COURSE 7

MODULE 2 - PROGRAMMING USING RSTUDIO

Using R can help you complete your analysis efficiently and effectively. In this part of the course, you’ll explore the fundamental concepts associated with R. You’ll learn about functions and variables for calculations and other programming. In addition, you'll discover R packages, which are collections of R functions, code and sample data that you’ll use in RStudio.

### Learning Objectives

* Describe the contents and components of the tidyverse package for R
* Describe the concept of packages in R programming language
* Describe the use of operators to complete calculations in the R programming language
* Describe the fundamental concepts associated with programming in R including functions, variables, data types, pipes, and vectors
* Install and load the tidyverse package
* Use the browseVignettes("packagename") function to read through vignettes of a loaded package
* Locate resources for help using R

BASIC PROGRAMMING CONCEPTS

[Programming using RStudio](https://www.coursera.org/learn/data-analysis-r/lecture/z5rIt/programming-using-rstudio)

We've given you a big-picture overview of R and RStudio. Now we'll turn our focus to the actual programming and coding you'll do using RStudio. I went pretty far in my career not knowing programming before it became clear, I needed to learn it. Getting to know R was such a valuable learning experience. It took some time, and I reached out to more-experienced R users with lots of questions. Eventually, it all came together for me. Being open to learning new skills is such an important part of your career. Now I'm able to help you learn some new skills too. I'll start by sharing the fundamentals of programming using R in RStudio. Earlier, we explained how R is like the engine of a car and RStudio is like the accelerator, steering wheel, and dashboard all in one. Getting to know fundamentals will help you keep your R car running smoothly. These fundamentals are both alike and different from the other analysis platforms you've come to know well: spreadsheets and SQL. Then we'll move on to coding in RStudio. We'll discuss the syntax for performing calculations and the standards and naming conventions for all code. We'll also explore the R tool known as a pipe, which you'll use to make a sequence of code easier to work with and read. Then we'll check out R packages. While these packages won't be delivered to your door, they are delivered by the R community. These packages contain reusable functions and more, and are usually built by users for users like yourself. We'll get to know a collection of packages called the Tidyverse. You'll learn how to install the Tidyverse so you can start using it in RStudio. We'll also work with some of the more popular Tidyverse packages like ggplot2 for visualization. You'll be able to carry over what you've learned about RStudio to the next part of the program, where you'll start working with data. As we explained earlier, for this program, we'll use the in-browser version of RStudio: RStudio Cloud. But RStudio is also available to be downloaded. So let's get going. See you soon.

[Programming fundamentals](https://www.coursera.org/learn/data-analysis-r/lecture/iMoxv/programming-fundamentals)

Anytime you're learning a new skill from cooking to driving to dancing, you should always start with the fundamentals. Programming with R is no different. To build this foundation, you'll get familiar with the basic concepts of R, including functions, comments, variables, data types, vectors, and pipes. Some of these terms might sound familiar. For example, we've come across functions in spreadsheets and SQL. As a quick refresher, functions are a body of reusable code used to perform specific tasks in R. Functions begin with function names like print or paste, and are usually followed by one or more arguments in parentheses. An argument is information that a function in R needs in order to run. Here's a simple function in action. Feel free to join in and try it yourself in RStudio using your cloud account. Check out the reading for more details on how to get started.

You can pause the video anytime you need to. We'll open RStudio Cloud to get started. We'll start our function in the console with the function name print. This function name will return whatever we include in the values in parentheses. We'll type an open parenthesis followed by a quotation mark. Both the close parenthesis and end quote automatically pop up because RStudio recognizes this syntax. Now we just have to add the text string. We'll type Coding in R.

Then we'll press enter.

Success! The code returns the words "Coding in R." If you want to find out more about the print function or any function, all you have to do is type a question mark, the function name, and a set of parentheses.

This returns a page in the Help window, which helps you learn more about the functions you're working with. Keep in mind that functions are case-sensitive, so typing Print with a Capital P brings back an error message.

Functions are great, but it can be pretty time-consuming to type out lots of values.

To save time, we can use variables to represent the values.

This lets us call out the values any time we need to with just the variable. Earlier, we learned about variables in SQL. A variable is a representation of a value in R that can be stored for use later during programming. Variables can also be called objects. As a data analyst, you'll find variables are very useful when programming. For example, if you want to filter a dataset, just assign a variable to the function you used to filter the data. That way, all you have to do is use that variable to filter the data later. When naming a variable in R, you can use a short phrase.

**A variable name should start with a letter and can also contain numbers and underscores.**

So the variable 5 penguin wouldn't work well because it starts with a number.

**Also just like functions, variable names are case-sensitive.**

Using all lowercase letters is good practice whenever possible. Now, before we get to coding a variable, let's add a comment. Comments are helpful when you want to describe or explain what's going on in your code. Use them as much as possible so that you and everyone can understand the reasoning behind it.

**Comments should be used to make an R script more readable. A comment shouldn't be treated as code, so we'll put a # in front of it**.

Then we'll add our comments. Here's an example of a variable.

Now let's go ahead with our example. It makes sense to use a variable name to connect to what the variable is representing. So we'll type the variable name first\_variable.

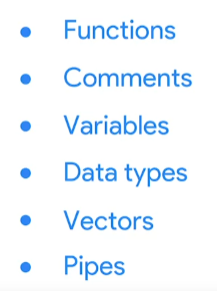
Then after the variable name, we'll type a < sign, followed by a -.

This is the assignment operator. It assigns the value to the variable. It looks like an arrow, which makes sense, since it's pointing from the value to the variable. There are other assignment operators that work too, but it's always good to stick with just one type in your code. Next, we'll add the value that our variable will represent. We'll use the text, "This is my variable."

If we type the variable and hit Run, it will return the value that the variable represents. This is a very basic way of using a variable. You'll learn more ways of using variables in your code soon. For now, let's assign a variable to a different data type, numeric. We'll name this second\_variable, and type our assignment operator. We'll give it the numeric value 12.5.

The Environment pane in the upper- right part of our work space now shows both of our variables and their values. There are other data types in R like logical, date, and date time. R has a few options for dealing with these data types. We'll explore them later. With functions, comments, variables, and data types, you've got a good foundation for working with R. We'll revisit these throughout this program, and show you how they're used in different ways during analysis.

Let's finish up with two more fundamental concepts, vectors and pipes.



Simply put, **a vector is a group of data elements of the same type stored in a sequence in R**. You can make a vector using the combined function. In R this function is just the letter c followed by the values you want in your vector inside parentheses. All right, let's create a vector. Imagine this vector is for a measurement data that we need to analyze. We'll start our code with the variable vec\_1 to assign to the vector.

Then we'll type c and the open parenthesis.Then we'll type our list of numbers separated by commas. We'll then close our parentheses and press enter.

This time when we type our variable and press enter, it returns our vector. We can use this vector anywhere in our analysis with only its variable name vec\_1. The values in the vector will automatically be applied to our analysis. That brings us to the last of our fundamentals, pipes.



**A pipe is a tool in R for expressing a sequence of multiple operations.**

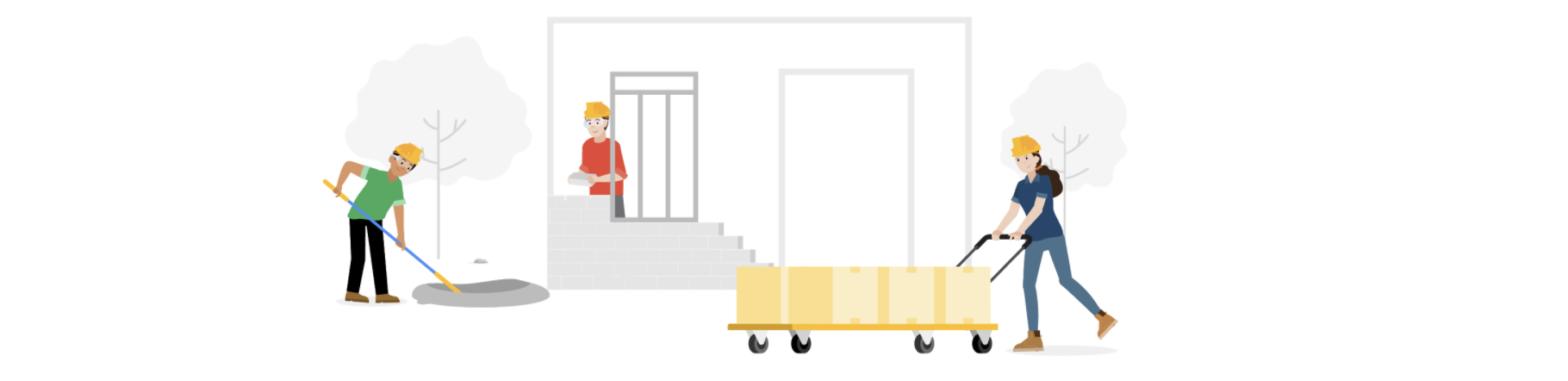
A pipe is represented by a % sign, followed by a > sign, and another % sign. It's used to apply the output of one function into another function. Pipes can make your code easier to read and understand. For example, this pipe filters and sorts the data. Later, we'll learn how each part of the pipe works. So there they are, the super six fundamentals: functions, comments, variables, data types, vectors, and pipes. They all work together as a foundation for using R. It's a lot to take in, so feel free to watch any of these videos again if you need a refresher. When you're ready, there's so much more to know about R and RStudio. So let's get to it.

[Vectors and lists in R](https://www.coursera.org/learn/data-analysis-r/supplement/7dRY6/vectors-and-lists-in-r)

In programming, a **data structure** is a format for organizing and storing data. Data structures are important to understand because you will work with them frequently when you use R for data analysis. The most common data structures in the R programming language include:

* Vectors
* Data frames
* Matrices
* Arrays

Think of a data structure like a house that contains your data.



This reading will focus on vectors. Later on, you’ll learn more about data frames, matrices, and arrays.

