "Larry's Commissary Shirt"  "Water"
[29] "Beef and Broccoli Side"    "Gift Cards"         "Grow-Your-Own-Mushroom Kit" "Chris"
[33] "Christmas Eve Vindaloo"    "Christmas Eve Roast Beef" "Salmon and Wheat Bran Salad" "Four"

```

A blue box at the bottom of the screen contains the text: "Grow Your Own Mushroom Kit"

So I need to use the string replace all function, and I will replace the minus with or the hyphen with a space. Now go ahead and run that unique J 17. I line item row and go down and look at the girl. Your own mushroom kit looks like it did what we expected it to do. Very good.

```
[29] "Beef and Broccoli Side"           "Gift Cards"
[33] "Christmas Eve Vindaloo"         "Christmas Eve Roast Beef"
> j17i$LineItem <- str_replace_all(j17i$LineItem, '-', ' ')
> unique(j17i$LineItem)
[1] "Glass Mug"                      "Lamb Chops"
[5] "Aubergine and Chickpea Vindaloo" "Beef and Squash Kabob"
[9] "Naan"                           "Chutney"
[13] "Lamb and Veggie Kabob"          "Beef and Broccoli"
[17] "Beef Stew"                      "Soya and Basil Salad"
[21] "Coconut Sauce Quart"           "Curry"
[25] "Catering"                       "Spring Greens"
[29] "Beef and Broccoli Side"          "Gift Cards"
[33] "Christmas Eve Vindaloo"         "Christmas Eve Ro
> # Split---
> str_split(testString, ' ')
[[1]]
[1] "This" "is"    "really" "really" "fun."
> |
```



The last function I want to demonstrate is the split function. The split function takes a character string and splits it into separate elements. Let's go ahead and test this out on the test during object, and we're going to use a space as a delimiter. So as I run this notice, what it prints out is a vector with five different elements. This is really, really fun. So why would you want to use this function on business data?

The screenshot shows a Jupyter Notebook interface. On the left, a code cell displays a data frame named 'Time' with 12 rows and columns for BarCode, CashierName, and LineItem. A cursor points to the date-time string in the first row. On the right, a video player window shows a man speaking.

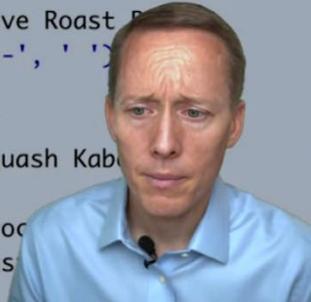
	BarCode	CashierName	LineItem
*	Nicholas Villines	Glass Mug	
*	Carla Knotts	Lamb Cho	
*	Carla Knotts	Salmon an	
*	Carla Knotts	Fountain D	
*	Carla Knotts	Aubergine	
*	Carla Knott*	Beef and S	
*	Carla Ki	Beef and S	
*	Carla K	Chicken an	
*	Carla K	Beef and S	
*	Kath	Beef and A	
*		nd A	
*			

There's all sorts of reasons why one example here is if we want to look at the time column and extract something out of that so we can see that in this column here we've got the date and then a capital T, and then the time. Let's say we want to separate out the date from the time and create two different columns in this data frame.

The screenshot shows a Jupyter Notebook interface. On the left, a code cell displays a data frame with a 'kabob' column (containing TRUE/FALSE) and a 'dateTime' column (containing date-time strings). A cursor highlights the date part of the first row's dateTime value. On the right, a video player window shows a man speaking.

kabob	dateTime
FALSE	c("2017-01-26", "21:18:00Z")
FALSE	c("2017-01-26", "30:00Z")
FALSE	c("2017-01-26", "20:30:00Z")
FALSE	c("2017-01-26", "20:30:00Z")
oo	c("2017-01-26", "20:14:00Z")
TRUE	c("2017-01-26", "20:14:00Z")
TRUE	c("2017-01-26", "20:13:00Z")
TRUE	c("2017-01-26", "20:13:00Z")

Let's test it out. First, we'll create a new column date. Time were based on the time column, and we identified that there is a T in between the date and the time. So that's what we will use as our pattern or R D limiter. And I'll run this. Yeah, and the scroll to the last column. We can see that the date time column doesn't look like the other columns there, so it actually has a list within each cell here. That's not good. That won't help us. So we want to separate this out into two separate columns, so we'll need to do this in two steps.



```

30
31 unique(j17i$LineItem)
32 j17i$LineItem <- str_replace_all(j17i$LineItem, '-', ' ')
33 # Split----
34 str_split(testString, pattern, n = Inf, simplify = FALSE)
35 j17i$dateTime <- str_split(j17i$Time, 'T')
36:51 # Split

```

Console Terminal × R Markdown × Jobs ×

