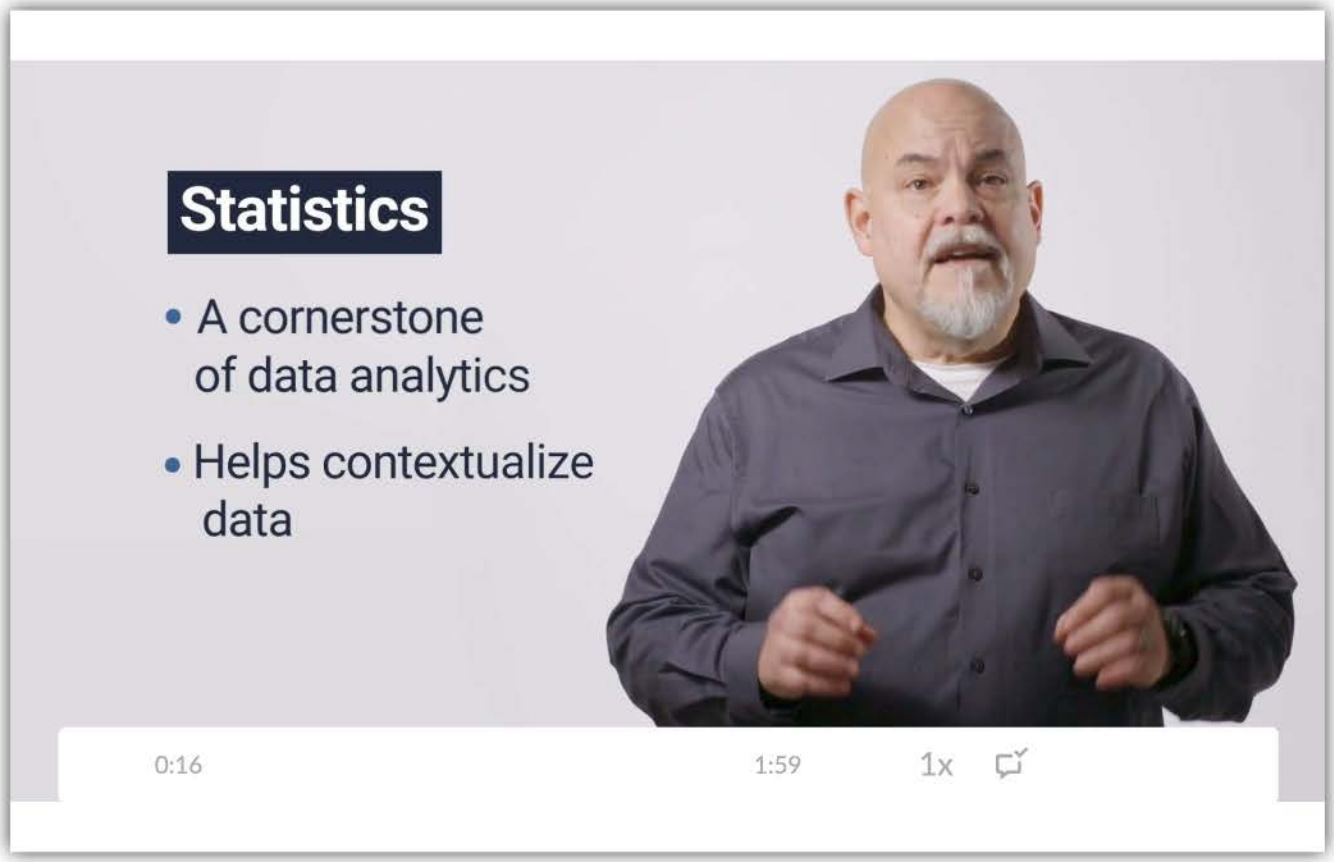
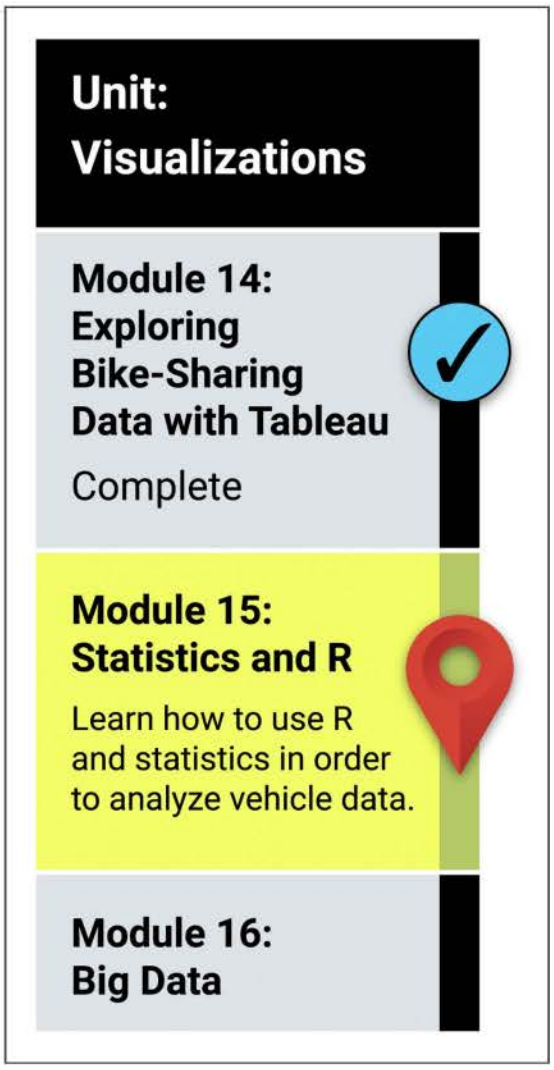
15.0.1: Using Statistics and R to Boost Your Data Science Repertoire



15.0.2: Module 15 Roadmap

Looking Ahead

In this module, you'll apply your understanding of statistics and hypothesis testing to analyze a series of datasets from the automotive industry. Your analysis will include visualizations, statistical tests, and interpretation of the results. All of your statistical analysis and visualizations will be written in the R programming language.



Throughout the module, you'll extract, transform, and load (ETL) data; visualize the data; and analyze the data using R. Additionally, you'll learn a variety of statistical tests, their real-world application in data science, and their implementation in R. The goal is for you to apply these statistical concepts beyond this module, to any dataset, using any programming language—including Python.

What You Will Learn

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

- Load, clean up, and reshape datasets using tidyverse in R.
- Visualize datasets with basic plots such as line, bar, and scatter plots using ggplot2.
- Generate and interpret more complex plots such as boxplots and heatmaps using ggplot2.
- Plot and identify distribution characteristics of a given dataset.
- Formulate null and alternative hypothesis tests for a given data problem.
- Implement and evaluate simple linear regression and multiple linear regression models for a given dataset.
- Implement and evaluate the one-sample t-Tests, two-sample t-Tests, and analysis of variance (ANOVA) models for a given dataset.
- Implement and evaluate a chi-squared test for a given dataset.
- Identify key characteristics of A/B and A/A testing.
- Determine the most appropriate statistical test for a given hypothesis and dataset.

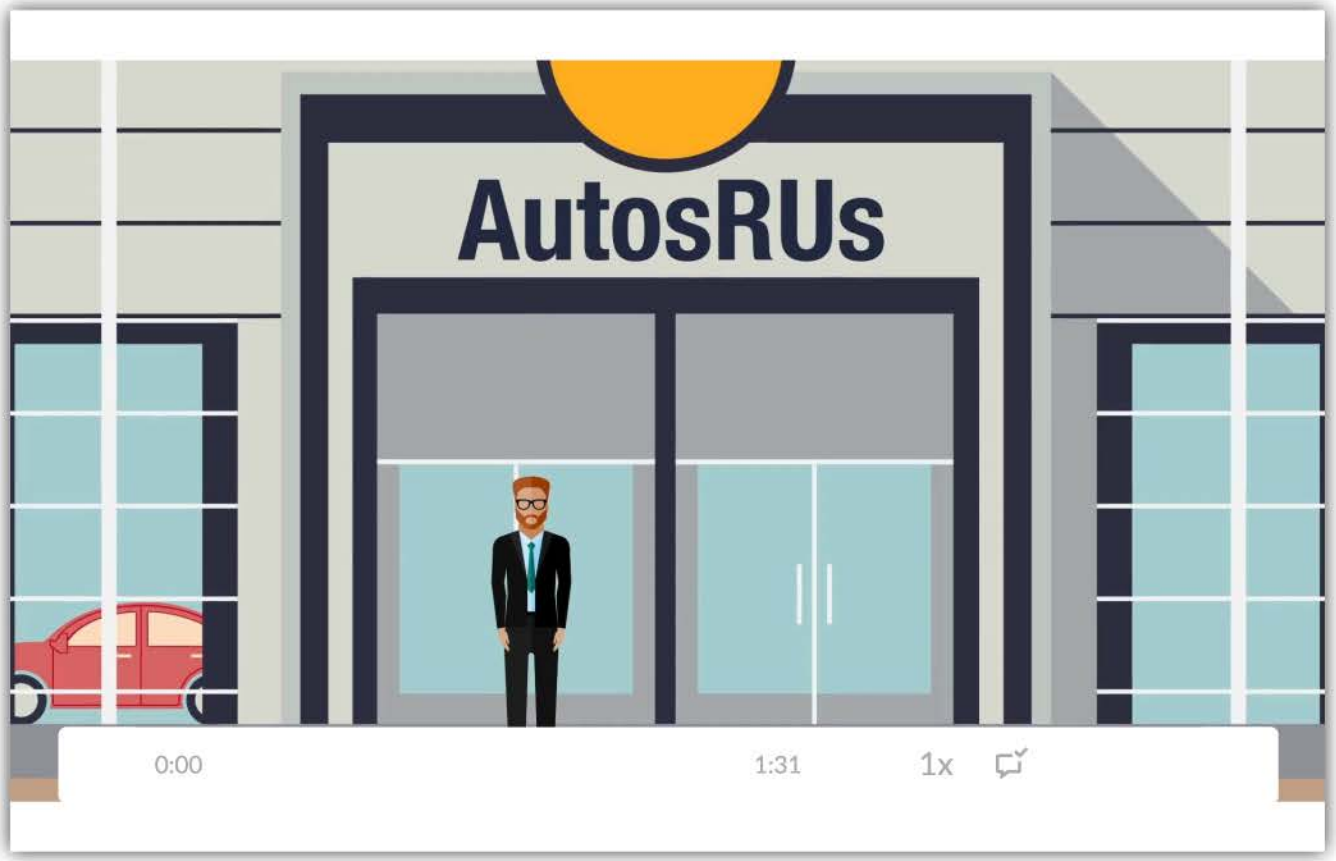
Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Introduction to Module 15 (15 minutes)
- Getting Started with R (30 minutes)
- Programming and ETL in R (2 hours)
- Visualize Your Data Using ggplot2 (2 hours)
- Introduction to Statistical Tests (1 hour)
- Introduction to Hypothesis Testing (30 minutes)
- Perform an Analysis of Means in R (1 hour)

- Correlation and Regression in R (1 hour)
- Characterize Categorical Data (30 minutes)
- Getting Real with A/B Testing (15 minutes)
- Choose the Right Test for Your Data (15 minutes)
- Application (5 hours)

15.0.3: Putting the “R” in AutosRUs



15.1.1: Introduction to R

Jeremy is ecstatic that he has been given the opportunity to lead the data analytics team. He's confident that his 10 years working with the manufacturing and research team has provided him with sufficient expertise in the subject matter. However, he is much less confident about his statistics background and his programming ability. He took some stats and programming courses in college, but it has been a long time since he had to think about either subject.

But Jeremy has never been one to walk away from a challenge. He knows that if he fully commits to learning stats and R programming, he'll feel comfortable in no time! With his previous experience in programming, learning R seems to be a good starting point to prepare for his new role.

R is a programming language that has a variety of uses in data science. R has solidified itself in academia and industry as one of the go-to programming languages for statistical modelling and hypothesis testing. In recent years, R developers have extended R's capabilities to generate machine learning algorithms and other advanced models to ensure that R can be used in every stage of data analytics.

Benefits of R

R is a versatile and extensible programming language with many benefits. One of the benefits of using an interpreted programming language such as R (or Python) is that the analysis scripts are written in plaintext. The versions of plaintext files are easy to control using tools such as Git, which means that **R analysis scripts** (or **RScripts**) are highly reproducible and easy to share with peers and collaborators.

Another benefit to using R is that the R programming language was specifically designed for data analysis. This means that the process of loading in a dataset, visualizing the data, and performing statistical tests is straightforward and easy to interpret. In fact, many of the statistical tests in Python have been directly ported from R due to how well they were implemented. In addition to the native statistical functions, there are many other useful data transformation and modelling libraries, such as the tidyverse package, that simplify the process of ETL and visualizations.

Drawbacks of R

Still, R is not perfect. The biggest drawback is its licensing. R and most of R's libraries are licensed as General Public License, version 2 (GPL 2). This means that if you program or model anything using R, GPL forces your application, program, or script to be open source.

In many personal and academic uses, this is not a problem because you're either (a) not trying to monetize your program or (b) going to publish your analysis and findings. However, if you are working for a company with intellectual property, or proprietary data and programs, this can be an issue. Therefore, many companies use R for internal analysis and regulatory testing, but use Python for any application or script that contains proprietary information.

Despite the licensing drawback, R is still a highly valuable programming language for data analysis and is used by data professionals at all levels across many

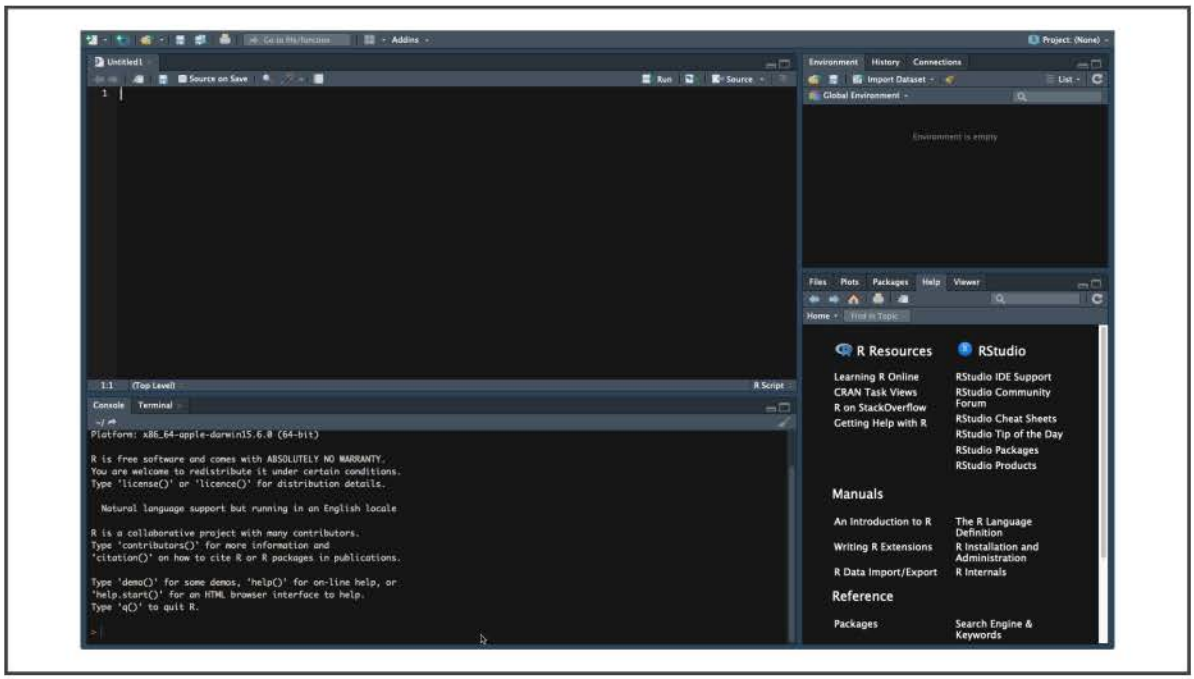
fields.