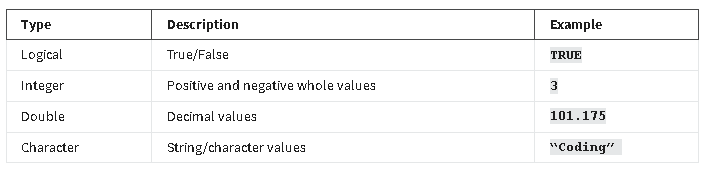
There are two types of vectors: **atomic vectors** and **lists**. Coming up, you’ll learn about the basic properties of atomic vectors and lists, and how to use R code to create them.

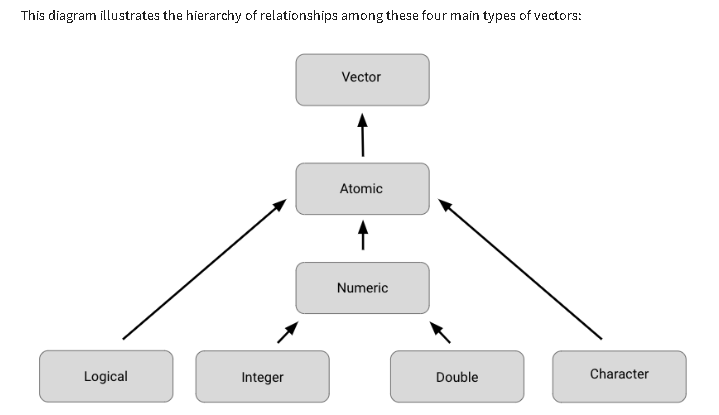
## **Atomic vectors**

First, we will go through the different types of atomic vectors. Then, you will learn how to use R code to create, identify, and name the vectors.

Earlier, you learned that a **vector** is a group of data elements of the *same* type, stored in a sequence in R. You cannot have a vector that contains both logicals and numerics.

There are six primary types of atomic vectors: logical, integer, double, character (which contains strings), complex, and raw. The last two–complex and raw–aren’t as common in data analysis, so we will focus on the first four. Together, integer and double vectors are known as numeric vectors because they both contain numbers. This table summarizes the four primary types:





### **Creating vectors**

One way to create a vector is by using the **c()** function (called the “combine” function). The c() function in R combines multiple values into a vector. In R, this function is just the letter “c” followed by the values you want in your vector inside the parentheses, separated by a comma: c(x, y, z, …).

For example, you can use the c() function to store numeric data in a vector.

**c(2.5, 48.5, 101.5)**

To create a vector of integers using the c() function, you must place the letter "L" directly after each number.

**c(1L, 5L, 15L)**

You can also create a vector containing characters or logicals.

**c(“Sara” , “Lisa” , “Anna”)**

**c(TRUE, FALSE, TRUE)**

### **Determining the properties of vectors**

Every vector you create will have two key properties: type and length.

You can determine what type of vector you are working with by using the **typeof()** function. Place the code for the vector inside the parentheses of the function. When you run the function, R will tell you the type. For example:

**typeof(c(“a” , “b”))**

**#> [1] "character"**

Notice that the output of the typeof function in this example is **“character”**. Similarly, if you use the typeof function on a vector with integer values, then the output will include **“integer”** instead:

**typeof(c(1L , 3L))**

**#> [1] "integer"**

You can determine the length of an existing vector–meaning the number of elements it contains–by using the **length()** function. In this example, we use an assignment operator to assign the vector to the variable *x*. Then, we apply the length() function to the variable. When we run the function, R tells us the length is **3**.

**x <- c(33.5, 57.75, 120.05)**

**length(x)**

**#> [1] 3**

You can also check if a vector is a specific type by using an **is** function: **is.logical(), is.double(), is.integer(), is.character()**. In this example, R returns a value of **TRUE** because the vector contains integers.

**x <- c(2L, 5L, 11L)**

**is.integer(x)**

**#> [1] TRUE**

In this example, R returns a value of **FALSE** because the vector does *not* contain characters, rather it contains logicals.

**y <- c(TRUE, TRUE, FALSE)**

**is.character(y)**

**#> [1] FALSE**

### **Naming vectors**

All types of vectors can be named. Names are useful for writing readable code and describing objects in R. You can name the elements of a vector with the **names()** function. As an example, let’s assign the variable x to a new vector with three elements.

**x <- c(1, 3, 5)**

You can use the names() function to assign a different name to each element of the vector.

**names(x) <- c("a", "b", "c")**

Now, when you run the code, R shows that the first element of the vector is named **a**, the second **b**, and the third **c**.

**x**

**#> a b c**

**#> 1 3 5**

Remember that an atomic vector can only contain elements of the same type. If you want to store elements of different types in the same data structure, you can use a list.

## **Creating lists**

**Lists** are different from atomic vectors because their elements can be of any type—like dates, data frames, vectors, matrices, and more. Lists can even contain other lists.

You can create a list with the **list()** function. Similar to the c() function, the list() function is just **list** followed by the values you want in your list inside parentheses: **list(x, y, z, …)**. In this example, we create a list that contains four different kinds of elements: character (**"a"**), integer (**1L**), double (**1.5**), and logical (**TRUE**).

**list("a", 1L, 1.5, TRUE)**

Like we already mentioned, lists can contain other lists. If you want, you can even store a list inside a list inside a list—and so on.

**list(list(list(1 , 3, 5)))**

### **Determining the structure of lists**

If you want to find out what types of elements a list contains, you can use the **str()** function. To do so, place the code for the list inside the parentheses of the function. When you run the function, R will display the data structure of the list by describing its elements and their types.

Let’s apply the str() function to our first example of a list.

**str(list("a", 1L, 1.5, TRUE))**

We run the function, then R tells us that the list contains four elements, and that the elements consist of four different types: character (**chr**), integer (**int**), number (**num**), and logical (**logi**).

**#> List of 4**

**#> $ : chr "a"**

**#> $ : int 1**

**#> $ : num 1.5**

**#> $ : logi TRUE**

Let’s use the str() function to discover the structure of our second example. First, let’s assign the list to the variable *z* to make it easier to input in the str() function.

**z <- list(list(list(1 , 3, 5)))**

Let’s run the function.

**str(z)**

**#> List of 1**

**#> $ :List of 1**

**#> ..$ :List of 3**

**#> .. ..$ : num 1**

**#> .. ..$ : num 3**

**#> .. ..$ : num 5**

The indentation of the **$** symbols reflect the nested structure of this list. Here, there are three levels (so there is a list within a list within a list).

### **Naming lists**

Lists, like vectors, can be named. You can name the elements of a list when you first create it with the list() function:

**list('Chicago' = 1, 'New York' = 2, 'Los Angeles' = 3)**

**$`Chicago`**

**[1] 1**

**$`New York`**

**[1] 2**

**$`Los Angeles`**

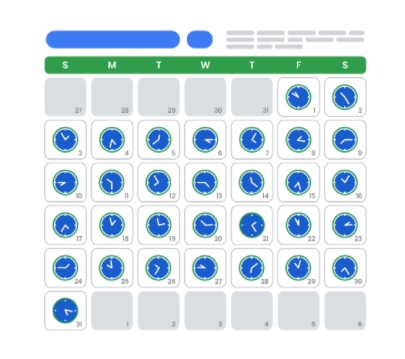
**[1] 3**

## **Additional resource**

To learn more about vectors and lists, check out [R for Data Science, Chapter 20: Vectors](https://r4ds.had.co.nz/vectors.html#vectors). R for Data Science is a classic resource for learning how to use R for data science and data analysis. It covers everything from cleaning to visualizing to communicating your data. If you want to get more details about the topic of vectors and lists, this chapter is a great place to start.

[Dates and times in R](https://www.coursera.org/learn/data-analysis-r/supplement/g0l4l/dates-and-times-in-r)

In this reading, you will learn how to work with dates and times in R using the **lubridate** package. Coming up, you will use tools in the lubridate package to convert different types of data in R into date and date-time formats.



## **Loading tidyverse and lubridate packages**

Before you get started working with dates and times, you should load both **tidyverse** and **lubridate**. Lubridate is part of tidyverse.

First, open RStudio.

If you haven't already installed tidyverse, you can use the **install.packages()** function to do so:

* **install.packages("tidyverse")**

Next, load the tidyverse and lubridate packages using the **library()** function. First, load the core tidyverse to make it available in your current R session:

* **library(tidyverse)**

Then, load the lubridate package:

* **library(lubridate)**

Now you’re ready to be introduced to the tools in the lubridate package.

## **Working with dates and times**

This section covers the data types for dates and times in R and how to convert strings to date-time formats.

### **Types**

In R, there are three types of data that refer to an instant in time:

* A date **("2016-08-16")**
* A time within a day **(“20:11:59 UTC")**
* And a date-time. This is a date plus a time **("2018-03-31 18:15:48 UTC")**

The time is given in UTC, which stands for Universal Time Coordinated, more commonly called Universal Coordinated Time. This is the primary standard by which the world regulates clocks and time.

For example, to get the current date you can run the **today()** function. The date appears as year, month, and day.

**today()**

**#> [1] "2021-01-20"**

To get the current date-time you can run the **now()** function. Note that the time appears to the nearest second.

**now()**

**#> [1] "2021-01-20 16:25:05 UTC"**

When working with R, there are three ways you are likely to create date-time formats:

* From a string
* From an individual date
* From an existing date/time object

R creates dates in the standard yyyy-mm-dd format by default.

Let's go over each.

### **Converting from strings**

Date/time data often comes as strings. You can convert strings into dates and date-times using the tools provided by lubridate. These tools automatically work out the date/time format. First, identify the order in which the year, month, and day appear in your dates. Then, arrange the letters *y*, *m*, and *d* in the same order. That gives you the name of the lubridate function that will parse your date. For example, for the date *2021-01-20,* you use the order *ymd*:

**ymd("2021-01-20")**

When you run the function, R returns the date in yyyy-mm-dd format.

**#> [1] "2021-01-20"**

It works the same way for any order. For example, month, day, and year. R still returns the date in yyyy-mm-dd format.

**mdy("January 20th, 2021")**

**#> [1] "2021-01-20"**

Or, day, month, and year. R still returns the date in yyyy-mm-dd format.