```

~/Box Sync/(Focus Area 4) Business Analytics/1. Course 1 (Guymon & Khandelwal)/MOOC Redesign/
[29] "Beer and Broccoli Slaw"           "GLTC Curds"          "GR
[33] "Christmas Eve Vindaloo"          "Christmas Eve Roast P
> j17i$LineItem <- str_replace_all(j17i$LineItem, '-', ' ')
> unique(j17i$LineItem)
[1] "Glass Mug"                      "Lamb Chops"          "Sa
[5] "Aubergine and Chickpea Vindaloo" "Beef and Squash Kabab" "Ch
[9] "Naan"                           "Chutney"             "Yo
[13] "Lamb and Veggie Kabob"          "Beef and Broccoli"   "Be
[17] "Beef Stew"                      "Soya and Basmati"   "Co
[21] "Coconut Sauce Quinoa"          "Curry"               "h

```

Really, the first step is to create a new date, time data frame or matrix. And so we'll use the string split function. And we also need to use this other argument to simplify argument and set it equal to true. So now, if I look at this date time Matrix here, it has two columns.

V1	V2
1 2017-01-26	21:18:00Z
2 2017-01-26	20:30:00Z
3 2017-01-26	20:30:00Z
4 2017-01-26	20:30:00Z
5 2017-01-26	20:14:00Z
6 2017-01-26	20:14:00Z
7 2017-01-26	20:13:00Z
8 2017-01-26	20:02:00Z
9 2017-01-26	20:02:00Z
10 2017-01-26	20:00:00Z
11 2017-01-26	20:00:00Z
12 2017-01-26	20:00:00Z
13 2017-01-26	20:00:00Z
14 2017-01-26	19:52:00Z
15 2017-01-26	19:51:00Z
16 2017-01-26	19:49:00Z
17 2017-01-26	19:49:00Z

Showing 1 to 17 of 8,899 entries, 2 total columns

First column with the dates second column with the time that's perfect. We want that.

```

33 > # Split
34 str_split(testString, ' ')
35 j17i$dateTime <- str_split(j17i$time, 'T')
36 dateTime <- str_split(j17i$time, 'T', simplify = TRUE)
37 j17i$date <- dateTime[,1]
38 j17i$time <- dateTime[,2]
38:25 # Split

```

Console Terminal × R Markdown × Jobs ×

~/Box Sync/(Focus Area 4) Business Analytics/1. Course Materials/Tuymon> unique(j17i\$dateTime)

[1] "Glass Mug"
[5] "Aubergine and Chickpea Vindaloo"
[9] "Naan"
[13] "Lamb and Veggie Kabob"

So let's go ahead and store this each each one of those columns in a new column in the J 79 data frame, the first one will be the date, so J 79 date will get the value from the daytime matrix that is in the first column j 79 time.



	dateTime	date	time
	c("2017-01-26", "21:18:00Z")	2017-01-26	21:18:00Z
	c("2017-01-26", "20:30:00Z")	2017-01-26	20:30:00Z
	c("2017-01-26", "20:30:00Z")	2017-01-26	20:30:00Z
	c("2017-01-26", "20:30:00Z")	2017-01-26	20:30:00Z
	c("2017-01-26", "20:14:00Z")	2017-01-26	20:14:00Z
	c("2017-01-26", "20:14:00Z")	2017-01-26	20:14:00Z
	c("2017-01-26", "20:13:00Z")	2017-01-26	20:13:00Z
	c("2017-01-26", "20:02:00Z")	2017-01-26	20:02:00Z
	c("2017-01-26", "20:02:00Z")	2017-01-26	20:02:00Z

We'll get the second column and let's go ahead and look at our data frame here. And sure enough, we've got the date in one column in the time in another column so you can explore many other functions in the stringer package. It's very powerful. I hope this is a good introduction that gives you an idea of how to get started manipulating and wrangling character strings in our

Module 3 Conclusion

Module 3: Conclusion

The video frame features a dark blue background with a red 'I' logo in the top right corner. On the left, the title 'Technical Aspects of Assembling Data' is displayed in white, bold, sans-serif font. To the right, Professor Ronald Guymon is shown from the chest up, wearing a dark blue button-down shirt. He has his hands slightly open and is gesturing while speaking. The overall composition is a professional video recording.

Technical Aspects of Assembling Data

1. Installing packages
2. Creating and using functions
3. Dealing with non-numeric data types

[MUSIC] As we conclude this module, your mind is probably focused on technical aspects of assembling data, installing packages, creating and using functions, and dealing with non numeric data types. Being focused on these details is both a blessing and a curse. It's a blessing because you are now better equipped to identify a number of small potential issues that could prevent a business analytic project from moving forward.

Big Picture Concepts

1. Assembling data is critical.



A favorite saying of mine is, it's not the Lions and Tigers, it's the gnats and skeeters that get you. Knowing how to deal with the gnats and Skeeters, or at least being aware that they exist, is really important, knowing that these little things can be a curse if it prevents you from being able to step back and see the bigger picture. So here are some key insights, first, assembling data may not be the most glamorous part of business analytics, but it's a critical part of the process. Just as photographer spend time setting up the lighting before taking a picture in a studio, business analysts need to spend a good amount of time assembling data into meaningful rows and columns.

Big Picture Concepts

1. Assembling data is critical.
2. Non-numeric data should not be ignored.



Another big picture concept is data type, numeric data is probably the most important because algorithms depend on converting data to numbers. However, non numeric data types should not be ignored. I hope the gaining experience working with dates and times, character strings and factors allows you to work more efficiently with data as well as manage those who work with it.

Big Picture Concepts

1. Assembling data is critical.
2. Non-numeric data should not be ignored.
3. Functions help process data efficiently.



Finally, functions are essential to business analytics because they're what makes it possible to assemble and process data very efficiently. Specifically, functions make it possible for us to

perform common tasks with numeric and non numeric data types without having to re-type large amounts of code and without even knowing the low level code. Perhaps the most important lesson about functions is knowing how to get.