RStudio Integrated Development Environment

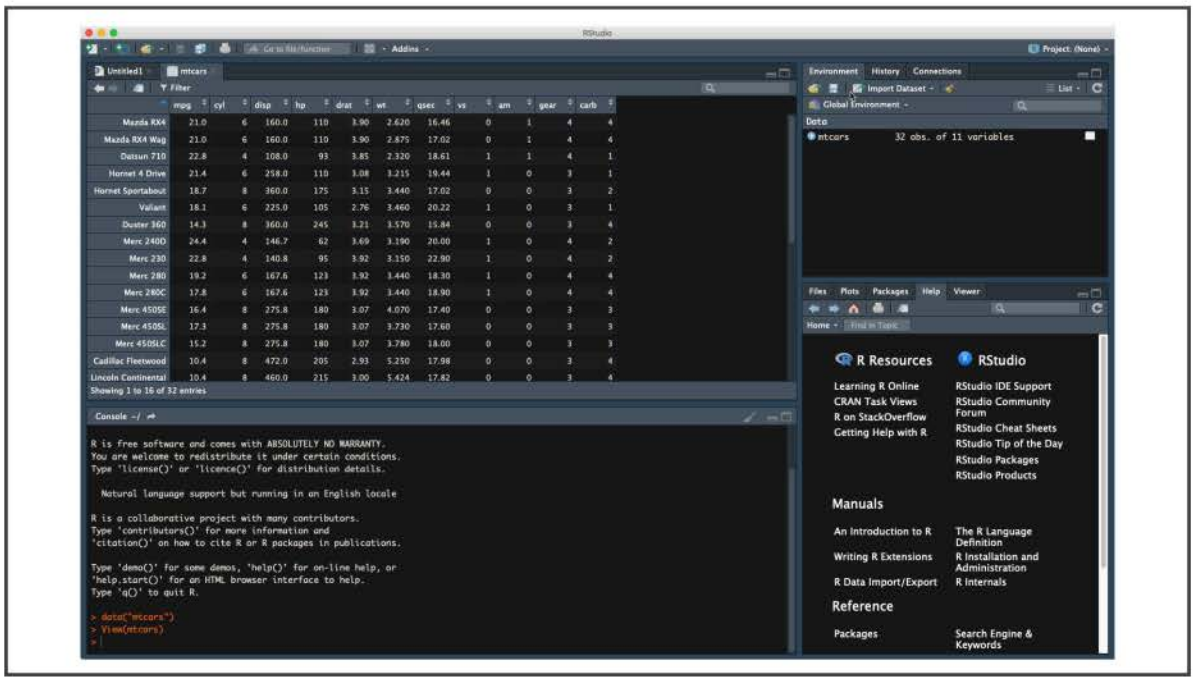
Just as Jupyter Notebooks are an integrated development environment (IDE) used to help design and test Python scripts, RStudio is an IDE used to help design and test RScripts. RStudio provides users a graphical user interface (GUI) with multiple dynamic windows and perpetual access to their RScript and R console.

Similar to Jupyter Notebooks, RStudio enables users to test their analysis scripts line by line while allowing users to view different environment variables and outputs. This means that for each line of code written and executed, users can verify the results and troubleshoot any problems quickly and easily.

The following image shows what an empty session in RStudio will look like:



And here's what an active RStudio looks like. Note the DataFrame in the top left pane and the mtcars object in the top right pane:



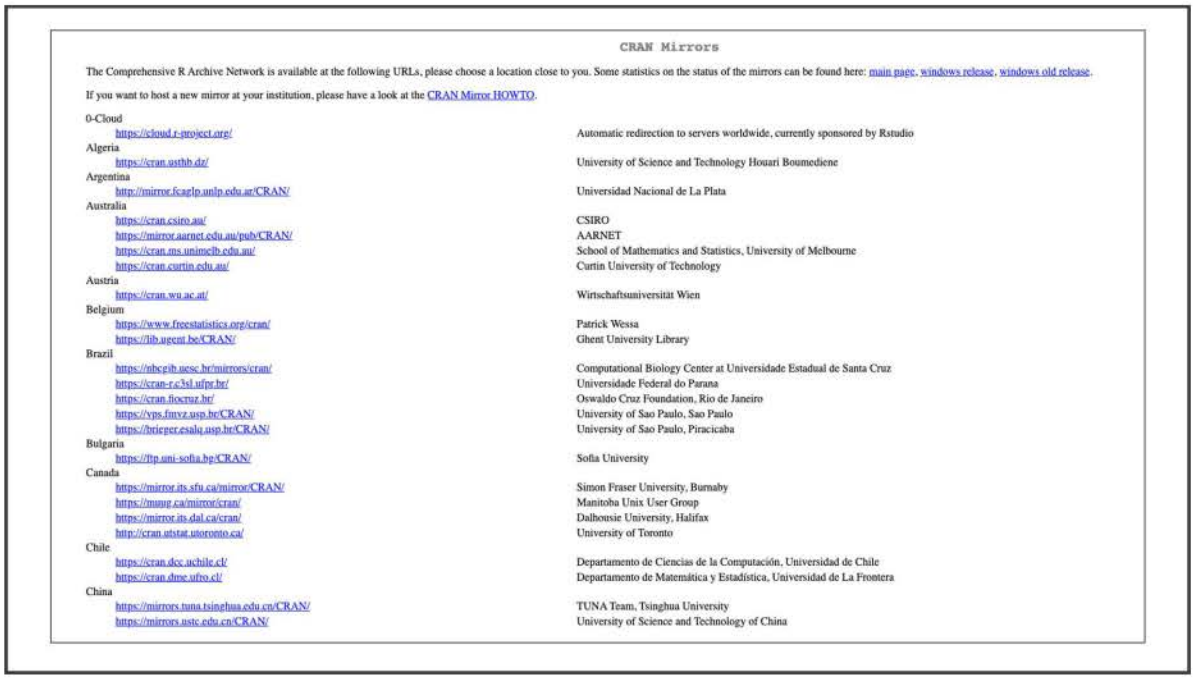
Now that you know what R and RStudio do, let's install them on your machine.

15.1.2: Install R

Now that he has a bit of background on R and RStudio, Jeremy is ready to get downloading. First he'll download R itself and then move on to RStudio.

We must first install R before installing RStudio. This way, RStudio can easily find our R installation while being configured; otherwise, we would have to manually tell RStudio where to find our installed applications.

To install R on macOS or Windows, navigate to **R's Comprehensive R Archive Network (CRAN) server (https://cran.r-project.org/mirrors.html)** and select a mirror link near our region. In most cases, any U.S. mirror link will do (see the following images):

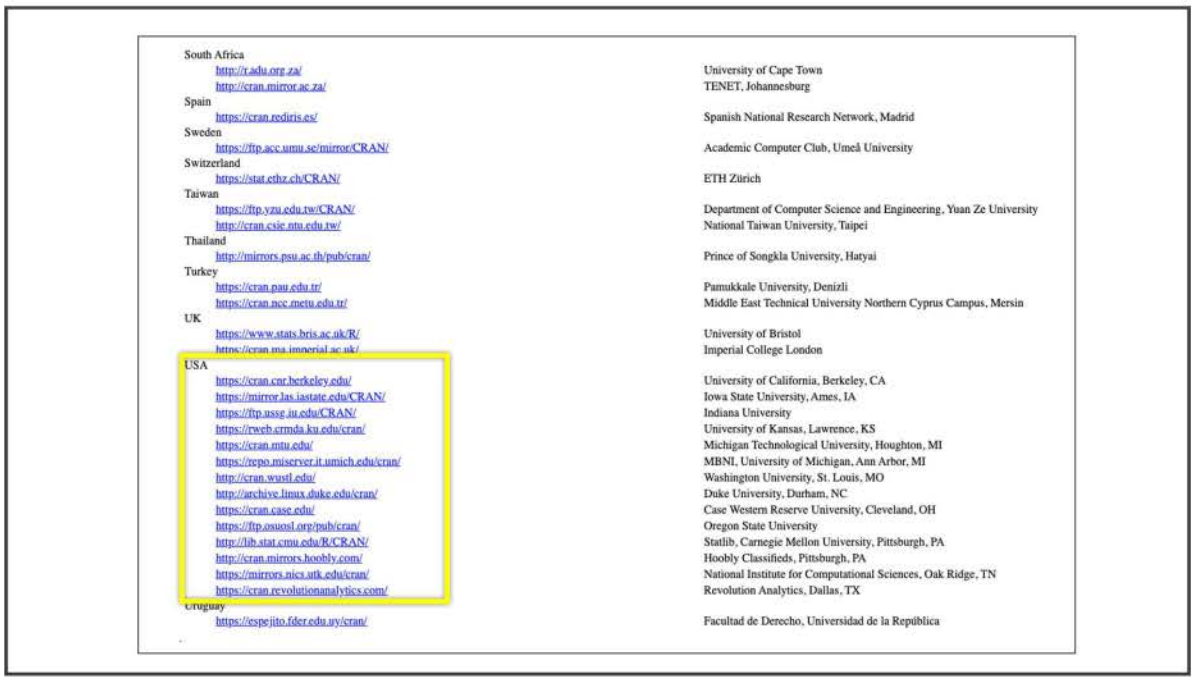


CRAN Mirrors

The Comprehensive R Archive Network is available at the following URLs, please choose a location close to you. Some statistics on the status of the mirrors can be found here: main page, windows release, windows old release.

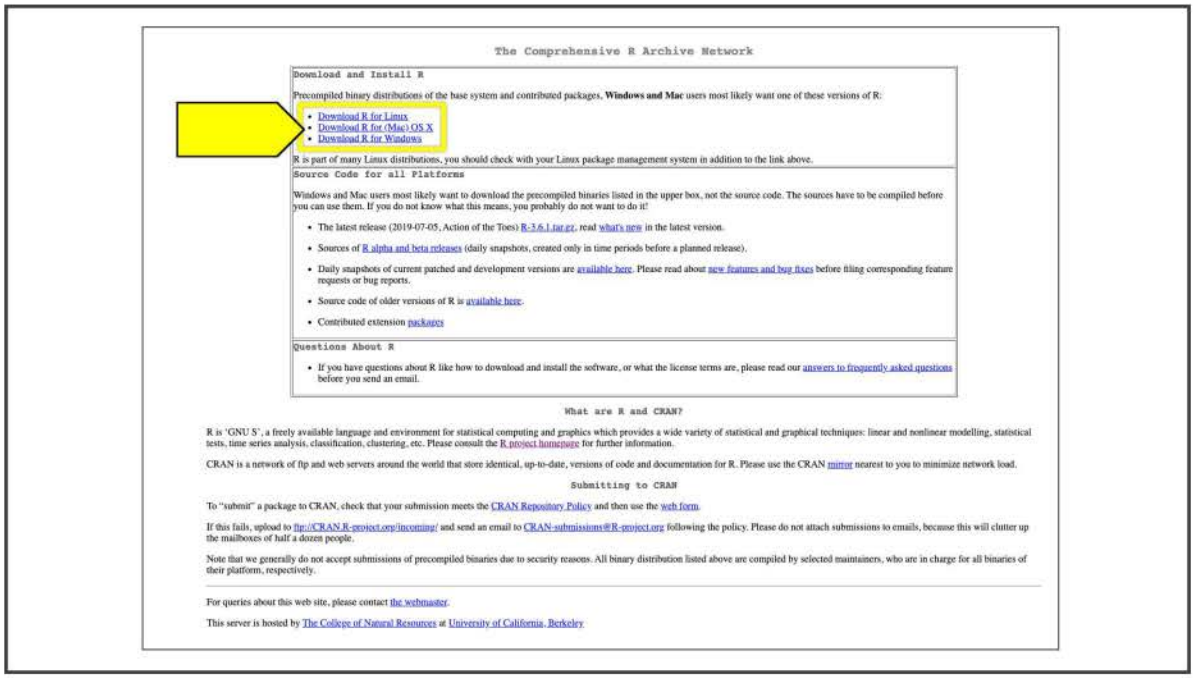
If you want to host a new mirror at your institution, please have a look at the CRAN Mirror HOWTO.

0-Cloud	
https://cloud.r-project.org/	Automatic redirection to servers worldwide, currently sponsored by Rstudio
Algeria	
https://cran.usthb.dz/	University of Science and Technology Houari Boumediene
Argentina	
http://mirror.fcaglp.unlp.edu.ar/CRAN/	Universidad Nacional de La Plata
Australia	
https://cran.csiro.au/	CSIRO
https://mirror.aarnet.edu.au/pub/CRAN/	AARNET
https://cran.ms.unimelb.edu.au/	School of Mathematics and Statistics, University of Melbourne
https://cran.curtin.edu.au/	Curtin University of Technology
Austria	
https://cran.wu.ac.at/	Wirtschaftsuniversität Wien
Belgium	
https://www.freestatistics.org/cran/	Patrick Wessa
https://lib.ugent.be/CRAN/	Ghent University Library
Brazil	
https://nbcgib.uesc.br/mirrors/cran/	Computational Biology Center at Universidade Estadual de Santa Cruz
https://cran-r.c3sl.ufpr.br/	Universidade Federal do Parana
https://cran.fiocruz.br/	Oswaldo Cruz Foundation, Rio de Janeiro
https://vps.fmvz.usp.br/CRAN/	University of Sao Paulo, Sao Paulo
https://brieger.esalq.usp.br/CRAN/	University of Sao Paulo, Piracicaba
Bulgaria	
https://ftp.uni-sofia.bg/CRAN/	Sofia University
Canada	
https://mirror.its.sfu.ca/mirror/CRAN/	Simon Fraser University, Burnaby
https://muug.ca/mirror/cran/	Manitoba Unix User Group
https://mirror.its.dal.ca/cran/	Dalhousie University, Halifax
http://cran.utstat.utoronto.ca/	University of Toronto
Chile	
https://cran.dcc.uchile.cl/	Departamento de Ciencias de la Computación, Universidad de Chile
https://cran.dme.ufro.cl/	Departamento de Matemática y Estadística, Universidad de La Frontera
China	
https://mirrors.tuna.tsinghua.edu.cn/CRAN/	TUNA Team, Tsinghua University
https://mirrors.ustc.edu.cn/CRAN/	University of Science and Technology of China