**dmy("20-Jan-2021")**

**#> [1] "2021-01-20"**

These functions also take unquoted numbers and convert them into the yyyy-mm-dd format.

**ymd(20210120)**

**#> [1] "2021-01-20"**

### **Creating date-time components**

The ymd() function and its variations create dates. To create a date-time from a date*,* add an underscore and one or more of the letters *h*, *m*, and s (hours, minutes, seconds) to the name of the function:

**ymd\_hms("2021-01-20 20:11:59")**

**#> [1] "2021-01-20 20:11:59 UTC"**

**mdy\_hm("01/20/2021 08:01")**

**#> [1] "2021-01-20 08:01:00 UTC"**

### **Optional: Switching between existing date-time objects**

Finally, you might want to switch between a date-time and a date.

You can use the function **as\_date()** to convert a date-time to a date. For example, put the current date-time—now()—in the parentheses of the function.

**as\_date(now())**

**#> [1] "2021-01-20"**

## **Additional resources**

To learn more about working with dates and times in R, check out the following resources:

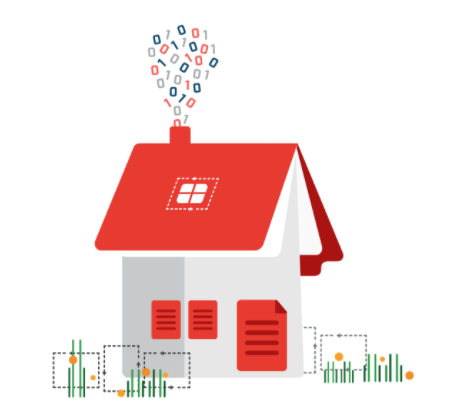
* [lubridate.tidyverse](https://lubridate.tidyverse.org/index.html): This is the “lubridate” entry from the official tidyverse documentation, which offers a comprehensive reference guide to the various tidyverse packages. Check out this link for an overview of key concepts and functions.
* [Dates and times with lubridate: Cheat Sheet](https://rawgit.com/rstudio/cheatsheets/master/lubridate.pdf): This “cheat sheet” gives you a detailed map of all the different things you can do with the lubridate package. You don’t need to know all of this information, but the cheat sheet is a useful reference for any questions you might have about working with dates and times in R.

[Other common data structures](https://www.coursera.org/learn/data-analysis-r/supplement/xEM9d/other-common-data-structures)

In this reading, you’ll continue exploring data structures through an introduction to data frames and matrices. You will learn about the basic properties of each structure, and simple ways to create them with R code. You’ll also briefly examine **files**, which are often used to access and store data and related information. The files and matrices sections of this reading are optional.

## **Data structures**

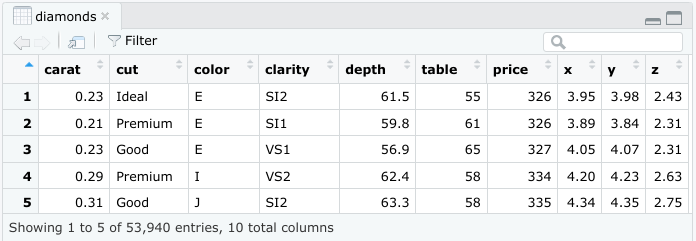
Recall that a data structure is like a house that contains your data, helping you to bring data elements together in a structured way that enables you to draw conclusions.



### **Data frames**

Data frames are the most common way of storing and analyzing data in R, so it’s important to understand what they are and how to create them. A **data frame** is a collection of columns containing data, similar to a spreadsheet or SQL table. Each column has a name that represents a variable and includes one observation per row. Data frames summarize data and organize it into a format that is easy to read and use.

For example, the data frame below shows the **diamonds** dataset, which is one of the preloaded datasets in R. Each column contains a single variable that is related to diamonds: carat, cut, color, clarity, depth, and so on. Each row represents a single observation.



There are a few key things to keep in mind when working with data frames:

* Data frames can include many different types of data, including numeric, logical, or character.
* Data frames can have only one element in each cell.
* Each column should be named.
* Each column should consist of elements of the same data type.

You will learn more about data frames later on in the program, but this is a great starting point.

If you need to manually create a data frame in R, you can use the **data.frame()** function. The **data.frame()** function takes vectors as input. In the parentheses, enter the name of the column, followed by an equals sign, and then the vector you want to input for that column. In this example, the **x** column is a vector with elements 1, 2, 3, and the **y** column is a vector with elements 1.5, 5.5, 7.5. Run the following code to create the data frame.

data.frame(x = c(1, 2, 3) , y = c(1.5, 5.5, 7.5))

Run

Reset

When you run the code, R displays the data frame in ordered rows and columns.

Use the extract operator to extract a subset from a data frame. When you use this operator on a data frame, it takes two arguments: the row(s) and column(s) you’d like to extract, separated by a comma. As an example, name the data frame above z. Then, to extract the element from the second row and the first column, use the code **z[2,1]**, which returns a value of 2:

z <- data.frame(x = c(1, 2, 3) , y = c(1.5, 5.5, 7.5))

z[2,1]

Run

Reset

You’ll learn more about data frames later on in the course, but this is enough to get you started!

## **Optional: Files**

When you’re doing data analysis, you won’t usually create a data frame yourself. Instead, you’ll import data from another source, such as a .csv file, a relational database, or a software program. For this reason, it’s essential to be able to work with files in R. In this section, you’ll explore a few of the most useful functions for working with files, including commands to create, copy, and delete files in R.

### **Create a file**

Use the **file.create()** function to create a blank file. Place the name and the type of the file in the parentheses of the function. Your file types will usually be something like .txt, .docx, or .csv.

file.create("new\_csv\_file.csv")

If the file is successfully created when you run the function, R will return a value of **TRUE**. Otherwise, R will return a value of **FALSE**.

# [1] TRUE

file.create("new\_csv\_file.csv")

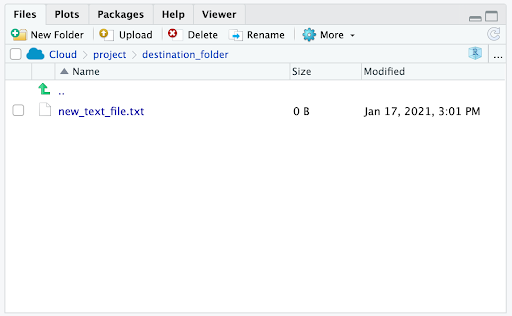
# code output:

### **Copy a file**

Copy a file with the **file.copy()** function. In the parentheses, add the name of the file to be copied. Then, enter a comma, and add the name of the destination folder that you want to copy the file to.

file.copy("new\_text\_file.txt", "destination\_folder")

If you check the **Files** tab in RStudio, a copy of the file appears in the relevant folder:



You can delete R files with the **unlink()** function. Enter the file’s name in the parentheses of the function.

unlink("some\_.file.csv")

You’ll learn techniques for importing files into R later in this course.

## **Optional: Matrices**

A **matrix** is a two-dimensional collection of data elements. This means it has both rows and columns. By contrast, a vector is a one-dimensional sequence of data elements. But like vectors, matrices can only contain a single data type. For example, you can’t have both logicals and numerics in a matrix.

To create a matrix in R, you can use the **matrix()** function. The **matrix()** function has two main arguments that you enter in the parentheses. First, add a vector. The vector contains the values you want to place in the matrix. Next, add at least one matrix dimension. You can choose to specify the number of rows or the number of columns by using the code **nrow =** or **ncol =**.

For example, to create a 2x3 (two rows by three columns) matrix containing the values 3-8, enter a vector containing that series of numbers: **c(3:8)**. Then, enter a comma. Finally, enter **nrow = 2** to specify the number of rows. Run the code:

matrix(c(3:8), nrow = 2)

Run

Reset

R displays a matrix with three columns and two rows (typically referred to as a “2x3”) that contain the numeric values 3, 4, 5, 6, 7, 8. R places the first value (3) of the vector in the uppermost row, and the leftmost column of the matrix, and continues the sequence from left to right.

You can also choose to specify the number of columns (**ncol =** ) instead of the number of rows (**nrow =** ). Run the code:

matrix(c(3:8), ncol = 2)

Run

Reset

R infers the number of rows automatically.

Similar to data frames, you can extract an element from a matrix with the extract operator, **[]**.

## **Key takeaways**

As a data analyst, you’ll work with data frames often. Data frames in R are a collection of columns containing data, similar to a spreadsheet or SQL table. Data frames can contain data of different types, although each column must be of the same data type. By contrast, matrices are a collection of two-dimensional data elements that can only contain one data type. Usually, you’ll import data into R before you analyze it, so knowing how to use R to work with files is critical. You’ll learn techniques to import files later in this course, but you can also use R functions to create, copy, and delete files.

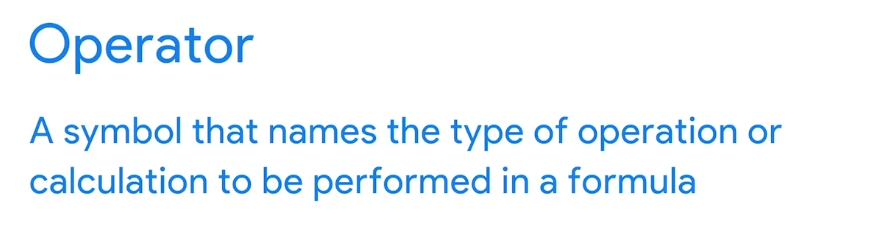
## **Resources for more information**

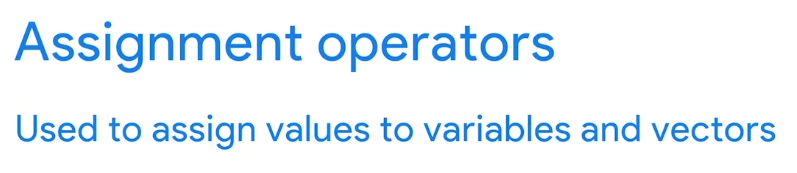
For more information on working with files in R, check out [R documentation: files](https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/files). It’s a useful reference guide for functions in R code.

[Test your knowledge on programming concepts](https://www.coursera.org/learn/data-analysis-r/quiz/ZAvv4/test-your-knowledge-on-programming-concepts)

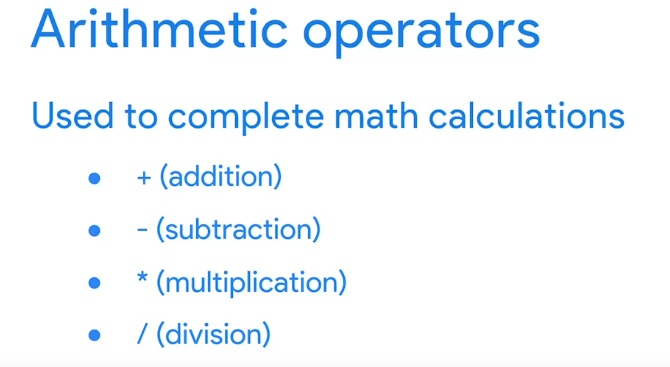
EXPLORE CODING IN R

[Operators and calculations](https://www.coursera.org/learn/data-analysis-r/lecture/quIEZ/operators-and-calculations)









# our first calculations

quarter\_1\_sales <- 35657.98

quarter\_2\_sales <- 43810.55

midyear\_sales <- quarter\_1\_sales + quarter\_2\_sales

**Note:** At closer glance, you will notice that the instructor highlighted the entire 4 lines of syntax together before running the code by pressing the "Run" button or pressing the hotkeys (**PC : CTRL + Enter** and **Mac: CMND + Enter**) .

As an alternative option, you may separately run each line of code one at a time after typing the syntax. Follow these steps in the script editor:

# our first calculations **(run the code)**

quarter\_1\_sales <- 35657.98 **(run the code)**

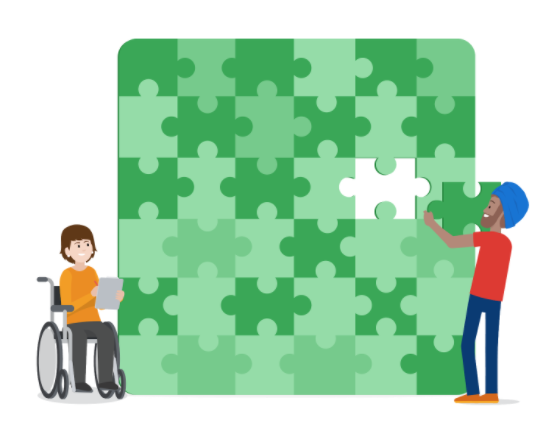
quarter\_2\_sales <- 43810.55 **(run the code)**

midyear\_sales <- quarter\_1\_sales + quarter\_2\_sales **(run the code)**

Each variable will be committed to memory in the script editor before getting to the next line, so you won't receive an error when running the final line of code containing the *midyear\_sales* variable.

[Logical operators and conditional statements](https://www.coursera.org/learn/data-analysis-r/supplement/I39VT/logical-operators-and-conditional-statements)

Earlier, you learned that an **operator** is a symbol that identifies the type of operation or calculation to be performed in a formula. In this reading, you will learn about the main types of logical operators and how they can be used to create conditional statements in R code.



## **Logical operators**

**Logical operators** return a logical data type such as TRUE or FALSE.

There are three primary types of logical operators:

* AND (sometimes represented as & or && in R)
* OR (sometimes represented as | or || in R)
* NOT (!)

Review the summarized logical operators below.

### **AND operator “&”**

* The AND operator takes two logical values. It returns **TRUE** only if both individual values are TRUE. This means that TRUE & TRUE evaluates to **TRUE**. However, FALSE & TRUE, TRUE & FALSE, and FALSE & FALSE all evaluate to **FALSE**.

If you run the corresponding code in R, you get the following results:

**> TRUE & TRUE**

**[1] TRUE**

**> TRUE & FALSE**

**[1] FALSE**

**> FALSE & TRUE**

**[1] FALSE**

**> FALSE & FALSE**

**[1] FALSE**

You can illustrate this using the results of our comparisons. Imagine you create a variable x that is equal to 10.

**x <- 10**

To check if x is greater than 3 but less than 12, you can use x > 3 and x < 12 as the values of an “AND” expression.

**x > 3 & x < 12**

When you run the function, R returns the result TRUE.

**[1] TRUE**

The first part, **x > 3** will evaluate to **TRUE** since 10 is greater than 3. The second part, **x < 12** will also evaluate to **TRUE** since 10 is less than 12. So, since both values are TRUE, the result of the AND expression is **TRUE**. The number 10 lies between the numbers 3 and 12.

However, if you make x equal to 20, the expression **x > 3 & x < 12** will return a different result.

**x <- 20**

**x > 3 & x < 12**

**[1] FALSE**

* Although **x > 3** is **TRUE** (20 > 3), **x < 12** is **FALSE** (20 < 12). If one part of an AND expression is FALSE, the entire expression is FALSE (TRUE & FALSE = FALSE). So, R returns the result **FALSE**.

### **OR operator “|”**

* The OR operator (|) works in a similar way to the AND operator (&). The main difference is that at least one of the values of the OR operation must be TRUE for the entire OR operation to evaluate to **TRUE**. This means that TRUE | TRUE, TRUE | FALSE, and FALSE | TRUE all evaluate to **TRUE**. When both values are FALSE, the result is **FALSE**.

If you write out the code, you get the following results:

**> TRUE | TRUE**

**[1] TRUE**

**> TRUE | FALSE**

**[1] TRUE**

**> FALSE | TRUE**

**[1] TRUE**

**> FALSE | FALSE**

**[1] FALSE**

For example, suppose you create a variable y equal to 7. To check if y is less than 8 or greater than 16, you can use the following expression:

**y <- 7**

**y < 8 | y > 16**

The comparison result is TRUE (7 is less than 8) | FALSE (7 is not greater than 16). Since only one value of an OR expression needs to be TRUE for the entire expression to be TRUE, R returns a result of TRUE.

**[1] TRUE**

Now, suppose y is 12. The expression y < 8 | y > 16 now evaluates to FALSE (12 < 8) | FALSE (12 > 16). Both comparisons are FALSE, so the result is **FALSE**.

**y <- 12**

**y < 8 | y > 16**

* **[1] FALSE**

### **NOT operator “!”**

* The NOT operator (!) simply negates the logical value it applies to. In other words, !TRUE evaluates to **FALSE**, and !FALSE evaluates to **TRUE**.

When you run the code, you get the following results:

**> !TRUE**

**[1] FALSE**

**> !FALSE**

**[1] TRUE**

Just like the OR and AND operators, you can use the NOT operator in combination with logical operators. Zero is considered FALSE and non-zero numbers are taken as TRUE. The NOT operator evaluates to the opposite logical value.

Let’s imagine you have a variable x that equals 2:

**x <- 2**

The NOT operation evaluates to FALSE because it takes the opposite logical value of a non-zero number (TRUE).

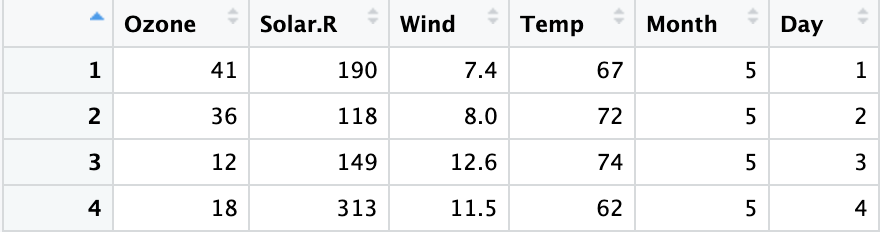
**> !x**

* **[1] FALSE**

-----------------

Let’s check out an example of how you might use logical operators to analyze data. Imagine you are working with the *airquality* dataset that is preloaded in RStudio. It contains data on daily air quality measurements in New York from May to September of 1973.

The data frame has six columns: *Ozone* (the ozone measurement), *Solar.R* (the solar measurement), *Wind* (the wind measurement), *Temp* (the temperature in Fahrenheit), and the *Month* and *Day* of these measurements (each row represents a specific month and day combination).



### **AND example**

Imagine you want to specify rows that are extremely sunny and windy, which you define as having a *Solar* measurement of over 150 anda *Wind* measurement of over 10.

In R, you can express this logical statement as **Solar.R > 150 & Wind > 10**.

Only the rows where *both* of these conditions are true fulfill the criteria:

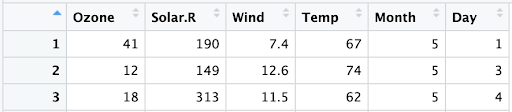
Image of a single row of the “airquality” dataset in the RStudio data viewer.

## **OR example**

Next, imagine you want to specify rows where it’s extremely sunny or it’s extremely windy, which you define as having a *Solar* measurement of over 150 or a *Wind* measurement of over 10.

In R, you can express this logical statement as **Solar.R > 150 | Wind > 10**.

All the rows where *either* of these conditions are true fulfill the criteria:

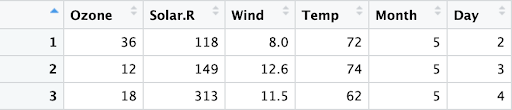


## **NOT example**

Now, imagine you just want to focus on the weather measurements for days that aren't the first day of the month.

In R, you can express this logical statement as **Day != 1**.

The rows where this condition is true fulfill the criteria:



Finally, imagine you want to focus on scenarios that aren't extremely sunny and not extremely windy, based on your previous definitions of extremely sunny and extremely windy. In other words, the following statement should not be true: either a *Solar* measurement greater than 150 or a *Wind* measurement greater than 10.

Notice that this statement is the opposite of the OR statement used above. To express this statement in R, you can put an exclamation point (!) in front of the previous OR statement: **!(Solar.R > 150 | Wind > 10)**. R will apply the NOT operator to everything within the parentheses.