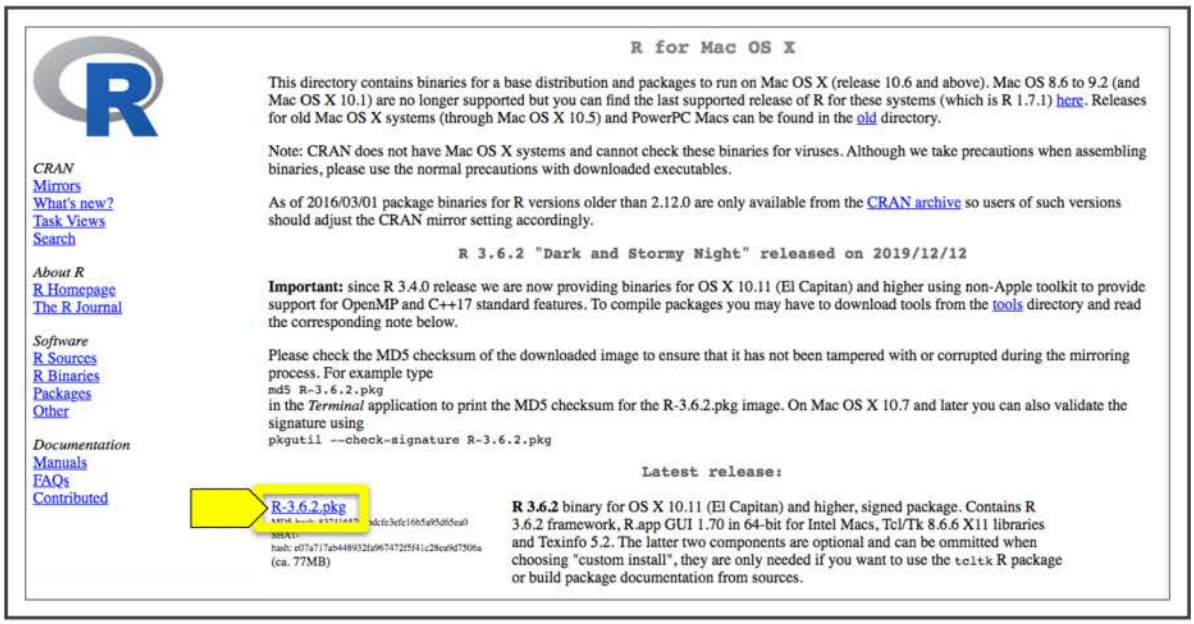


South Africa		
	http://r.adu.org.za/	University of Cape Town
	http://cran.mirror.ac.za/	TENET, Johannesburg
Spain		
	https://cran.rediris.es/	Spanish National Research Network, Madrid
Sweden		
	https://ftp.acc.umu.se/mirror/CRAN/	Academic Computer Club, Umeå University
Switzerland		
	https://stat.ethz.ch/CRAN/	ETH Zürich
Taiwan		
	https://ftp.yzu.edu.tw/CRAN/	Department of Computer Science and Engineering, Yuan Ze University
	http://cran.csie.ntu.edu.tw/	National Taiwan University, Taipei
Thailand		
	http://mirrors.psu.ac.th/pub/cran/	Prince of Songkla University, Hatyai
Turkey		
	https://cran.pau.edu.tr/	Pamukkale University, Denizli
	https://cran.ncc.metu.edu.tr/	Middle East Technical University Northern Cyprus Campus, Mersin
UK		
	https://www.stats.bris.ac.uk/R/	University of Bristol
	https://cran.ma.imperial.ac.uk/	Imperial College London
USA		
	https://cran.cnr.berkeley.edu/	University of California, Berkeley, CA
	https://mirror.las.iastate.edu/CRAN/	Iowa State University, Ames, IA
	https://ftp.ussg.iu.edu/CRAN/	Indiana University
	https://rweb.crmda.ku.edu/cran/	University of Kansas, Lawrence, KS
	https://cran.mtu.edu/	Michigan Technological University, Houghton, MI
	https://repo.miserver.it.umich.edu/cran/	MBNI, University of Michigan, Ann Arbor, MI
	http://cran.wustl.edu/	Washington University, St. Louis, MO
	http://archive.linux.duke.edu/cran/	Duke University, Durham, NC
	https://cran.case.edu/	Case Western Reserve University, Cleveland, OH
	https://ftp.osuosl.org/pub/cran/	Oregon State University
	http://lib.stat.cmu.edu/R/CRAN/	Statlib, Carnegie Mellon University, Pittsburgh, PA
	http://cran.mirrors.hoobly.com/	Hoobly Classifieds, Pittsburgh, PA
	https://mirrors.nics.utk.edu/cran/	National Institute for Computational Sciences, Oak Ridge, TN
	https://cran.revolutionanalytics.com/	Revolution Analytics, Dallas, TX
Uruguay		
	https://espejito.fder.edu.uy/cran/	Facultad de Derecho, Universidad de la República

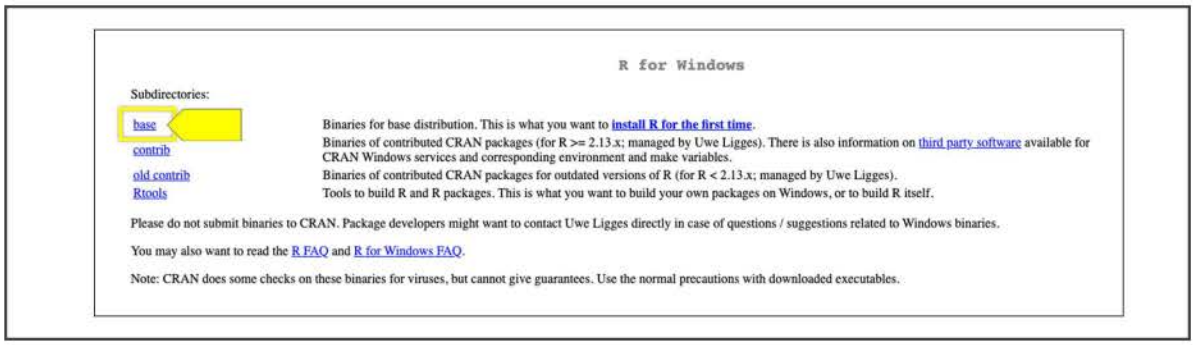
After you navigate to a CRAN mirror site, you'll reach a self-explanatory download page. Follow the appropriate download link for either your macOS or Windows environment:



For those running a macOS environment, select the latest release `.pkg` file (the link is midpage):



For those running a Windows environment, click on the base installer link. On the next page, click the “Download R for Windows” link to start downloading the installer:



Once your installer files are successfully downloaded (`.pkg` for macOS or `.exe` for Windows), run them just as you would for any other installation program. Use all default install options and, if prompted, check all boxes to allow all R components to install.

In the command line, type `R --version`. Does your terminal show the R version information without errors?

- Yes
- No

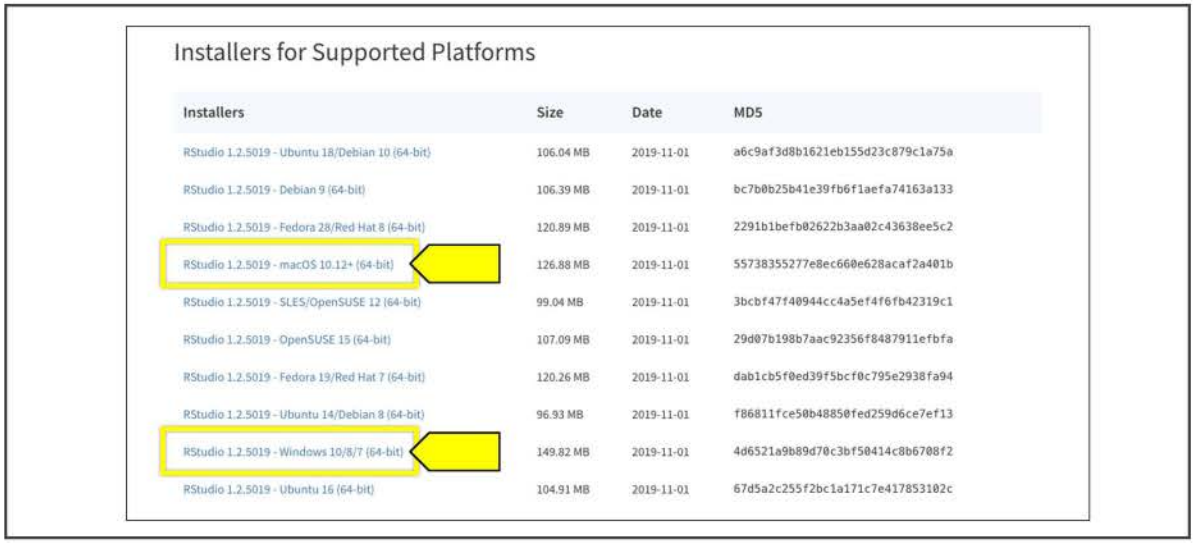
Check Answer

Finish ▸

15.1.3: Install RStudio

Jeremy's new leadership role is starting to feel a bit more real now that R is actually installed. Jeremy's next step is to install RStudio and some libraries his lead analyst suggested might be helpful.

Once you have completed the installation for R, it's time to install RStudio. Now navigate to the **RStudio Download Page (https://rstudio.com/products/rstudio/download/?utm_source=downloadrstudio&utm_medium=Site&utm_campaign=home-hero-cta#download)** and select the most appropriate installer link. Refer to the following image:

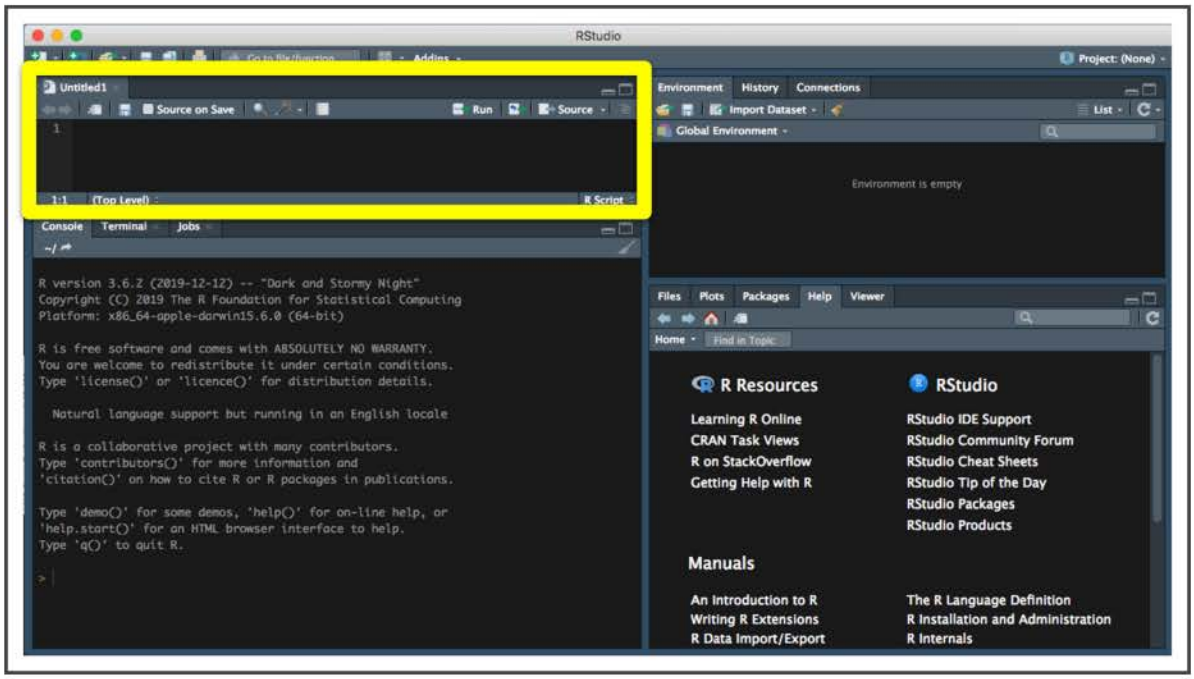


If you're using macOS, drag the RStudio application into your application folder. If you're a Windows user, run it through the installer as you would with any other Windows program.

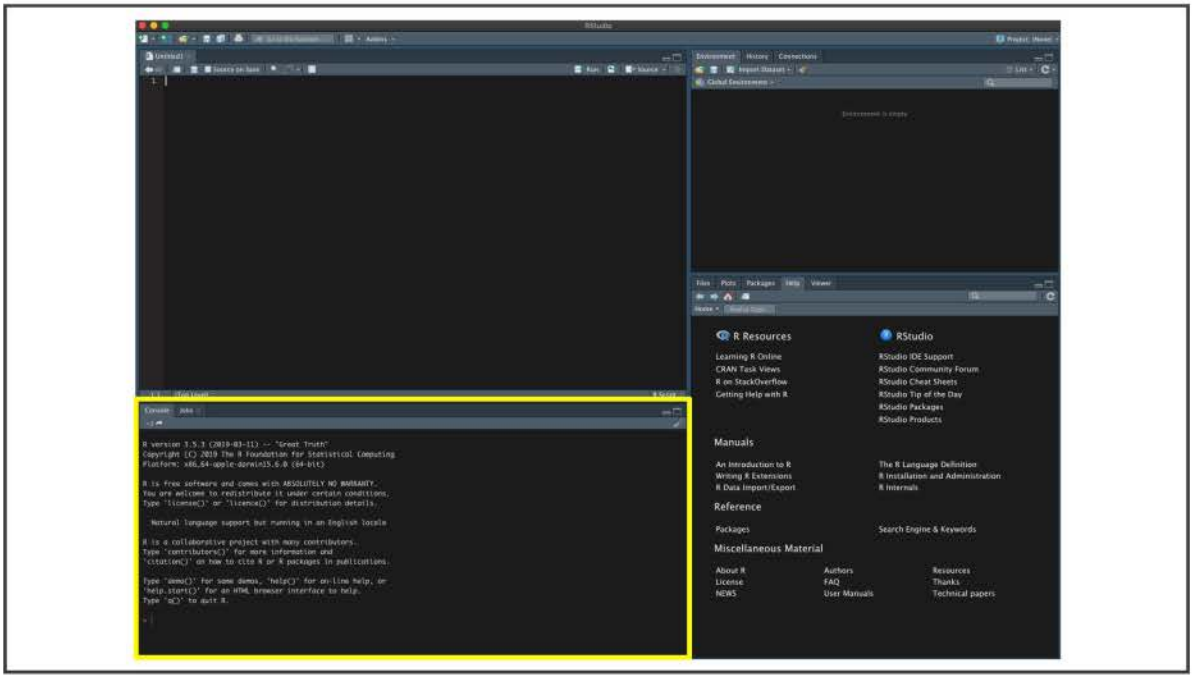
Once you have R and RStudio installed, run RStudio for the first time, get acquainted with the software, and install our required packages.