In this case, only one row fulfills the criteria:



----------------------------------------------------------------------------------------------------------------------------------------

## **Optional: Conditional statements**

A **conditional statement** is a declaration that if a certain condition holds, then a certain event must take place. For example, “*If* the temperature is above freezing, *then* I will go outside for a walk.” If the first condition is true (the temperature is above freezing), then the second condition will occur (I will go for a walk). Conditional statements in R code have a similar logic.

Let’s discuss how to create conditional statements in R using three related statements:

* **if() / else() / else if()**

### **if statement**

The **if** statement sets a condition, and if the condition evaluates to **TRUE**, the R code associated with the if statement is executed.

In R, you place the code for the condition inside the parentheses of the if statement. The code that has to be executed if the condition is TRUE follows in curly braces (**expr**). Note that in this case, the second curly brace is placed on its own line of code and identifies the end of the code that you want to execute.

**if (condition) {**

**expr**

**}**

For example, let’s create a variable *x* equal to 4.

**x <- 4**

Next, let’s create a conditional statement: if *x* is greater than 0, then R will print out the string **“x is a positive number".**

**if (x > 0) {**

**print("x is a positive number")**

**}**

Since x = 4, the condition is true (4 > 0). Therefore, when you run the code, R prints out the string **“x is a positive number"**.

**[1] "x is a positive number"**

But if you change x to a negative number, like -4, then the condition will be FALSE (-4 > 0). If you run the code, R will not execute the print statement. Instead, a blank line will appear as the result.

### **else statement**

The **else** statement is used in combination with an if statement. This is how the code is structured in R:

**if (condition) {**

**expr1**

**} else {**

**expr2**

**}**

The code associated with the else statement gets executed whenever the condition of the if statement is *not* TRUE. In other words, if the condition is TRUE, then R will execute the code in the if statement (*expr1*); if the condition is *not* TRUE, then R will execute the code in the else statement (*expr2*).

Let’s try an example. First, create a variable *x* equal to 7.

**x <- 7**

Next, let’s set up the following conditions:

* If x is greater than 0, R will print **“x is a positive number”**.
* If x is less than or equal to 0, R will print **“x is either a negative number or zero”**.

In our code, the first condition (x > 0) will be part of the if statement. The second condition of x less than or equal to 0 is implied in the else statement. If x > 0, then R will print **“x is a positive number”**. Otherwise, R will print **“x is either a negative number or zero”**.

**x <- 7**

**if (x > 0) {**

**print ("x is a positive number")**

**} else {**

**print ("x is either a negative number or zero")**

**}**

Since 7 is greater than 0, the condition of the if statement is true. So, when you run the code, R prints out **“x is a positive number”**.

**[1] "x is a positive number"**

But if you make x equal to -7, the condition of the if statement is *not* true (-7 is not greater than 0). Therefore, R will execute the code in the else statement. When you run the code, R prints out **“x is either a negative number or zero”**.

**x <- -7**

**if (x > 0) {**

**print("x is a positive number")**

**} else {**

**print ("x is either a negative number or zero")**

**}**

**[1] "x is either a negative number or zero"**

### **else if statement**

In some cases, you might want to customize your conditional statement even further by adding the **else if** statement. The else if statement comes in between the if statement and the else statement. This is the code structure:

**if (condition1) {**

**expr1**

**} else if (condition2) {**

**expr2**

**} else {**

**expr3**

**}**

If the if condition (*condition1*) is met, then R executes the code in the first expression (*expr1*). If the if condition is not met, and the else if condition (*condition2*) is met, then R executes the code in the second expression (*expr2*). If neither of the two conditions are met, R executes the code in the third expression (*expr3*).

In our previous example, using only the if and else statements, R can only print **“x is either a negative number or zero”** if x equals 0 or x is less than zero. Imagine you want R to print the string **“x is zero”** if x equals 0. You need to add another condition using the else if statement.

Let’s try an example. First, create a variable *x* equal to negative 1 (“-1”), and *run the code* to save the variable to memory.

**x <- -1**

Now, you want to set up the following conditions:

* If x is less than 0, print **“x is a negative number”**
* If x equals 0, print **“x is zero”**
* Otherwise, print **“x is a positive number”**

In the code, the first condition will be part of the if statement, the second condition will be part of the else if statement, and the third condition will be part of the else statement. If x < 0, then R will print **“x is a negative number”.** If x = 0, then R will print **“x is zero”**. Otherwise, R will print **“x is a positive number”**.

**x <- -1**

**# run the code**

**if (x < 0) {**

**print("x is a negative number")**

**} else if (x == 0) {**

**print("x is zero")**

**} else {**

**print("x is a positive number")**

**}**

Run the code. Since -1 is less than 0, the condition for the if statement evaluates to **TRUE**, and R prints **“x is a negative number”**.

**[1] "x is a negative number"**

If you make x equal to 0, R will first check the if condition **(x < 0)**, and determine that it is FALSE. Then, R will evaluate the else if condition. This condition, **x==0**, is TRUE. So, in this case, R prints **“x is zero”**.

If you make x equal to 1, both the if condition and the else if condition evaluate to **FALSE**. So, R will execute the else statement and print **“x is a positive number”**.

As soon as R discovers a condition that evaluates to TRUE, R executes the corresponding code and ignores the rest.

## **Additional resource**

To learn more about logical operators and conditional statements, check out DataCamp's tutorial [Conditionals and Control Flow in R](https://www.datacamp.com/community/tutorials/conditionals-and-control-flow-in-r). DataCamp is a popular resource for people learning about computer programming. The tutorial is filled with useful examples of coding applications for logical operators and conditional statements (and relational operators), and offers a helpful overview of each topic and the connections between them.

[Guide: Keeping your code readable](https://www.coursera.org/learn/data-analysis-r/supplement/fQFvb/guide-keeping-your-code-readable)

[Hands-On Activity: R sandbox](https://www.coursera.org/learn/data-analysis-r/quiz/lJaOP/hands-on-activity-r-sandbox)

Question 1



## **Activity overview**

****

So far, you’ve learned about the R programming language and why data analysts use it. In this activity, you will preview some of the cool things you can do in R. You will also learn more about working with packages and data and try out some important functions.

By the end of this activity, you will know how to install and load R packages, practice using functions to view, clean, and visualize data, and learn more about using R markdown to document your analysis. This will enable you to use R markdown, which helps to facilitate collaboration and document analysis which is needed for more complex projects.

## **Work in RStudio Cloud**

****

To start, log into your RStudio (Posit) Cloud account. Open the project you will work on in the activity with [this link](https://posit.cloud/content/6208304). At the top right portion of the screen you will see a "red stamp" indicating this project as a Temporary Copy. Click on the adjacent button, Save a Permanent Copy, and the project will be saved in your main dashboard for use with future lessons. Once that is completed, navigate to the file explorer in the bottom right and click the following: Course 7 -> Week 2 -> Lesson3\_Sandbox.Rmd.

If you’re having trouble finding the correct activity, check out this [step-by-step guide](https://scribehow.com/shared/Access_and_Install_Course_Material_for_Lesson_3__JGhlL8PLSxuqtK2KRWZkJw) on how to navigate in RStudio (Posit) Cloud. Make sure to select the correct R markdown (Rmd) file. The other Rmd files will be used in different activities.

If you are using RStudio Desktop, you can download the Rmd file directly here:

[Lesson3\_Sandbox](https://d3c33hcgiwev3.cloudfront.net/k7wCfkohTdG8An5KIU3R_g_c39319760e5344c1916586dbc1594af1_Lesson3_Sandbox.Rmd?Expires=1718755200&Signature=Jv~ppKtlIbjwZtBZ~SMheY53MDaiojtPeywjEZqW4obUuGQIcWq53dMqMCshSmqrYfvgpE19ri38dAjndSr7wsvZh4~eH5SYK4EZgX2sOLTdmqN5T44KkeAu3HecXD1QTO3H2Io4VuzkDxUrQMInQXMs3lDKANfJFuIcd0fSwAY_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A)

[RMD File](https://d3c33hcgiwev3.cloudfront.net/k7wCfkohTdG8An5KIU3R_g_c39319760e5344c1916586dbc1594af1_Lesson3_Sandbox.Rmd?Expires=1718755200&Signature=Jv~ppKtlIbjwZtBZ~SMheY53MDaiojtPeywjEZqW4obUuGQIcWq53dMqMCshSmqrYfvgpE19ri38dAjndSr7wsvZh4~eH5SYK4EZgX2sOLTdmqN5T44KkeAu3HecXD1QTO3H2Io4VuzkDxUrQMInQXMs3lDKANfJFuIcd0fSwAY_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A)

Carefully read the instructions in the comments of the Rmd file and complete each step. Some steps may be as simple as running pre-written code, while others may require you to write your own functions. After you finish the steps in the Rmd file, return here to confirm that your work is complete.

[Queries and programming](https://www.coursera.org/learn/data-analysis-r/discussionPrompt/LociE/queries-and-programming)

[Basic Concepts of R](https://www.coursera.org/learn/data-analysis-r/ungradedWidget/rr7Yj/basic-concepts-of-r)

[Test your knowledge on coding in R](https://www.coursera.org/learn/data-analysis-r/quiz/O9zRl/test-your-knowledge-on-coding-in-r)

**Package**: In R, a package is a collection of R functions, data, and compiled code in a well-defined format. Packages are used for organizing code into reusable units. They can contain functions (for specific tasks or analyses), datasets (for sample or real data), documentation (to explain usage and functionality), and tests (to verify correctness).

**Data type**: A data type in programming refers to the classification of data items. While important in programming, it does not encompass the entire scope of functions, documentation, datasets, and tests required for sharing and reusing code across projects.

**Tidyverse**: Tidyverse is a collection of R packages designed for data science, providing tools for data manipulation, visualization, and analysis. While tidyverse packages can be part of a package you create, tidyverse itself does not directly encompass the creation of functions, documentation, datasets, and tests in a reusable manner.

**Data frame**: A data frame is a fundamental data structure in R for storing tabular data. It alone does not facilitate the creation and sharing of functions, documentation, datasets, and tests across projects.

ChatGPT

To make nested functions more understandable for stakeholders in R, the analyst can use **pipes**.

Here’s why pipes (using %>% operator from the magrittr or dplyr package) can accomplish this:

**Pipe (%>%)**: Pipes allow you to chain operations together, passing the output of one function as the input to the next function. This makes the code more readable by emphasizing the sequence of operations rather than nested function calls. For example:  
r  
Copy code  
data %>%

filter(condition) %>%

group\_by(variable) %>%

summarize(mean\_value = mean(value))

* In this example, data is piped through a series of operations (filter(), group\_by(), summarize()), each of which operates on the result of the previous operation.
* **Vector**: Vectors in R are used to store multiple elements of the same type. They do not directly relate to improving the readability of nested functions.
* **Comment**: Comments in R (using #) are used to annotate code and explain its functionality, but they do not change the structure or readability of nested functions.
* **Argument**: Arguments in functions specify values to be passed into the function when it is called. They are not directly related to restructuring code for readability.

LEARN ABOUT R PACKAGES

[The gift that keeps on giving](https://www.coursera.org/learn/data-analysis-r/lecture/2xqTb/the-gift-that-keeps-on-giving)

I have to say getting a package delivered to you is one of life's simple pleasures. It doesn't matter if it's a surprise package or something you ordered yourself. It's exciting to open your package to discover what's inside. No wonder those unboxing videos on Youtube are so popular. Well, R has a different kind of package that our R users can open.

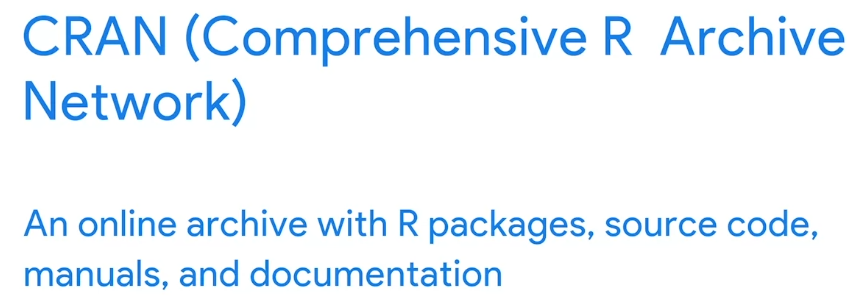
These packages are units of reproducible R code and they make it easier to keep track of code. They're created by members of the R community to keep track of the R functions that they write and reuse. These community members might then make the packages available to other users. It's one of the great things about being part of this community.

Packages in R include reusable R functions and documentation about the functions including how to use them. They also contain sample datasets and tests for checking your code to make sure it does what you want it to do. By default, R includes a set of packages called base R that are available to use in RStudio when you start your first programming session. There's also recommended packages that are loaded but not installed. Before using functions from one of these packages, you'd have to load it with a library command like library boot, for example. Let's find out which packages we already have in RStudio. We'll work in our console instead of a script for now because we're practicing and don't need to save this code for later.

To check out our packages, we'll just run the command **installed.packages** and there's our list. Let's focus on the package and priority columns. The package column gives the name of the package, like cluster or graphics. The priority column tells us what's needed to use functions from the package. If you come across the word base in the priority column, then the package is already installed and loaded. You can use all of the functions of that package as soon as you open RStudio. If you find the word recommended, then the package is installed but not loaded. You'll also notice a list of packages in the bottom right part of our workspace. This list includes a brief description of each package.

To load class and other uninstalled packages, we'll need to use the **library function followed by the name of the package**. Now, the class package has a check next to it, so it's been successfully loaded for use. If you want to learn even more about your loaded packages, you can click on their names in the Packages tab. This opens the Help tab and shows topics related to the package you selected. You can also use the Help function in your programming to call up the Help tab. While the pre-installed packages give you tons of useful functions, there's even more packages that will further expand your programming abilities. You can find thousands of R packages just by doing an online search.

One of the most commonly used sources of packages is CRAN. CRAN stands for comprehensive R archive network.



It's an online archive with R packages, source code, manuals, and documentation. When you start working with R, you'll be able to do your own searches to find packages in CRAN or elsewhere.

It's almost always easier to just search with your favorite search engine though. **So packages are a pretty big part of using R.** They give you most of what you need to complete your programming throughout the data analysis process. Who knows? You might even turn your own code into packages for others to use. Up next, we'll keep unpacking R packages. See you soon.

[Available R packages](https://www.coursera.org/learn/data-analysis-r/supplement/PvhrW/available-r-packages)

To make the most of R for your data analysis, you will need to install packages. **Packages** are units of reproducible R code that you can use to add more functionality to R. The best part is that the R community creates and shares packages so that other users can access them! In this reading, you will learn more about widely used packages and where to find them.



Packages can be found in repositories, which are collections of useful packages that are ready to install. You can find repositories on [**Bioconductor**](http://bioconductor.org/), [**R-Forge**](https://r-forge.r-project.org/), [**rOpenSci**](https://ropensci.org/), or [**GitHub**](https://github.com/), but the most commonly used repository is the Comprehensive R Archive Network or [**CRAN**](https://cran.r-project.org/). CRAN stores code and documentation so that you can install packages into your own RStudio space.

## **Package documentation**

Packages will not only include the code itself, but also documentation that explains the package’s author, function, and any other packages that you will need to download. When you are using CRAN, you can find the package documentation in the DESCRIPTION file.

Check out Karl Broman's [**R Package Primer**](https://kbroman.org/pkg_primer/)to learn more.

## **Choosing the right packages**

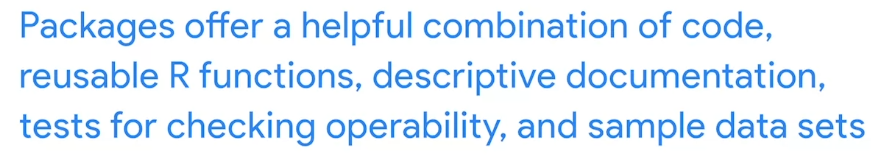
With so many packages out there, it can be hard to know which ones will be the most useful for your library or directory of installed packages. Luckily, there are some great resources out there:

* [**Tidyverse**](https://www.tidyverse.org/): the tidyverse is a collection of R packages specifically designed for working with data. It’s a standard library for most data analysts, but you can also download the packages individually.
* [**Quick list of useful R packages**](https://support.rstudio.com/hc/en-us/articles/201057987-Quick-list-of-useful-R-packages): this is RStudio Support’s list of useful packages with installation instructions and functionality descriptions.
* [**CRAN Task Views**](https://cran.r-project.org/web/views/): this is an index of CRAN packages sorted by task. You can search for the type of task you need to perform and it will pull up a page with packages related to that task for you to explore.

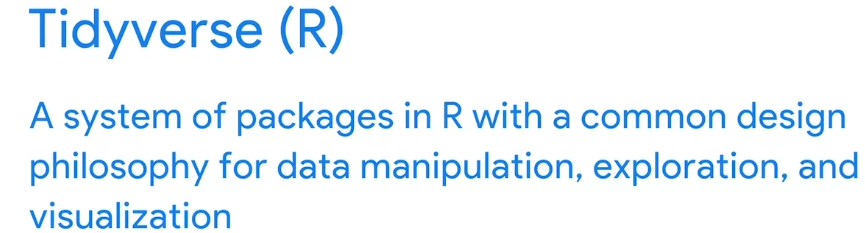
You will discover more packages throughout this course and as you use R more often, but this is a great starting point for building your own library.

[Welcome to the tidyverse](https://www.coursera.org/learn/data-analysis-r/lecture/MbsrZ/welcome-to-the-tidyverse)

As we discussed earlier, packages are a big part of what makes R so great.



And for lots of data analysts, at the top of the list of useful packages is tidyverse.



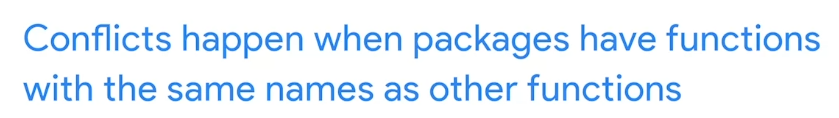
Tidyverse is actually a collection of packages in R with a common design philosophy for data manipulation, exploration, and visualization. Using tidyverse can help you work your way through pretty much the entire data analysis process. The packages in tidyverse work together naturally. I started learning about tidyverse when I was working on a survey project. It felt like I was stepping into a more advanced zone of R. I understood the basics, but now I was finding out how the tidyverse improves on the basics. That's when I got even more excited about working in R. I realized that the more I put into learning about the tidyverse, the more I get out of it. On top of that, the community support for tidyverse is strong too. It's one of the reasons why tidyverse is considered a key part of programming for most R users. The principles associated with tidyverse, which you'll learn both here and at your job, have been widely adopted by the R community. You'll find lots of tutorials and examples related to the tidyverse online that show you these principles and how they're applied to data analytics.

Okay, let's install the tidyverse. You can follow along on your own, using your RStudio cloud account. Check out the reading for more details. Earlier, you learned how to find Base R packages using the function install packages. To install packages like the tidyverse that aren't in Base R, we'll use the install packages function. As we discussed earlier, this function calls the tidyverse and other packages from CRAN. Let's talk about why CRAN was created. Since packages not in Base R are mostly made by R users, people need a reliable way to check and validate submitted code. CRAN makes sure any R content open to the public meets the required quality standards. So, if it's sourced through CRAN, you can feel good that the package is authentic and valid. Another major source of packages and other R content is GitHub. Now, we'll get back to installing the tidyverse. We'll first type install.packages. Then, between the parentheses, we'll type tidyverse in quotes. The quotes aren't always necessary, but best practice is to use quotes to make sure that we are accurate. We'll press Enter and wait for RStudio to install tidyverse.

When we click on our packages tab, we come across a lot of new packages on the list. That's tidyverse. You might have noticed that none of the packages are checked off. We need to load them first before we can use them. But that's a mighty long list. So, let's just load the package named tidyverse for now, using the library function.

The return shows that not only was tidyverse loaded, but eight other packages were too. It also shows a list of conflicts.

Conflicts happen when packages have functions with the same names as other functions.



Basically, the last package loaded is the one whose functions will be used, so we'll stick with the tidyverse functions. But it's important to note that these messages only appear once.