Navigate and Configure RStudio

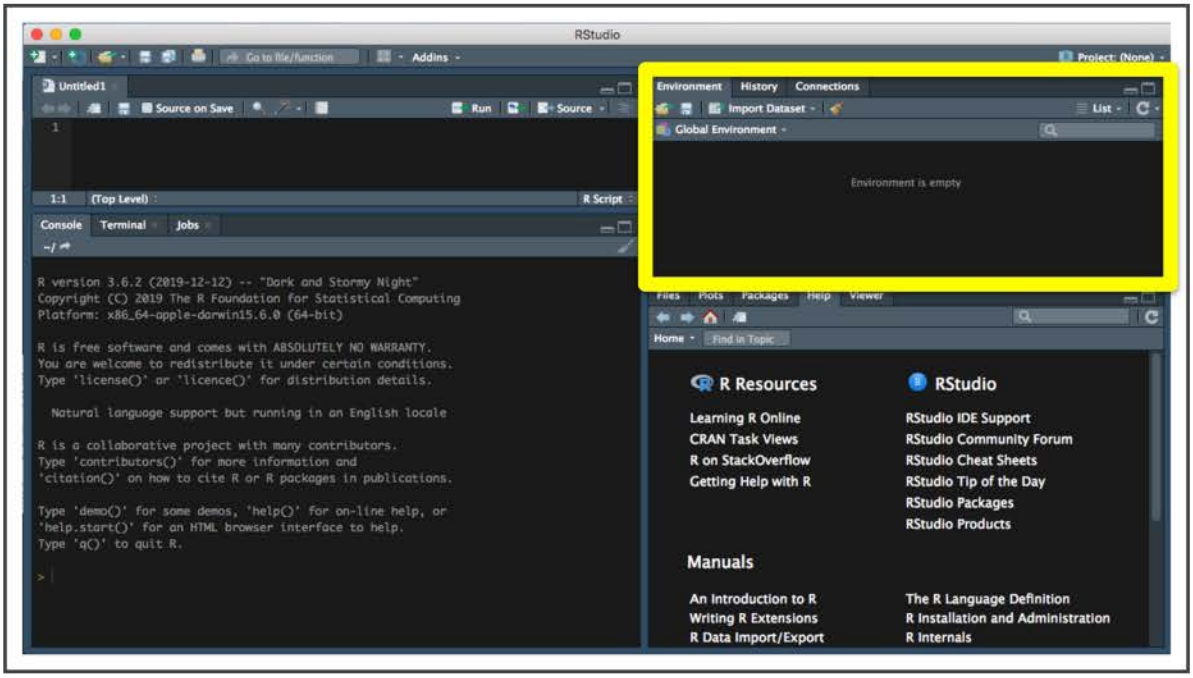
When you first open up RStudio, you'll notice four panes laid out within the application window. The top-left pane contains your source, or (any RScripts, tables, and files you open within RStudio). By default, RStudio will open an untitled RScript file in the pane for you, so you can start programming right away:



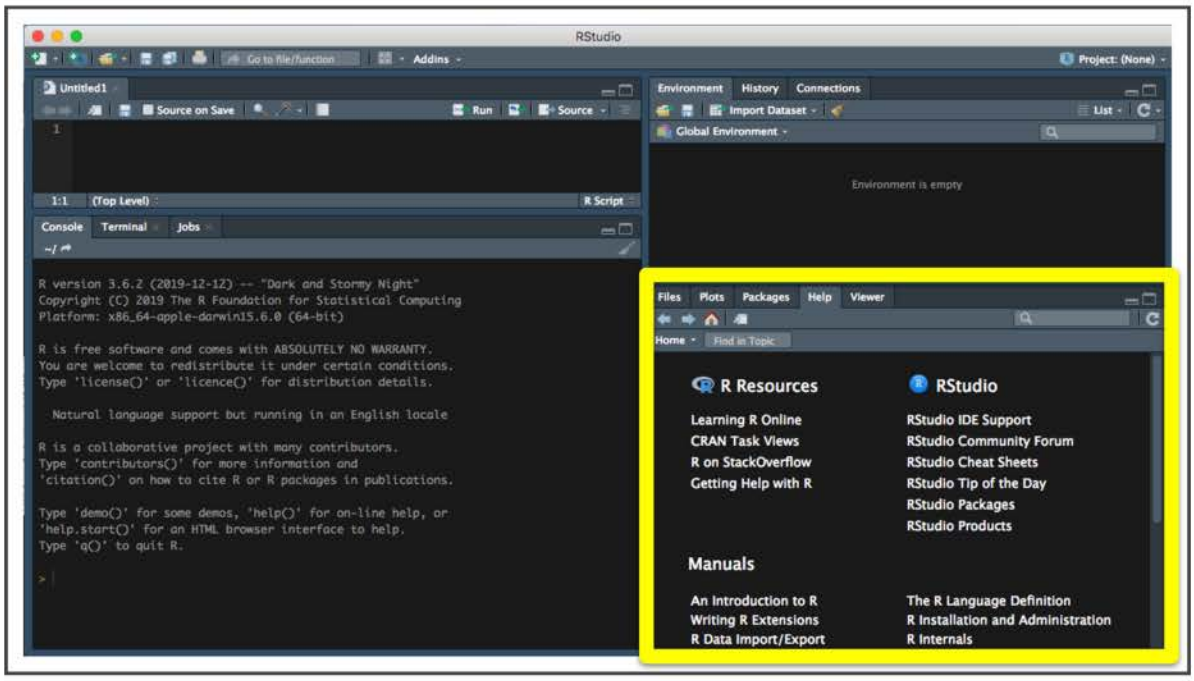
The bottom-left pane contains the R console. Similar to Python, R can either run an RScript as an executable script or R can run interactively. RStudio combines the best of both worlds where the source RScript (in the top-left pane) can be run all at once, or line by line. By including the R console within the application, we can interact with our environment in real time and test parts of our code before we write them in our scripts:



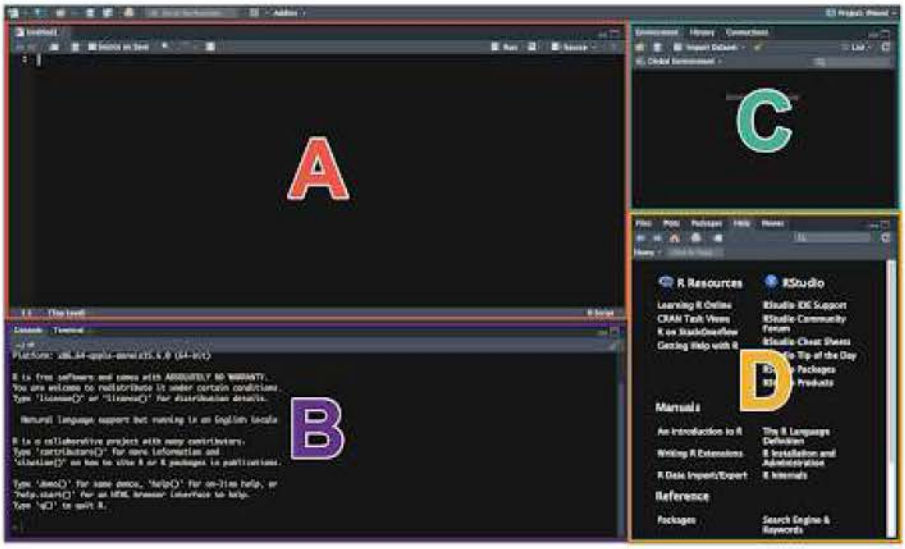
The top-right pane contains our environment objects, such as variables, functions, and data frames. As we execute commands in the R console, either using our source RScript or manually, any objects generated in the R environment will show up in the top-right pane. This environment pane helps us keep track of the shape, data type, and contents of each variable within our environment without having to print out our variables in the console. As we explore R in this module, the environment pane will prove even more useful for tracking what each line of code does to our data:



On the bottom right is the multi-tool pane, which contains tabs for a file explorer, R documentation help, installed package list, and a plot viewing tool. Later, we'll refer to the Plots tab for exploring our generated plots. Additionally, you can use the Files tab to open RScripts from your computer or to copy file paths to include within your RScripts. Finally, to learn more about a function or object from a library in R, simply type `?<name of function or object>` in the R console to open the documentation in the Help tab of the multi-tool pane:



Match the names of the RStudio panes to the correct quadrant in the following image:



A

B

C

D

Multi-tool Pane

Environment Pane

Source Pane

R Console Pane

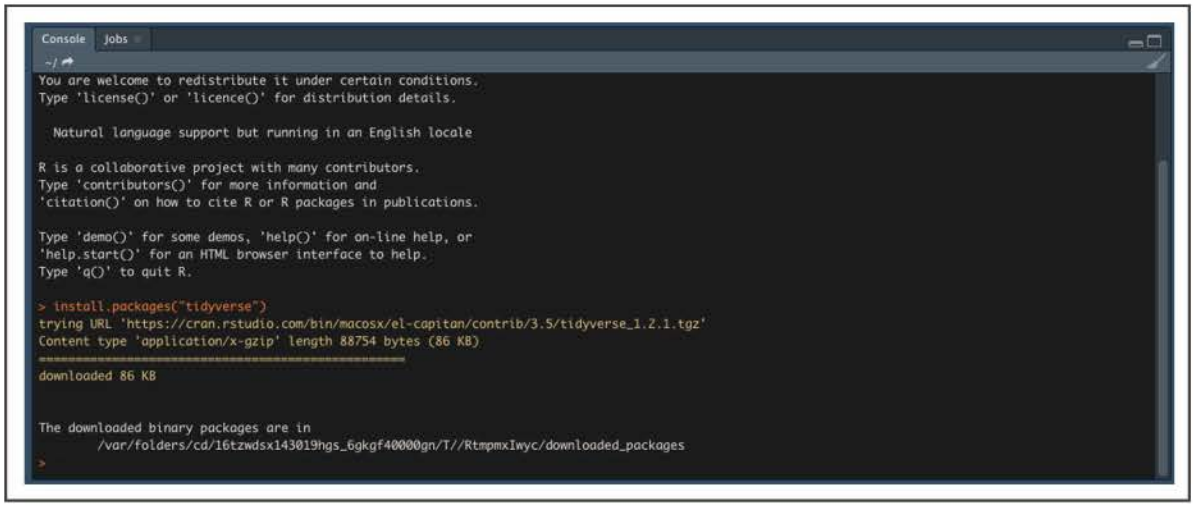
Check Answer

Finish ▸

Now that we understand RStudio's layout, we'll install our required libraries to use them in our RScripts for this module. Thankfully, R developers have built robust library collections, such as the **tidyverse** **(https://www.tidyverse.org/)**, that simplify the installation process for the most common data analysis packages in R. To install packages in our R environment, use the `install.packages()` function.

Therefore, to install the tidyverse in our R environment, simply run the following command in the R console:

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



We'll need the `jsonlite` library for this module. To install our other required libraries and packages, run the following in the R console:

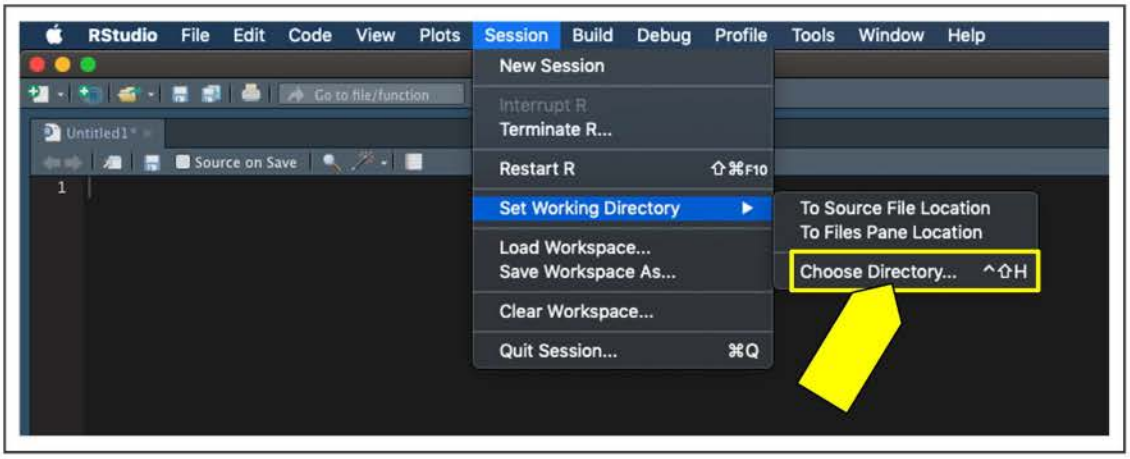
```
> install.packages("jsonlite")
```

> **Caution**
> When installing packages in R, be sure to wrap the package name in quotation marks, otherwise R will throw an error.

Lastly, before we start learning how to program in R, we need to create a working directory folder structure on our computers. This will allow us to keep all of our RScripts and analysis results in a neat, organized structure and to simplify the process of reading in any external files into our R environment. Follow these steps to create a working directory in R:

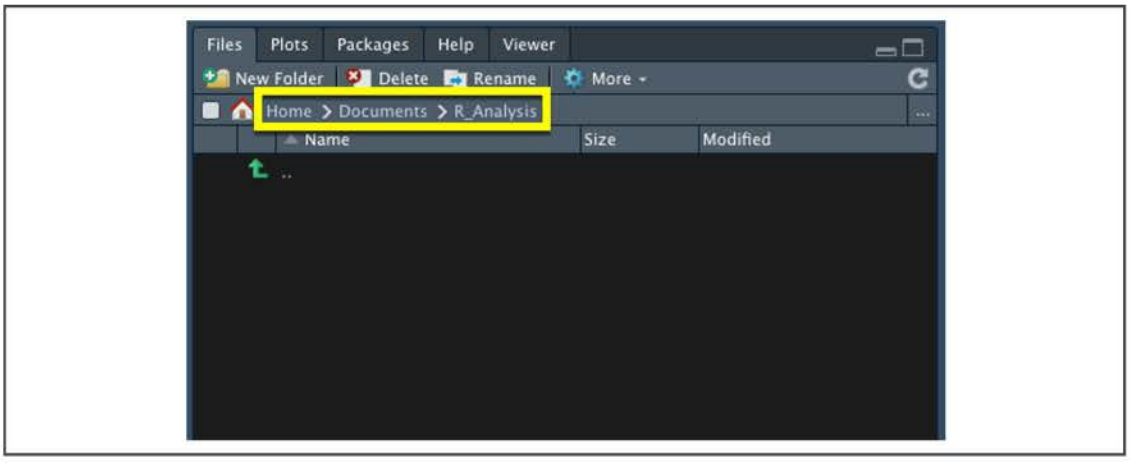
1. Make a folder on your computer called "R_Analysis," or whatever would help identify your R analysis and scripts folder.

2. In the R menu screen, go to Session, click Set Working Directory, then select Choose Directory:

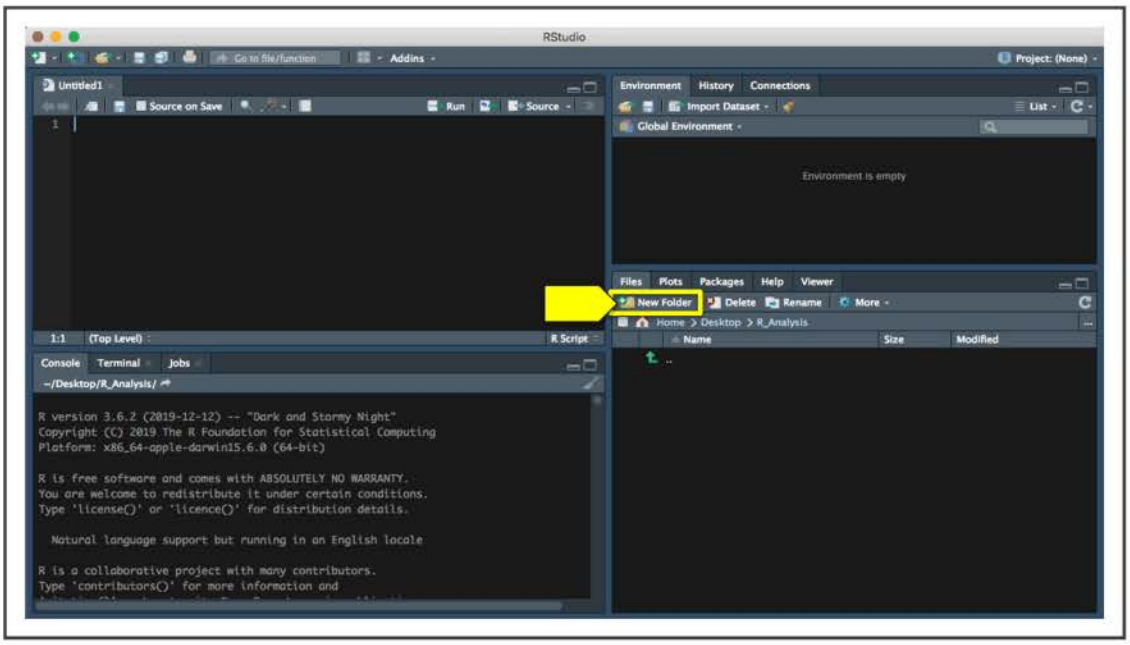


> **Note**
>
> Although your menu bar may look slightly different, the menu options are the same for both macOS and Windows.