So, as you get more used to R, you'll be able to figure out if you want to use certain functions over others. The loaded packages are ggplot2, tibble, tidyr, readr, purrr, dplyr, stringr, and forcats. These packages are the core of the tidyverse because you'll use them in almost every analysis. All of them work together to make your data analysis smooth and efficient. With these packages, tidyverse helps you do everything from importing and transforming data to exploring and visualizing it. We'll check out this core of packages soon, and we'll use them even more as we continue working in RStudio. If you're working on your own in R, you can check out some of the other packages too.

The packages available in tidyverse change a lot, but you can always check for updates by running **tidyverse\_update()** in your console. You can then update the packages in a couple of ways.

If you use the **update packages function**, it'll update all of your packages. That might take a while.



So, if you just want to update one package, you can use the install packages function again with the package name as your argument in parentheses. You should update packages regularly to make sure you've got the latest version in your code. Conflict notifications are just one type of message that can show up in the console. You might find warnings and error messages as well. A quick search using the help tab will usually tell you what the message means and what, if anything, you'll need to do to address it. Coming up, we'll keep moving through the tidyverse. You'll find out more about why tidyverse is such an integral part of R. See you.

[Hands-On Activity: Installing and loading tidyverse](https://www.coursera.org/learn/data-analysis-r/quiz/CWrfL/hands-on-activity-installing-and-loading-tidyverse)

Question 1



## **Activity overview**

****

In the last activity, you explored the R sandbox and used some R packages such as the tidyverse. In this activity, you’ll explore further with the tidyverse collection of packages and learn about them using the browseVignettes function.

By the end of this activity, you will know how to easily load vignettes. Moving forward, you can use the browseVignettes function to access and review included documentation to better understand each R package you will use.

## **Install the tidyverse**

****

If you have not yet installed the tidyverse, open RStudio.

Log in, navigate to the console, type install.packages("tidyverse"), and press Enter (Windows) or Return (Mac).

Then wait as RStudio installs the tidyverse packages (be patient, this can take a little bit). You’ll receive a message that the install is done.



\*installing \*binary\* package 'tidyverse'

\*DONE (tidyverse)

The downloaded source packages are in

'/tmp/RtmpUCbADH/downloaded\_packages'

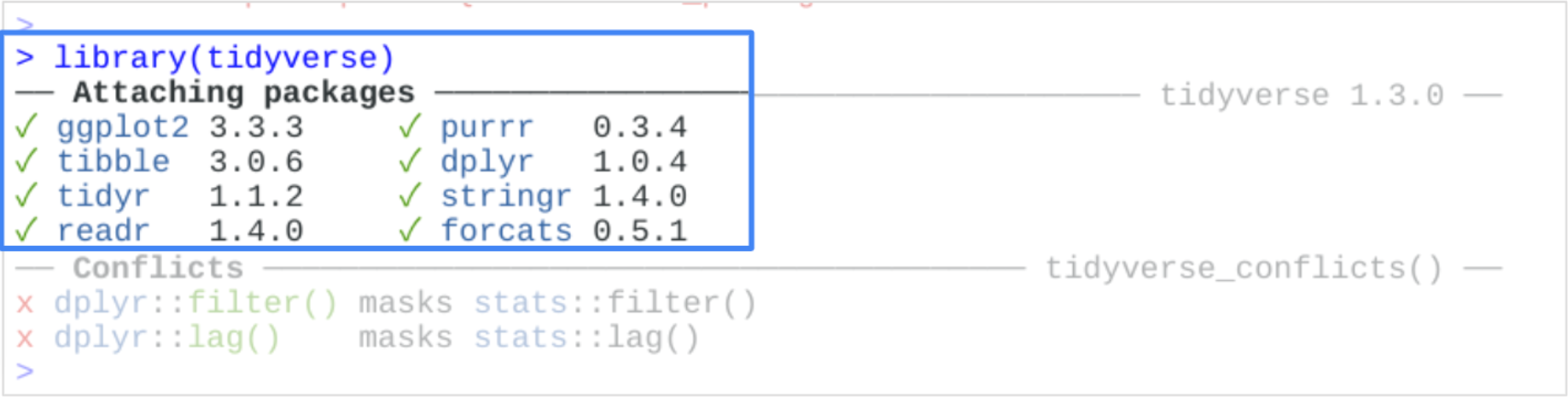
>

## **Load the tidyverse**

****

Once the tidyverse packages have been installed, load them so that they are available in your current R session. Load the core tidyverse with the library command. The core tidyverse contains the main packages that work together to make your data analysis smooth and efficient.

To load the core tidyverse, type library(tidyverse) and press Enter (Windows) or Return (Mac).



The output in the console indicates that you have loaded the core tidyverse. Each of the core packages has a green check next to it.

The output also lists conflicts. Conflicts report which objects have the same name in two or more places within your session. This usually happens because an object in your workspace or a package you installed is masking a system object of the same name.

Since you most recently loaded the tidyverse packages, they will be the default packages for your current session.

## **Read tidyverse vignettes**

****

A vignette is documentation that acts as a guide to an R package. A vignette shares details about the problem that the package is designed to solve and how the included functions can help you solve it. The browseVignettes function allows you to read through vignettes of a loaded package.

To check out vignettes for one specific package, type browseVignettes(“packagename”) and press Enter (Windows) or Return (Mac). Remember that functions are case-sensitive in R, so “Vignettes” must have a capital V.

For example if you execute the browseVignettes() function on ggplot2, browseVignettes(“ggplot2”), you will have the following outcome:



If you are using RStudio (Posit) Cloud, running this function on the Posit Cloud server may lead to a page where the linked contents do not exist. If this is the case, downloading the RStudio Desktop version and running the same browseVignette() functions as above will open a new browser tab with the *HTML, source, and R code* links leading to the vignettes, and these vignette description pages will be functional.

[Test your knowledge on R packages](https://www.coursera.org/learn/data-analysis-r/quiz/kjAEM/test-your-knowledge-on-r-packages)

EXPLORE THE TIDYVERSE

[More on the tidyverse](https://www.coursera.org/learn/data-analysis-r/lecture/oiFFN/more-on-the-tidyverse)

Have you ever taken a tour of a famous landmark or an unfamiliar city? It can be pretty exciting. You get to learn all about the features of the landmark or city. Eventually, you get to know them pretty well, and you can share what you learned with others. Well we're here to take a different kind of tour: a tour of the tidyverse.

For this tour, we won't be traveling anywhere special, but we will help you learn about the exciting tidyverse features. And once you know them a little better, you can most definitely share what you learned with others. For this tour we'll focus on the core packages of tidyverse we discussed earlier: ggplot2, tidyr, readr, dplyr, tibble, purrr, stringr and forcats. We also learned how to install and load them in RStudio.

Once they're loaded, you won't need to do anything else with their actual packages. They'll do their thing as you program.

So what is their thing? Well, it depends, but there's four packages that are an essential part of the workflow for data analysts: ggplot2, dplyr, tidyr and readr. **You'll most likely use these more often than the others**.

Ggplot2 is used for data visualization, specifically plots. With ggplot2, you can create a variety of data viz by applying different visual properties to the data variables. Here's an example of ggplot2 in action. You'll have your own chance to use ggplot2 later.

Tidyr is a package used for data cleaning to make tidy data. We covered tidy or clean data earlier, but as a quick reminder, it's data where every part of a data table or data frame is the right type in the right place. Tidyr works with wide and long data to make sure this happens.

Next, we have readr, which is used for importing data. The most common function from readr is read\_csv. This will import a CSV file into R. A CSV file contains data separated by commas in a table format. To accurately read a dataset with readr, you combine the function with a column specification. The column specification describes how each column should be converted to the most appropriate data type. It's good to keep in mind this isn't usually necessary because readr will figure it out for you automatically. We'll come across readr functions as we continue to explore R.

Now on to dplyr. Dplyr offers a consistent set of functions that help you complete some common data manipulation tasks. For example, the select function picks variables based on their names, and the filter function finds cases where certain conditions are true. And, yes, dplyr is another package we'll get to later. There's plenty to look forward to, so that's the fab four of the tidyverse. They'll all make your programming in R more straightforward and efficient. The other four packages are definitely useful, too, but you might not use them as often. Tibble works with data frames. Purrr works with functions and vectors helping make your code easier to write and more expressive. Stringr includes functions that make it easier to work with strings.

Forcats provide tools that solve common problems with factors. As a quick reminder, factors store categorical data in R where the data values are limited and usually based on a finite group like country or year. Using the tidyverse and its packages will help you fine-tune your analysis. And besides tidyverse, you also learned the fundamentals of R from variables to vectors and more.

You explored the different operators in R and saw how they can help you complete calculations. You had the chance to check out pipes and how they can make your programming more efficient. And you unpacked packages to find out how they're a big part of what you can do in R.

We've covered a lot of ground in just a few videos, so this might be a good time for you to do a little review. You can rewatch videos and revisit any other resources that can help you get an even better grasp of all the terms, concepts and processes that are part of R. Looking ahead, you'll start working with data in R including a more thorough exploration of how tidyverse impacts your process. You'll see tibble, readr and other tidyverse packages in action. And you'll find out how to clean and organize your data in R.

[Use pipes to nest code](https://www.coursera.org/learn/data-analysis-r/lecture/5AFvs/use-pipes-to-nest-code)

Earlier we introduced something called pipes. A pipe is a tool in R that helps make your code more efficient and easier to read and understand. In this video, we'll explore pipes in more detail. As a quick reminder, a pipe is a tool in R for expressing a sequence of multiple operations. In other words, it takes the output of one statement and makes it the input of the next statement. Instead of typing out functions contained inside other functions, you could use the pipe operator to do the same work. In programming, we describe this as nested.

**Nested describes code that performs a particular function and is contained within code that performs a broader function**.

You can think of a pipe as a way to code the phrase and then. Say you've got sales data and you need to find the mean or average. You can create a pipe by calling up the data and then grouping the data and then summarizing the group data using a mean function. Let's check out an example. First, we'll open RStudio. Then we'll start a new script so we can save our work. We'll save it as ToothGrowth exploration.

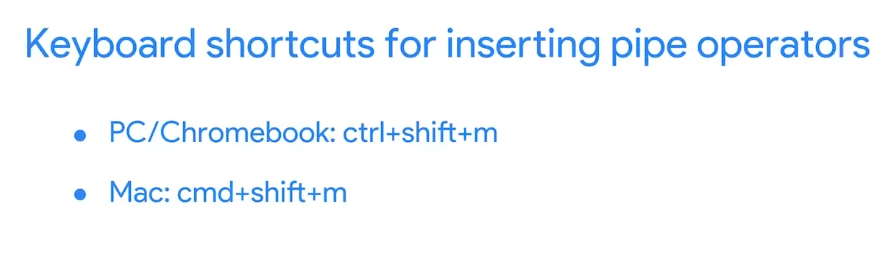
We'll use the ToothGrowth dataset, which is already installed in R. This dataset contains data about the effect of vitamin C on the growth of teeth in guinea pigs. It's a well-known dataset that'll help us learn about how pipes work. To load any dataset already installed, we use the data function. We then add the name of the dataset, ToothGrowth. Now that the data is loaded, we can check it out with the View function. Notice how View begins with a capital V. It's a good reminder that functions and variables are case- sensitive in R. In a script we use the Run button to run our code. The return usually shows up in the console. But with View, a new tab appears in the script showing the contents of the dataset. Now, let's say we need to filter and sort this data to organize it for analysis. Without pipes, we could do this either by nesting commands or by creating a sequence of data frames. We'll talk more about data frames soon. Let's start by filtering the dataset. Note that we want to first install and load the correct filter function, which comes as part of a package. Installing a package may take a few moments. This function comes as part of the dplyr package. We'll assign a name to the new dataset and then apply the filter function.

This filters the data so that we only see rows where the dose of vitamin C is exactly 0.5. This includes both types of vitamin C used in the study. Orange juice or OJ in our dataset, and ascorbic acid or VC. Next, we'll sort it with the arrange function. We'll include the name of the filter dataset followed by the column name we want to sort by. In this case len, which stands for length of tooth. When we run this, the return appears in the console.

The data is arranged in ascending order by len. The return only shows rows where the dose amount is 0.5.

The data has been filtered and sorted based on our code. Let's try another way to get the same return. We'll use a nested function, which is a function that is completely contained within another function. Here's the nested function for filtering and sorting this dataset.

Notice that the filter function from our previous code is the nested function. With nested functions, we read from the inside out. The code filters the data first. Then it arranges or sorts it. Now, let's run this. We tweaked the code, but we get the same result. Now, we'll use a pipe. As a quick reminder, the operator used to call out a pipe is a percentage sign followed by a greater than sign and another percentage sign.



You can also use keyboard shortcuts to insert pipe operators. Control shift M for PCs and Chromebooks, and command shift M for Macs. We'll start this pipe by assigning it to a variable.

Then we'll type the name of the dataset we're pulling data from, ToothGrowth. We'll use our keyboard shortcut to add the pipe operator after that. Now we can press enter to go to the next line. RStudio automatically indents the next line, recognizing that it's part of the pipe. Next, we'll filter the data.

We don't have to call out the dataset inside parentheses, like we did in earlier examples, because we started our pipe with it. The pipe automatically applies the dataset to each step. Alright! Let's finish up our pipe on a new line with the arrange function and sort the data.

Since this is our last line of code, we don't need a pipe operator. Finally, click "Run" and presto, we get the same return as our other methods. Our pipe is setup to call the dataset and then filter the dataset and then sort the dataset. All three methods work, but you can see how pipes help make your programming more efficient and less cluttered. This means fewer chances for mistakes and better readability for anyone looking at your code, and because of the structure of a pipe, we can easily add to or change the code without having to start over. Let's do that. Building on our example, let's say we also wanted to compute the average tooth length or len for each of the two supplements used in the study: orange juice or OJ and ascorbic acid or VC. We'll replace the arrange function with the group by function. This'll group our results by the two supplements. We type supp in the parentheses and add a pipe. We're adding a pipe this time because we have another line of code to add. We group by and then we summarize. Our argument, which comes after the function summarize, looks pretty complex, but it basically tells R what to do with missing values and to make sure the data is grouped the right way when we add the summarize function. Now, we'll run our new pipe and get the average length of tooth when the dose is equal to 0.5 for each of our supplements.

Nice. Now, there's a couple of things to remember when using pipes.



Remember, RStudio automatically indents lines of code that are part of a pipe. If a line in your code isn't indented, it probably hasn't been added to the pipe. That could lead to an error statement. Another reason to use pipes when you can. Pipes or piping, and the functions that are part of the piping process, are building blocks for putting together analyzes in R. In upcoming videos, you'll learn how you can use these building blocks to clean, transform, and analyze your data. For now, feel free to take your time reviewing and maybe even practicing with the functions, operations and other elements in R and RStudio that we've already covered.

[R resources for more help](https://www.coursera.org/learn/data-analysis-r/supplement/aFF6V/r-resources-for-more-help)

The R community is full of dedicated users helping each other find solutions to problems and new ways of using R. There are also a lot of great blogs where you can find tutorials and other resources. Here are a few of them:

**Note:** due to the corporate change from R Studio to Posit, references in the following resources may have changed.

* [**Posit (RStudio)**](https://posit.co/): The best place to find help with R is in R itself! You can input ‘?’ or the help() command to search in R. You can also open the Help pane to find more R resources.
* [**Posit Blog:**](https://posit.co/blog/) Posit's blog is a great place to find information about RStudio, including company news. You can read the most recent [**featured posts**](https://blog.rstudio.com/categories/featured/) or use the search bar and the list of categories on the left side of the page to explore specific topics you might find interesting or to search for a specific post.
* [**Stack Overflow:**](https://stackoverflow.blog/)The Stack Overflow blog posts opinions and advice from other coders. This is a great place to stay in touch with conversations happening in the community.
* [**R-Bloggers:**](https://www.r-bloggers.com/)The R-Bloggers blog has useful tutorials and news articles posted by other R users in the community.
* [**R-Bloggers' tutorials for learning R:**](https://www.r-bloggers.com/2015/12/how-to-learn-r-2/#h.y5b98o9o2h1r)This blog post from R-Bloggers compiles some basic R tutorials and also links to more advanced guides.

[Connor: Coding tips](https://www.coursera.org/learn/data-analysis-r/lecture/DnANW/connor-coding-tips)

I'm a Marketing Analytics Manager at Google Cloud. I was running into barriers of not being able to do certain analysis because it was too time-consuming with my limited technical knowledge. So I started to teach myself things like SQL to help me access data through the current company's database that I had so that I could manipulate that data to better understand it.

I can tell you at first it is an incredibly frustrating thing to move through because it takes a lot of time and effort to do something that seems very simple or something that would be very easy to do in spreadsheets, but may be very difficult to do at first as you're learning how to code.

But also one of the most fulfilling things that I've ever done because once you're able to understand something, it opens up an entire new realm.

Learning coding was revolutionary for my job. I remember when I first started as an analyst, all of the data that I used was in spreadsheets and I had to run analysis and create formulas to manipulate the data, understand the data, and even analyze the data. Now, when we started to get more and more data, the formulas that I would have run would take hours, and I remember at one point I spent a few hours creating a formula and then executed it, and it took over ten hours to run. So I left my computer open and let it run throughout the night, woke up and it was still running.

Fast forward, a year later, after I'd learned SQL and Python, I was able to run the same type of analysis in milliseconds. So really understanding what it is that you're trying to do. Coding helps you manipulate and analyze data at a rate that previously or without coding knowledge would be very difficult to do.

**An important aspect of any type of script, or when you are coding is to structure it for overall readability.** More often than not, you're going to be working on a team. Now it's important that when you're writing a script, that you understand how it works, but also that somebody else who you work with can also come and understand what it is that you are trying to do within that script.

Now, it's very important that it not only works and is efficient, but it is also not too verbose, meaning that it is not overly complex. So an important aspect of readability is **if you are looking through your code and you realize that I've written the same thing multiple times, or I'm using the same logic or algorithm multiple times, that is a point in time where you can really consolidate your code and make it a lot more concise, which vastly helps with readability, vastly helps anybody who comes and is trying to read your code, and that includes you two weeks from now.** Because I can promise you when you start coding, you will realize that what makes sense to you right now may not make sense to you three weeks from now.

An important aspect for readability and overall understanding of your code is using comments. Comments are a way to write something out in a standardized language like English, and a way that somebody can understand it, but the computer doesn't pick up as actual code. So explaining every line that you write or explaining an entire section of your code in a comment allows somebody to walk themselves through your code and read exactly what it is that you are trying to accomplish with the code that you've written. Now without comments, you are leaving it up to the person to really follow your code and understand it themselves, which may not be an easy task for somebody because they may have a different way of coding, the same thing that you are doing.

Documenting your work is an important aspect. Documentation will explain in depth exactly what your code is doing, why it was built, what is the purpose for it, and any limitations.

The last one is a rather difficult concept to understand as you are first diving into learning a coding language, and that is **building it for scalability as well as making it dynamic.** Now when I say building something for scalability, what I mean is, if you are building a specific script of code to solve a task that you're running now, what you want to be sure of and answer is, will this code, or could this code be used in the future for something else? Now if it is, it's important that you make your code available to be scalable. That means that it is efficiently run so that if the size of the data that it is running any manipulations on increases, it doesn't bog down your code too much and that it can handle large data loads as well as small. Another aspect to that is making your code dynamic. What that means is not hard coding any values within your code that don't change when they need to. So these are just a few of the best practices and as you continue down your path as a data analyst, you'll pick up many, many more. There's always more to learn, there's always more to understand, but this should help you in beginning down your path to understanding coding.

[Test your knowledge on the tidyverse](https://www.coursera.org/learn/data-analysis-r/quiz/9hVei/test-your-knowledge-on-the-tidyverse)

M2 CHALLENGE

[Glossary: Terms and definitions](https://www.coursera.org/learn/data-analysis-r/supplement/8yMUQ/glossary-terms-and-definitions)