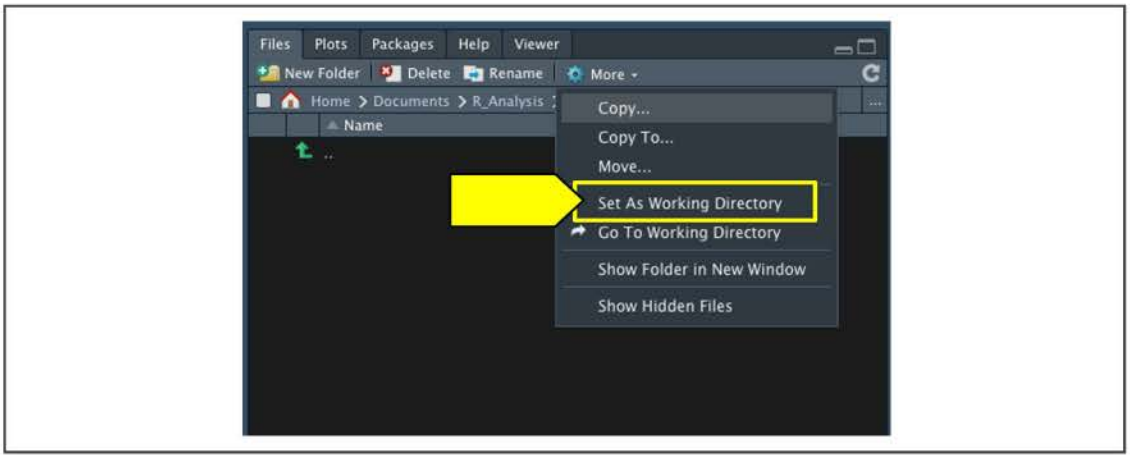
3. Navigate to the folder on your computer and select Open. If you click on the Files tab in your bottom-right multi-tool pane, notice that the folder now represents your active working directory:



4. Now create a new folder in your active directory using the "New Folder" button. For the purposes of this first section of the module, let's name this new folder "01_Demo." Once you have created the folder, press the refresh ( ) button to refresh the directory to see your new folder:



5. Once you have created your new 01_Demo folder, use the file pane and click on the folder to navigate within it. Within the file pane, click on More and select the Set As Working Directory option to make your 01_Demo folder your new working directory:



As you progress through the module generating RScripts and analyzing data using R, it's always a good idea to keep a clean working directory. Feel free to make subfolders within your R_Analysis folder, but be sure to always change your working directory so your output tables and figures do not get lost.

15.2.1: Fundamentals of R Programming

Now that Jeremy has installed R and RStudio, as well as completed some necessary prerequisites, it's time to start programming. Since Jeremy has learned some programming previously, he decides that the easiest way to learn R is to teach himself how to do the functions he is familiar with. Once he feels comfortable loading in data and manipulating the data within R, he can learn more about the functions in R that will be relevant to his new role.

Now that we understand RStudio's layout and have installed our required libraries and packages, it's time to start programming in R. The two fundamental components to programming in R are creating **data structures** and using **functions.**

When learning a new programming language, it's good to always start with the basic structures and functions you're familiar with, and learn how to implement them using the new language. This way, you can map new concepts and documentation to your previous experience. Once you feel you have mastered the basics of the language, you can always learn the more advanced functionality later.

Just like other object-oriented programming languages, R uses named data structures to store values and properties and uses functions to perform operations. The most straightforward R data structures are named values and vectors.

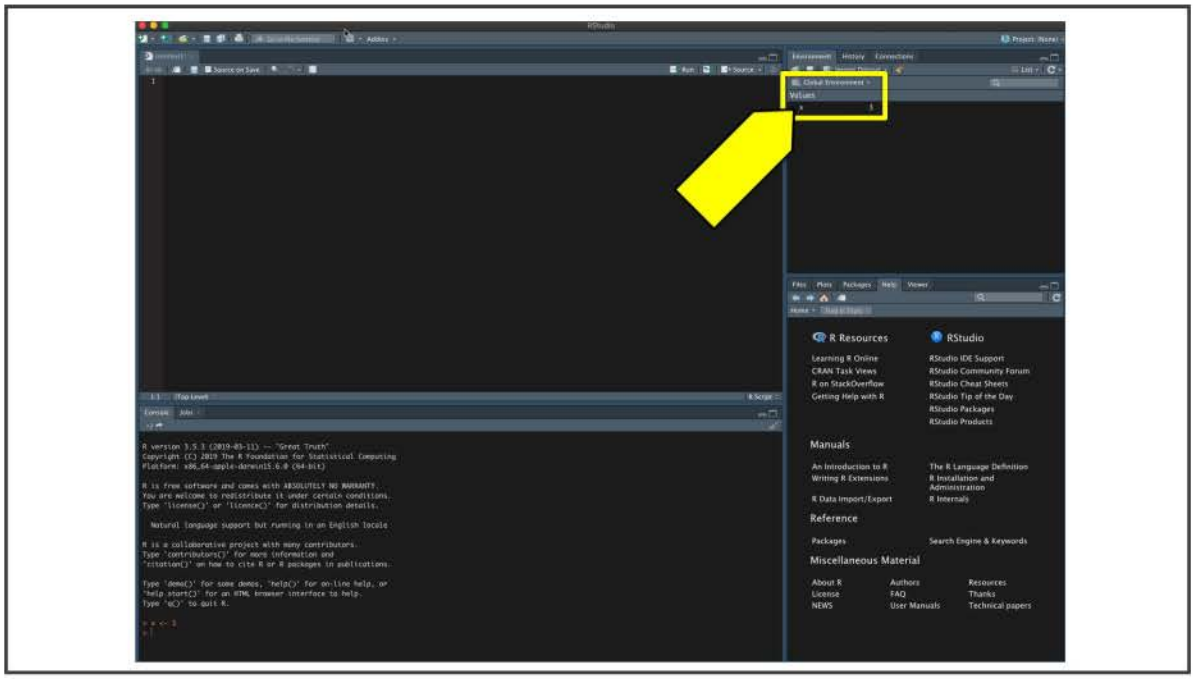
Named values are exactly what they sound like—they are a value that has been given a name. We can think of named values as a variable that has been given a single value. **Vectors** are R's version of arrays, where a list of numbers are assigned a location and stored as a single data structure.

To create our first data structure in R, we use an assignment statement. An assignment statement in R simply tells R the name of the object and assigns a value or data structure to that name.

For example, say we want to create a named value *x* and assign it a value of 3, we would use this R command:

```
> x <- 3
```

The **assignment operator** (`<-`) tells R to assign whatever is right of the arrow to the name that is left of the arrow. In this case, we have given 3 the name of "x." This is similar to assigning a variable in Python, except we use an assignment operator instead of the equals sign (=). Take a moment to run the command in your R console. This next image shows an RStudio session with the newly created "x" value:



Did you notice that after you created a named value in the console, that named value appeared in your environmental pane? No matter what data structure you use, as you assign objects into your environment, they will appear in this pane. But what happens if you wanted to change that value to something new?

In R, all environment objects are mutable, which means they can be assigned and reassigned multiple times. If we want to assign a new value to *x*, we can do so using an assignment operator to assign a new value.

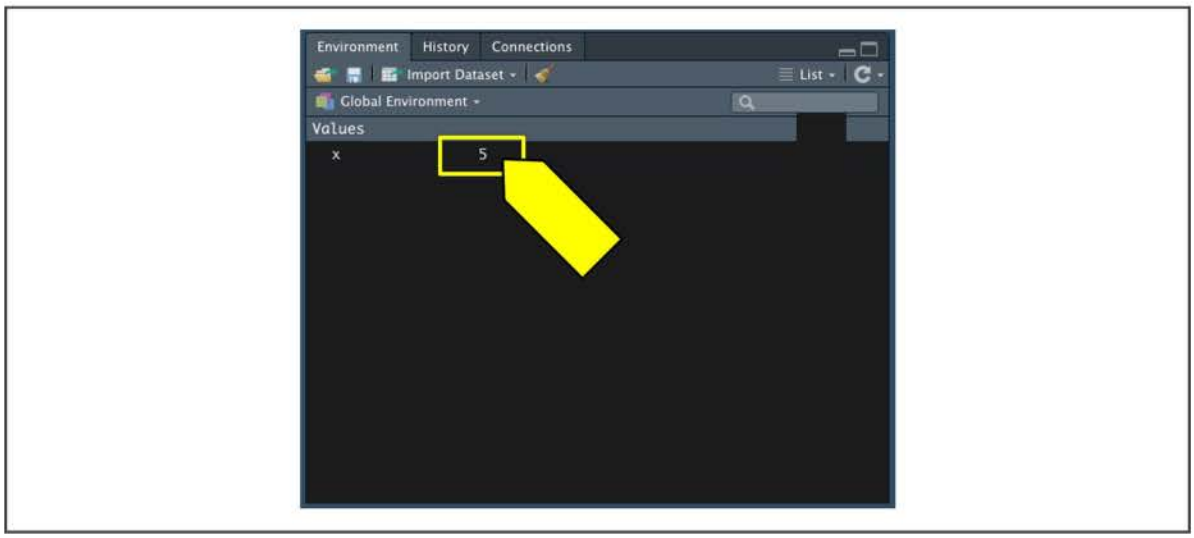
Write an R statement to create a named value x and assign it a value of 3.

Write an R statement to reassign the named value x to the value of 5.

Check Answer

Finish ▸

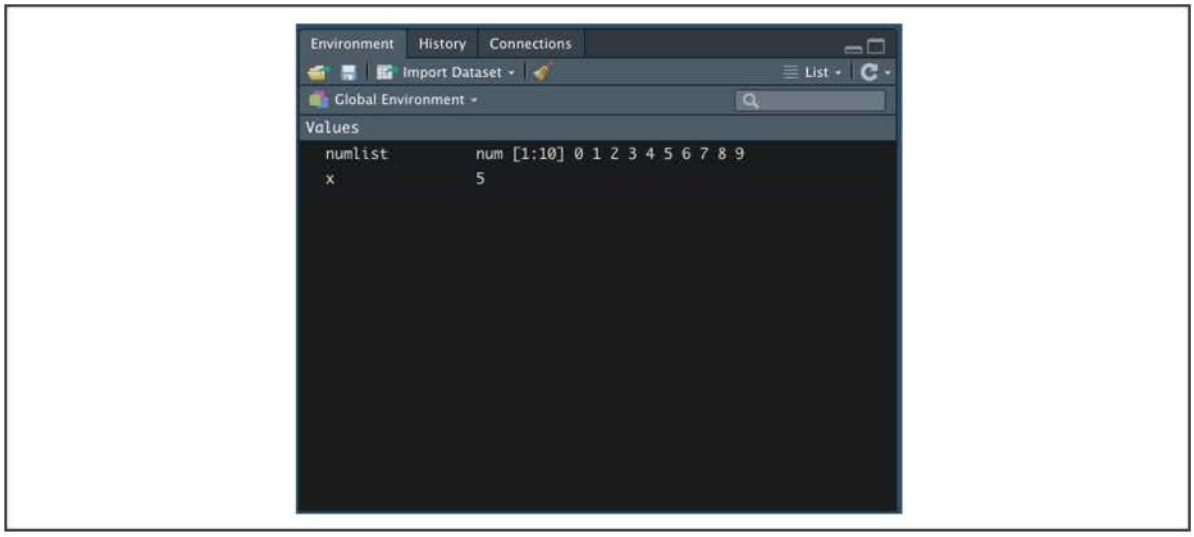
And if we look back to our environment pane, our named value will have been reassigned and the changes reflected in real time:



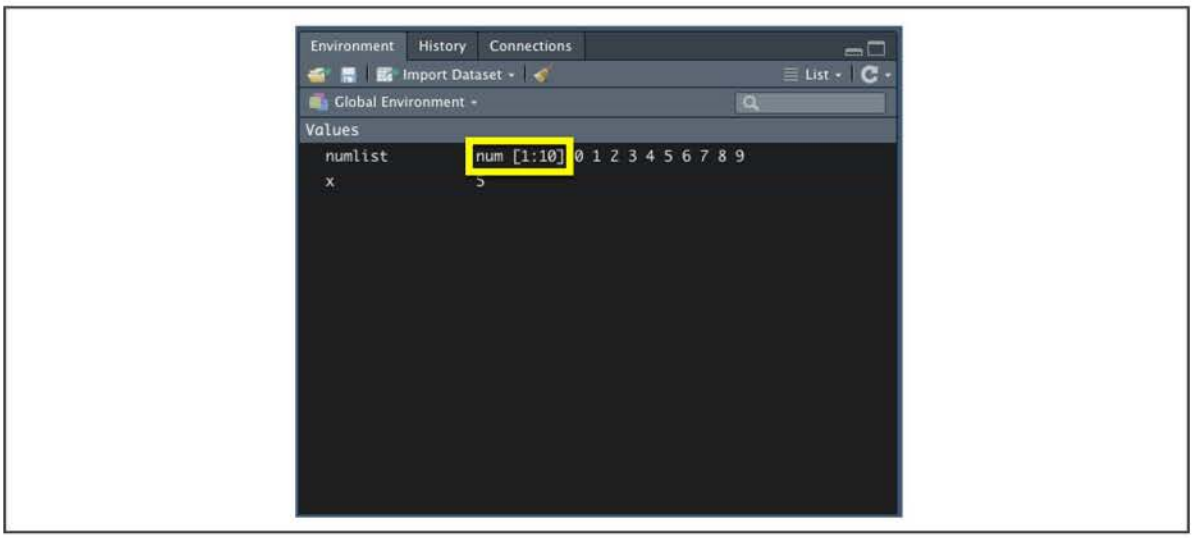
The other simple data structure in R is the numeric vector. A **numeric vector** is an ordered list of numbers, very similar to a **numeric list** in Python. To create a numeric vector, we use the `c()` function. The `c()` function is short for concatenate, which means to link together. In R, we link together a comma-separated list of values into a single numerical vector. For instance, if we want to make a list of numbers from 0 to 9, we can pass the following command into the R console:

```
> numlist <- c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

The result of this command would assign the vector `numlist` into the environment:



Notice that the environment pane shows the object name. If the object is not a named value, it will provide the data type and dimension. In this case, the `numlist` is a numeric (num) object with one row and 10 columns of values, as shown in the following image:



R also supports a number of more advanced data structures such as **matrices**, **data frames**, and **tibbles**—all of which are variations of the same data frame concept:

- A **matrix** can be thought of as a vector of vectors, where each value in the matrix is the same data type.
- A **data frame** is very similar to a Pandas DataFrame where each column can be a different data type.
- A **tibble** is a recent data object introduced by the tidyverse package in R and is an optimized data frame with extra metadata and features. The most current libraries and packages in R use data frames or tibbles; however, older R packages and analysis scripts will still use matrix objects to perform specific functions or analyses.

Now that we are familiar with assigning data structures and looking at environment objects, it's time to look at the other half of programming in R—using functions.

> **Note**
> Those curious about the more advanced R data structures can refer to the **R Introduction documentation (https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf)** .

15.2.2: Functions in R

What functions will Jeremy need to master in R? His lead analyst, Colleen, suggested he start with the structure of functions themselves. Functions in R have a similar structure as Python functions, so Colleen thinks Jeremy will get the hang of it pretty quickly.

Functions behave similarly to methods (or even functions) in Python. In R, a function is used to perform a specific task and is denoted by parentheses. Functions can either be built-in, come from imported libraries, or be defined by the user.

In this module, we have already used two built-in functions—the `install.packages()` function and the `c()` function. One required the name of the package to install and returned no value. The other could contain an unlimited number of arguments and would return a vector containing a list of equal size and order.

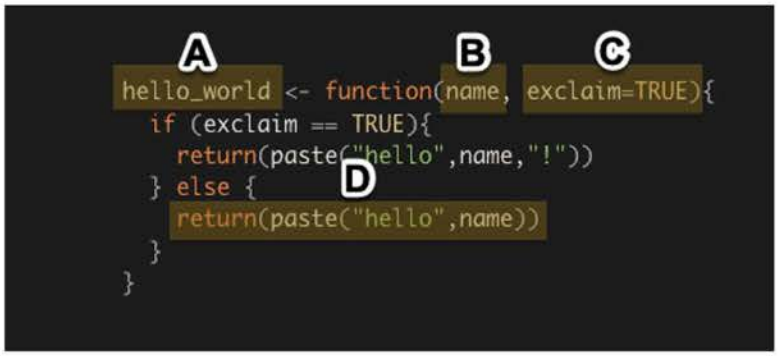
Regardless of where a function comes from, all R functions use the same basic syntax:

```
function_name <- function(arg1, arg2=T, …){

<BODY OF FUNCTION>

return <RETURN VALUE>
}
```

There are four components of an R function:

- The **function name** is the name of the function, which can be used in the R console to call the function itself.
- Just like Python methods, R functions can have any number of **required** or **optional arguments**, depending on the design of the function.
- The **function body** includes data structures, if-statements, and other logical statements that define what the function does.
- The **return statement** is the last evaluated statement before returning the resulting value out of the function.

Match the following function components to the elements in the following image:



A

B

C

D

Function Name | Optional Argument | Return Statement | Required Argument

Check Answer

Finish ▸

As we continue learning how to analyze and program using R, we'll encounter a wide variety of functions with different inputs, arguments, and outputs. Many of these functions, arguments, and outputs will be very similar to their Python counterparts.

If at any point you are unsure what an R function does or what it needs to execute, you can always type `?<name of function>` in the R console and it'll open the

documentation in the Help pane. As we progress, we'll cover some RStudio shortcuts that help us easily navigate and implement obscure functions that we might need in specific situations.

Now that we have learned about assigning data structures and using functions, it's time to bring these concepts together and write our very first RScript.

15.2.3: Read and Write Using R

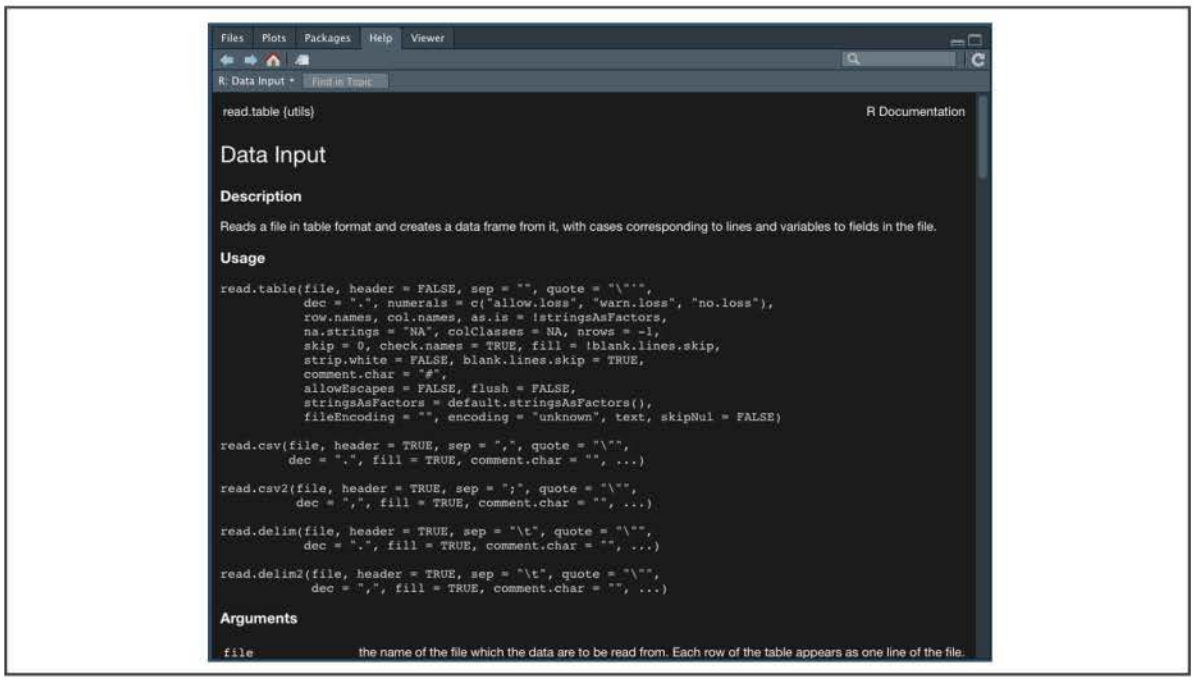
Now that Jeremy understands how to structure a function in R, he's ready to start loading up some data.

Data analysis and visualization typically begin with reading in an external data source into our programming environment.

There are built-in R functions to import the most common data formats, such as comma-separated values (CSV) and JavaScript Object Notation (JSON), as well as plenty of documentation and support online to import more advanced data structures.

To read in a CSV file, we use R's `read.csv()` function. `read.csv()` has a few required arguments to work properly. To identify the required arguments, type the following code into the R console to look at the `read.csv()` documentation in the Help pane, listed under the subhead "Usage" in the image below:

```
> ?read.csv()
```



As we can see from the documentation, `read.csv()` is one of many `read` functions that all serve the same purpose: to read in tabular, character-delimited files and create a data frame object within our R environment.

Depending on what delimiter (or value-separating character) is used, we can use `read.csv()` for comma-delimited files, `read.delim()` for tab-delimited files, or `read.table()` if we need to manually tell the function what delimiter is used.

Although optional arguments are used to parse more complicated datasets, for our purposes, we'll only concentrate on the following arguments:

- **file**
- **header**
- **sep**
- **check.names**
- **stringsAsFactors**

Referring to the R documentation for the `read.csv()` function, fill in the name of the argument that corresponds to each of the following descriptions:

1. [ ] tells the function if a header is present in the CSV file. By default, this is TRUE.
2. [ ] tells the function that if the column is a string data type, to cast it as a factor. We'll discuss factors later in the module, but most of the time we'll want to manually create our factors. Therefore, we need to set this flag to FALSE.
3. [ ] will tell the function to check for spaces, punctuation, and other characters in the header and, if present, change them to periods (“.”). By default, this is TRUE.
4. [ ] is the local file path of the file we wish to read into our environment. Technically, it is the only required argument to run `read.csv()`.
5. [ ] tells the function what the file uses as a delimiter. By default, this is a comma (“,”).

Check Answer

Now that we understand how to use the function, let's practice reading in a demo CSV file containing a hypothetical coworker's vehicle information.

To practice reading in a CSV file, first download our sample CSV file:

Download demo.csv
(https://courses.bootcampspot.com/courses/138/files/18509/download?wrap=1)
(https://courses.bootcampspot.com/courses/138/files/18509/download?wrap=1)

After `demo.csv` has downloaded, place the data file into your active working directory. Next, we'll use `read.csv()` in our source RScript pane to read in the demo file into our R environment. Type the following code:

```
demo_table <- read.csv(file='demo.csv',check.names=F,stringsAsFactors = F
```

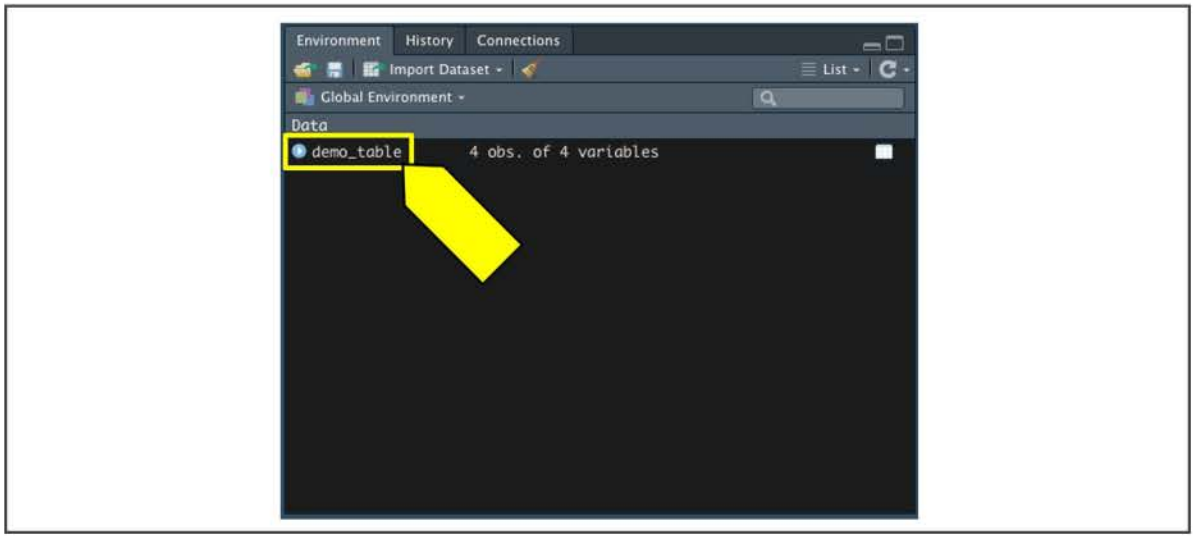
> **Note**
>
> It isn't necessary to put a source file into our active working directory. If we ever want to read in a file from elsewhere on our computer, we would provide the full file path to our file argument.

By writing our read statement in our RScript, we can always quickly create and recreate the data frame by sending our assignment function in the RScript to our R console.

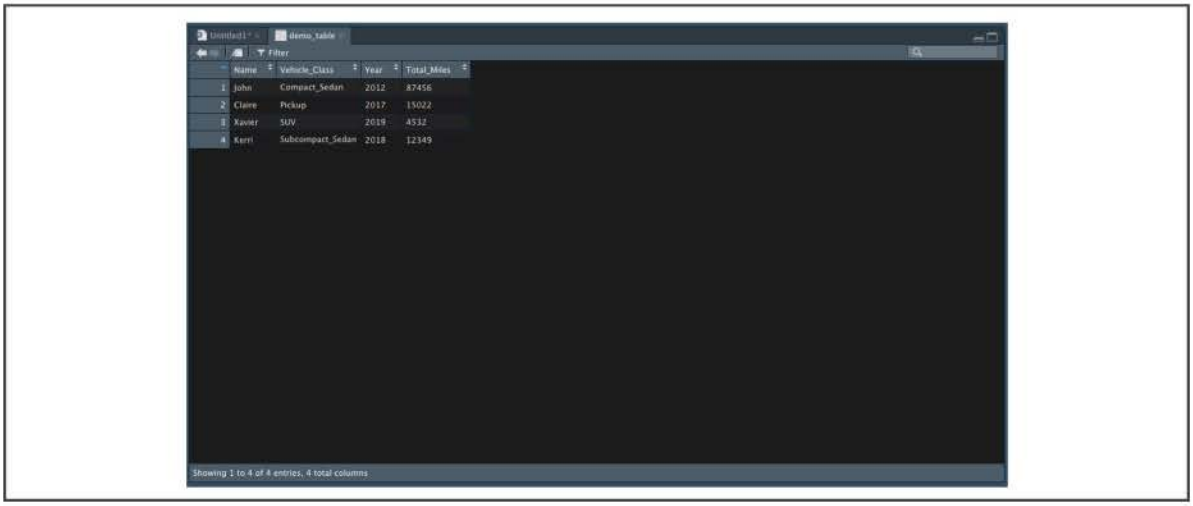
There are two ways to send RScript lines to our R console. We can either use the "Run" button in the top-right source pane or use the following shortcut:

- Command + Enter (Mac)
- CTRL + Enter (Windows)

If we send our `read.csv()` function to the R console, we should see our `demo_table` created in our R environment without any errors, as shown in the following image:



Additionally, if we click on the `demo_table` in our environment pane, it will show us our data frame in a view-only tab in the source pane. Refer to the following image:



This view-only data frame tab can be very helpful when we are trying to transform columns and rows of data in our RScripts, or when trying to select data to use in a visualization or statistical model.

But what if we want to bring in a dataset from an application programming interface (API) query?

What is one of the most popular data formats returned by a URL request?

- CSV
- JSON
- XLSX
- TXT

Check Answer

Finish ▸

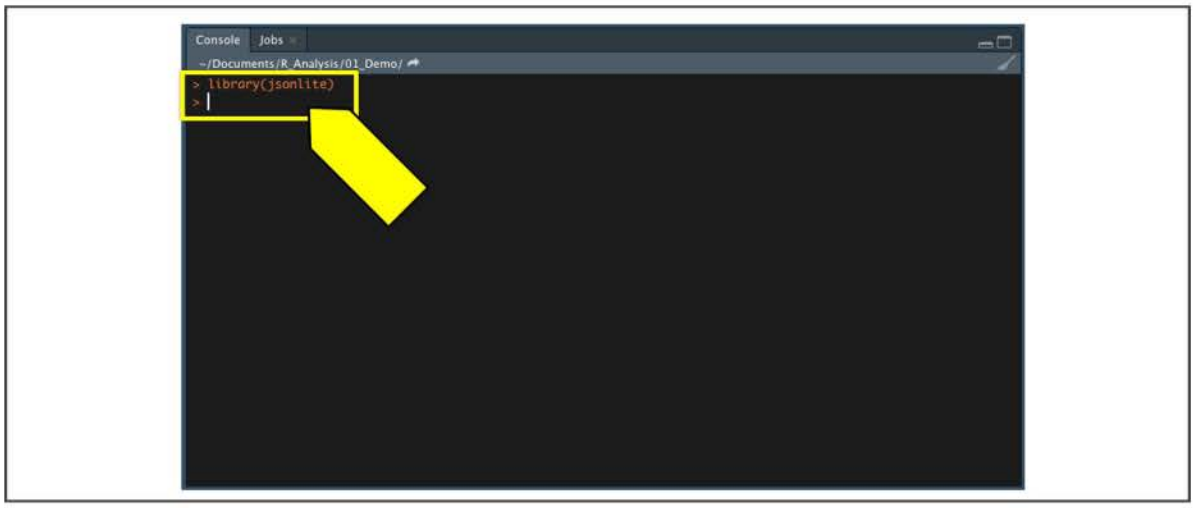
The JSON format is one of the most common data formats returned from a URL request. Although native JSON data can be easier to work with in Python, many data scientists still prefer to use R for their data analysis. To accommodate this need, R developers created the `jsonlite` library to read in JSON data structures and convert them to an R data frame. Because the `jsonlite` library was not built into R, we must import it into our R environment.

To import a library into R, we'll use the `library(package)` function. Just like in Python, it's good practice to import any required libraries at the top of our RScript.

Let's try loading in our installed `jsonlite` package using the `library(jsonlite)` function. Be sure to write the statement in your RScript and then send the statement to your R console.

> **Shortcut**
> When loading a package in R, do not use quotation marks.

If we imported our library correctly, we should see the R console return characters without any errors:



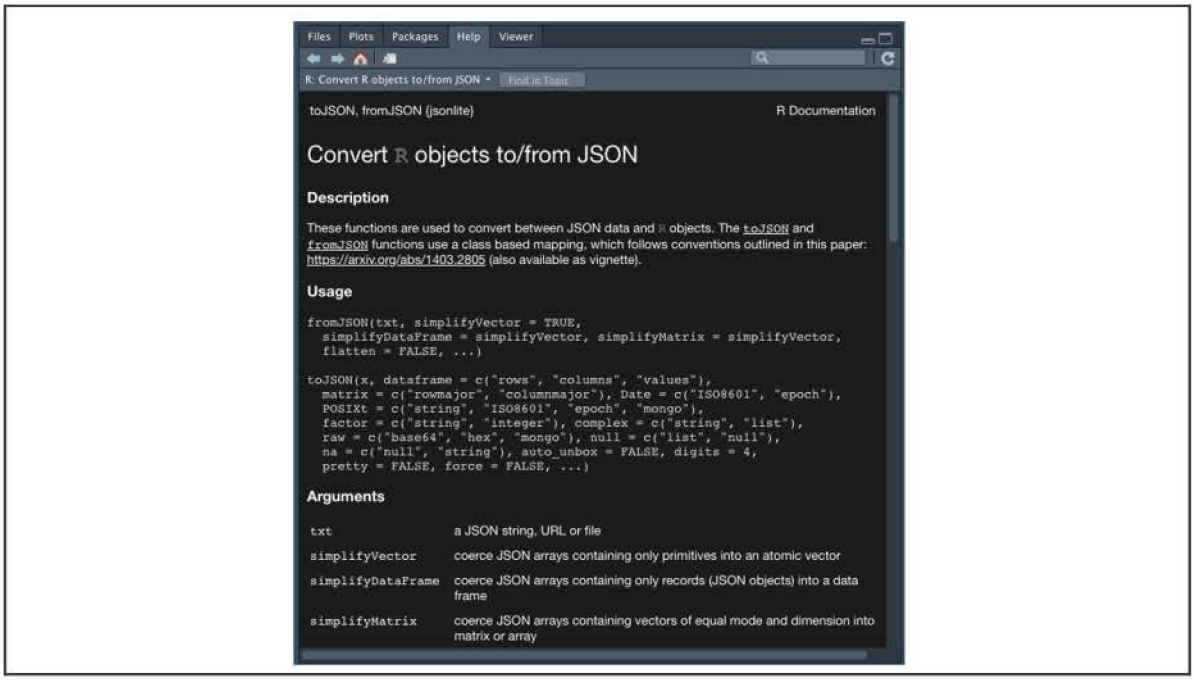
> **Note**
> Often you'll see conflicts or warnings when loading up packages in R, and usually these can be ignored. Only worry if your commands throw an error, which will print out in red text.

If at any time your R console throws errors when trying to import a library, you can always try to reinstall the package using the `install.packages()` function.

Now that we have successfully imported our `jsonlite` package, we can use the `fromJSON()` function to read in a JSON file into R.

First, type the following code into the R console to look at the `fromJSON()` documentation in the Help pane:

```
> ?fromJSON()
```



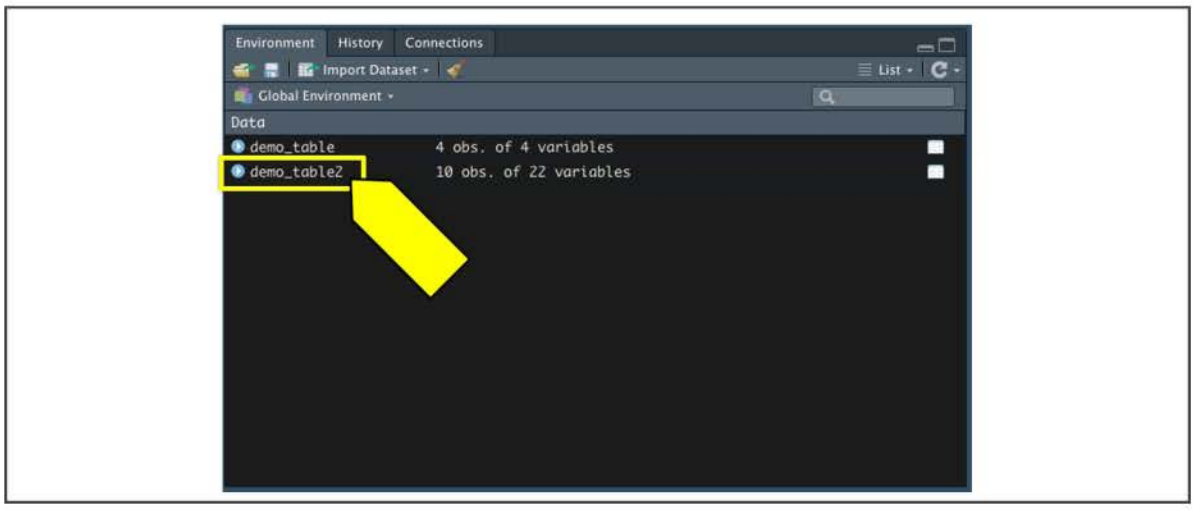
As we can see, we only need to provide the `txt` argument to properly read in a JSON file into R. `txt` is the file path of the JSON file on our machine. Alternatively, we can provide the `fromJSON()` function a JSON URL directly. Now let's practice reading in our first JSON file.

First, download the sample JSON file containing used car data using the link below.

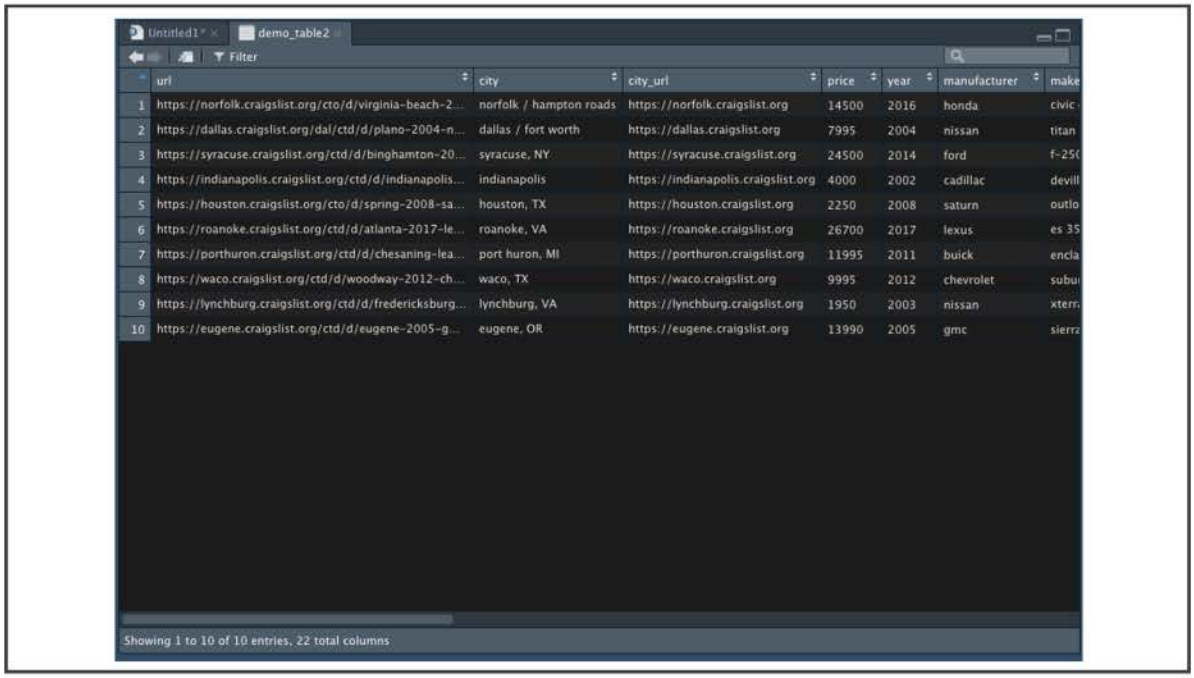
Download demo.json (https://courses.bootcampspot.com/courses/138/files/18500/download?wrap=1)

Place the downloaded data file in your active working directory. Next, use `fromJSON()` in our source RScript pane to read in the used car data into our R environment, as follows:

```
demo_table2 <- fromJSON(txt='demo.json')
```



Once again, if we click the `demo_table2` in our environment pane, it will show us our data frame in a view-only tab in the source pane:



	url	city	city_url	price	year	manufacturer	make
1	https://norfolk.craigslist.org/cto/d/virginia-beach-2...	norfolk / hampton roads	https://norfolk.craigslist.org	14500	2016	honda	civic
2	https://dallas.craigslist.org/dal/ctd/d/plano-2004-n...	dallas / fort worth	https://dallas.craigslist.org	7995	2004	nissan	titan
3	https://syracuse.craigslist.org/ctd/d/binghamton-20...	syracuse, NY	https://syracuse.craigslist.org	24500	2014	ford	f-25(
4	https://indianapolis.craigslist.org/ctd/d/indianapolis...	indianapolis	https://indianapolis.craigslist.org	4000	2002	cadillac	devill
5	https://houston.craigslist.org/cto/d/spring-2008-sa...	houston, TX	https://houston.craigslist.org	2250	2008	saturn	outlo
6	https://roanoke.craigslist.org/ctd/d/atlanta-2017-le...	roanoke, VA	https://roanoke.craigslist.org	26700	2017	lexus	es 35
7	https://porthuron.craigslist.org/ctd/d/chesaning-lea...	port huron, MI	https://porthuron.craigslist.org	11995	2011	buick	encla
8	https://waco.craigslist.org/ctd/d/woodway-2012-ch...	waco, TX	https://waco.craigslist.org	9995	2012	chevrolet	subu
9	https://lynchburg.craigslist.org/ctd/d/fredericksburg...	lynchburg, VA	https://lynchburg.craigslist.org	1950	2003	nissan	xterr
10	https://eugene.craigslist.org/ctd/d/eugene-2005-g...	eugene, OR	https://eugene.craigslist.org	13990	2005	gmc	sierr

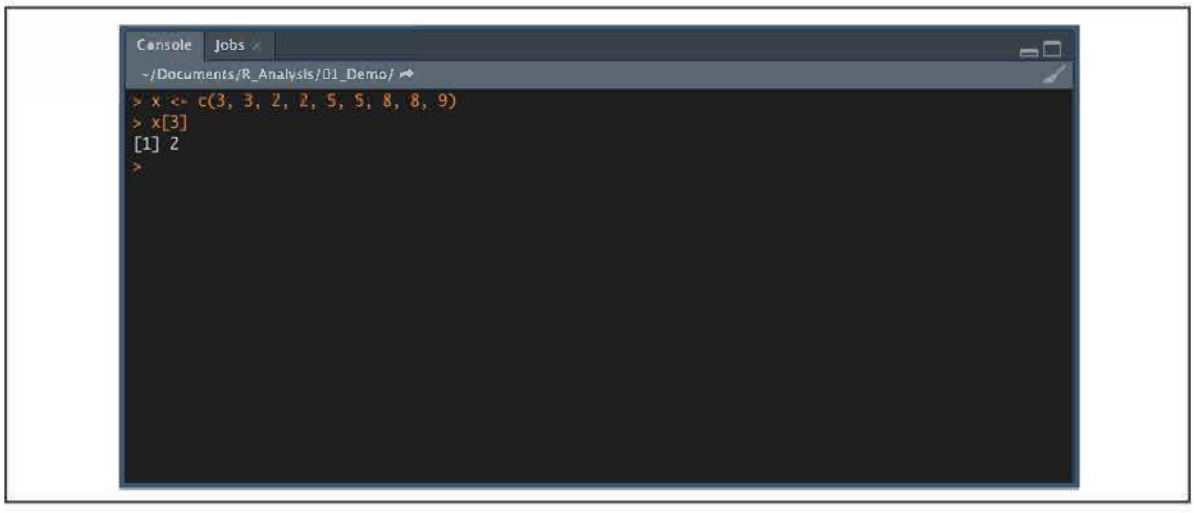
Now that we know how to create data structures and data frames in R, let's learn how to slice and sample our datasets.

15.2.4: Select Data in R

Jeremy has his data loaded up, and he is ready to make some moves! He wants to start actually parsing the data—figuring out what insights he might be able to find for AutosRUs. His first step is to make sure he can select data in R.

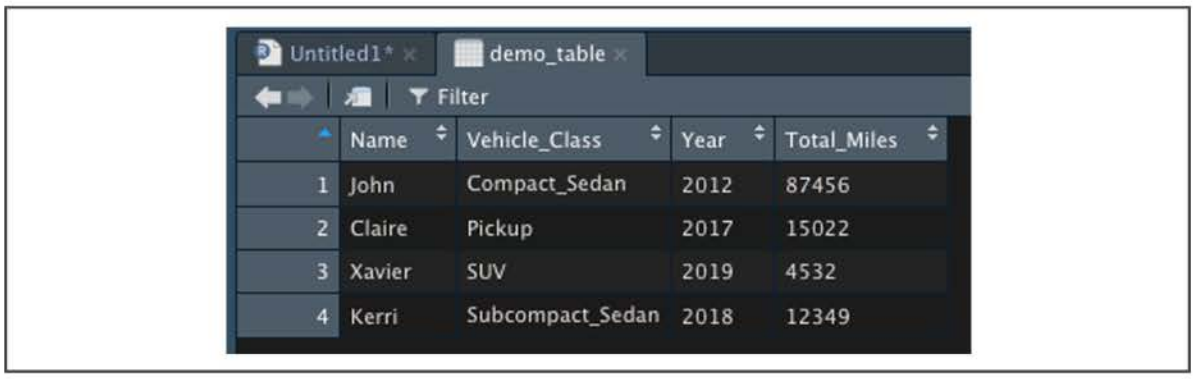
There are many ways to select and subset data in R, depending on what data structure is being used. When it comes to vectors, the easiest way to select data is using the bracket ("[ ]") notation. For example, if we have a numeric vector `x` with 10 values and want to select the third value, we would use the following statements:

```
> x <- c(3, 3, 2, 2, 5, 5, 8, 8, 9)
> x[3]
```



> **Important**
>
> Unlike Python, R's index starts at 1. So, the third element would be `index = 3`.

You can also use bracket notation to select data from two-dimensional data structures, such as matrices, data frames, and tibbles. For example, let's look at our `demo_table` again:

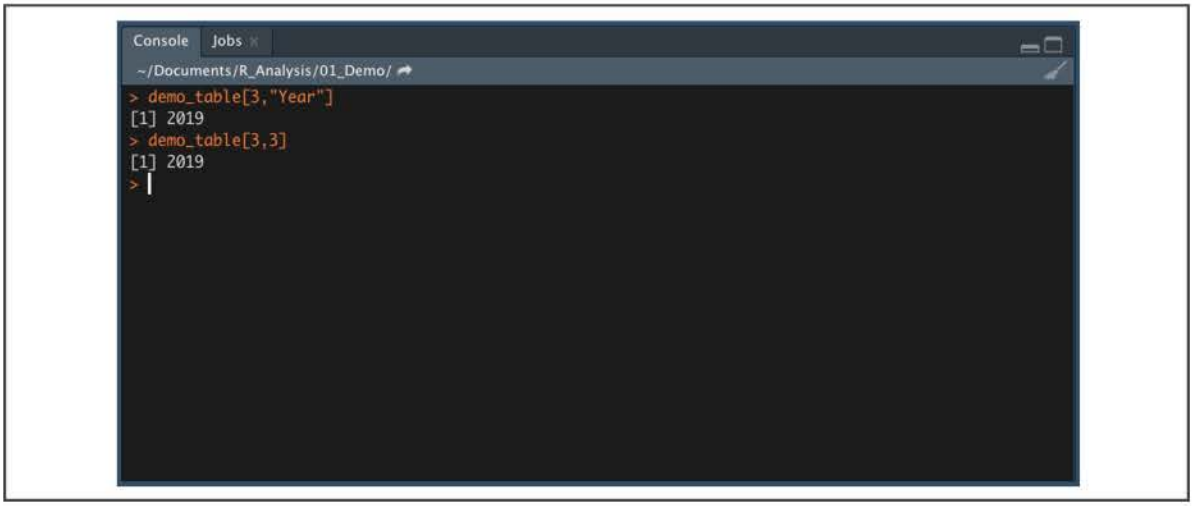


If we want to select the third row of the Year column using bracket notation, our statement would appear as follows:

```
> demo_table[3,"Year"]
```

Because R keeps track of both the row indices as well as the column indices as integers under the hood, we can also select the same data using just number indices:

```
> demo_table[3,3]
```

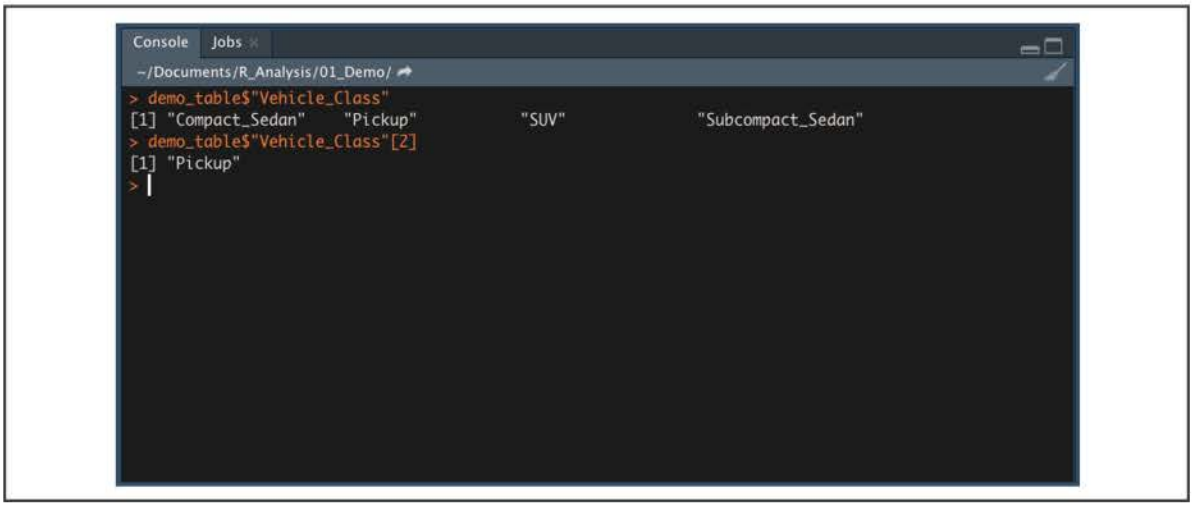


There is a third way to select data from an R data frame that behaves very similarly to Pandas. By using the `$` operator, we can select columns from any two-dimensional R data structure as a single vector, similar to selecting a series from a Pandas DataFrame. For example, if we want to select the vector of vehicle classes from `demo_table`, we would use the following statement:

```
> demo_table$"Vehicle_Class"
```

Once we have selected the single vector, we can use bracket notation to select a single value.

```
> demo_table$"Vehicle_Class"[2]
```



Select Data with Logic

Just as it is for selecting single values, there are multiple ways to subset and filter data from our larger data frames. As with most programming languages, we use a combination of operators and logical statements to tell R what data to filter. Thankfully, most operators are the same between R and Python, as shown below:

Category	R Operator	Description	Python Equivalent
Arithmetical	+	Addition operator	+
	-	Subtraction operator	-
	*	Multiplication operator	*
	/	Division operator	/
	^ or **	Exponent operator	**
	%%	Modulus operator (finds the remainder of the first element divided by the second)	%
Relational	<	Each element	<

	in the first data structure is less than each element in the second data structure.	
<=	Each element in the first data structure is less than or equal to each element in the second data structure.	<=
>	Each element in the first data structure is greater than each element in the second data structure.	>
>=	Each element in the first data structure is greater than or equal to each element in the second data structure.	>=
==	Each element in the first data structure is	numpy.equal()

		equal to each element in the second data structure.	
	!=	Each element in the first data structure is unequal to each element in the second data structure.	numpy.not_equal()
Logical	x|y	Element-wise OR operator—each element of *x* and *y* structures are combined and returns TRUE if either element is TRUE.	numpy.array(x) | numpy.array(y)
	x&y	Element-wise AND operator—each element of *x* and *y* structures are combined and returns TRUE if both elements are TRUE.	numpy.array(x) & numpy.array(y)
	x||y	Logical OR operator—the	x[0] or y[0]

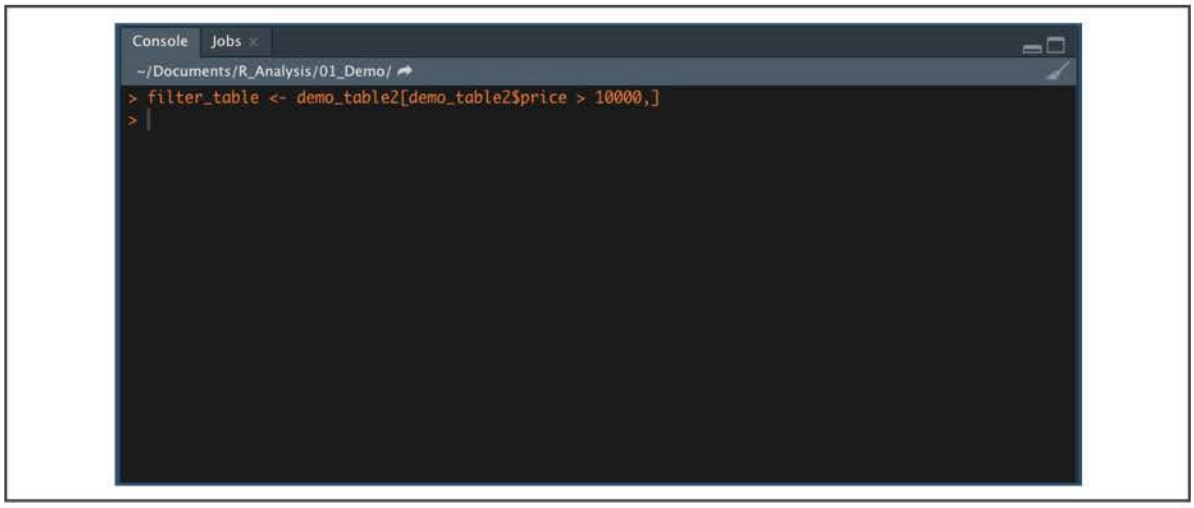
		first element of *x* and *y* structures are combined and returns TRUE if either element is TRUE.	
	x&&y	Logical AND operator—the first element of *x* and *y* structures are combined and returns TRUE if either element is TRUE.	x[0] and y[0]
Miscellany	isTRUE(x)	Checks if the logic *x* is TRUE, otherwise FALSE.	if x:
	x %in% y	Checks if *x* is contained within *y*.	x in y
	x:y	Creates a range of integer values from *x* to *y*.	range(x,y)

One of the most common ways to filter and subset a dataset in R is to use bracket notation. To use bracket notation to filter a data frame, we can supply a logical

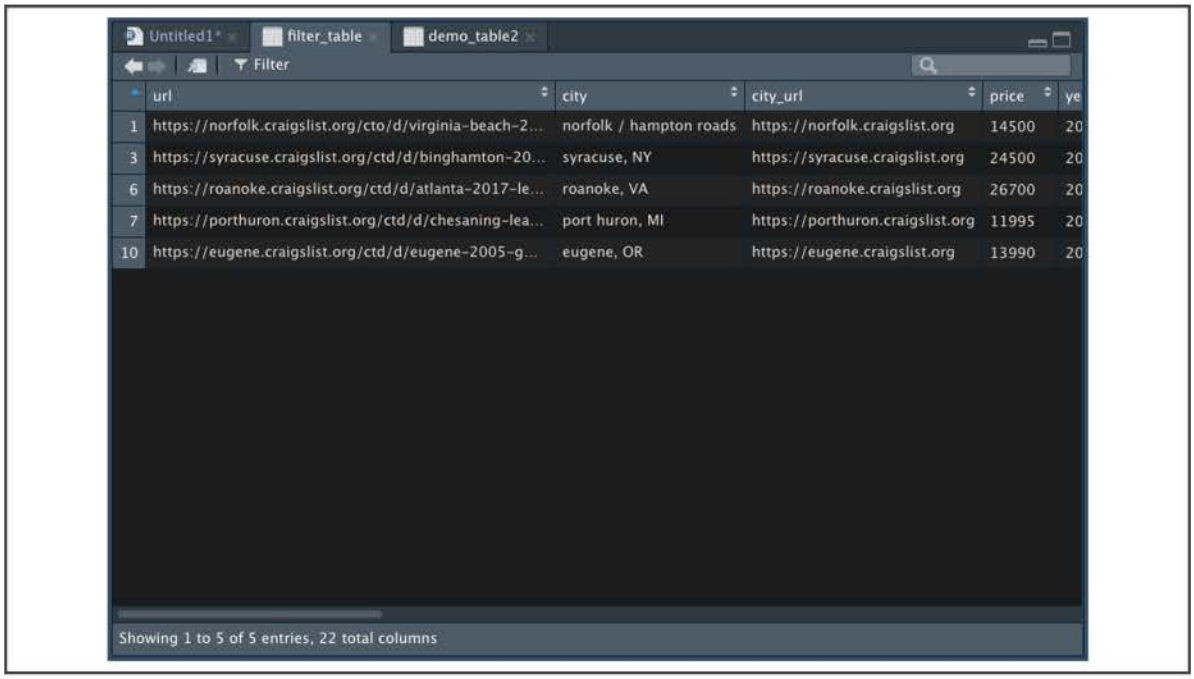
statement to assert our row and columns.

For example, if we want to filter our used car data `demo_table2` so that we only have rows where the vehicle price is greater than $10,000, we would use the following statement:

```
> filter_table <- demo_table2[demo_table2$price > 10000,]
```



This filter statement generates a view-only data frame tab listing vehicles priced greater than $10,000, as shown in the following image:



NOTE

> If you do not supply a logical statement to either rows or columns, R will default to returning all elements in that dimension.

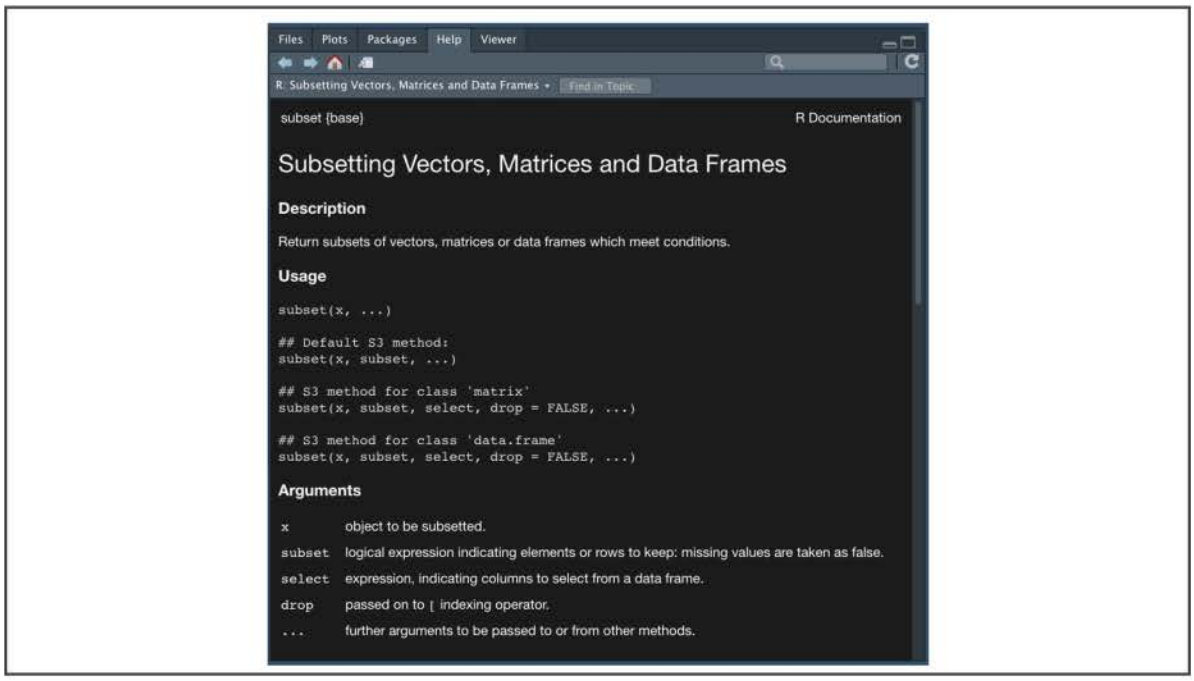
In this example, the `demo_table2$price > 10000` logical statement creates a vector of TRUE/FALSE values that R uses to consider all rows that satisfy our logical statement.

When our logical statements are simple (using only one or two operators), bracket notation is easy to read and write. However, if we need to filter and subset our data using more complicated logic, bracket notation can become cumbersome. In these cases, we'll use an R function such as `subset()` to filter our data.

Subset Data in R

Another method to filter and subset data frames in R is to use the function `subset()`. Type the following code into the R console to look at the `subset()` documentation in the Help pane:

```
> ?subset()
```



The `subset()` function uses a few arguments to subset and filter a two-dimensional R data structure:

- **x**
- **subset**
- **select**

Looking at the R documentation for the `subset()` function, fill in the name of the argument that corresponds to each of the following descriptions:

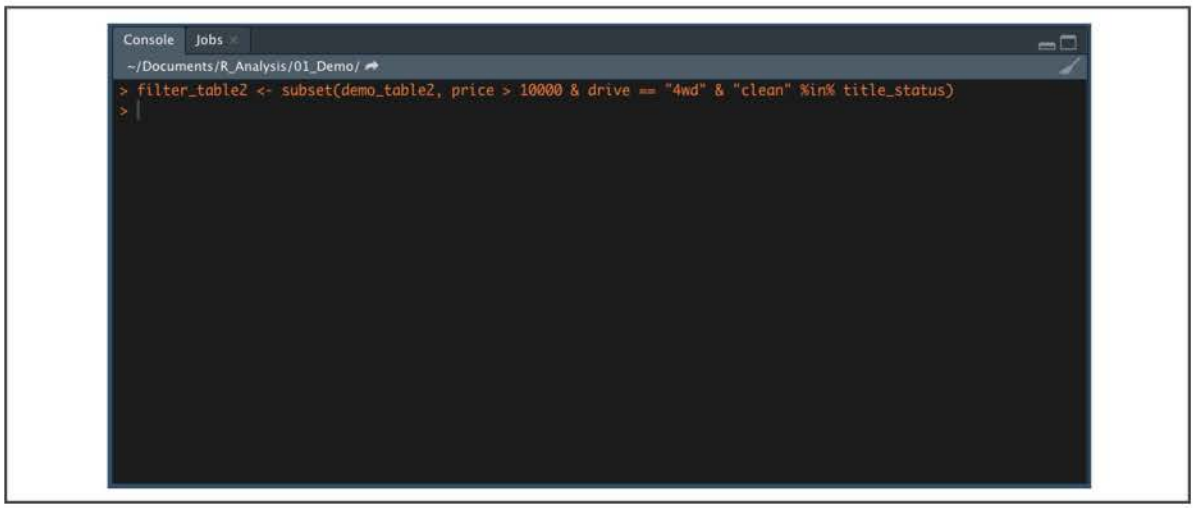
1. [ ] is the matrix, data frame, or tibble we wish to subset.
2. [ ] contains the logical statements that determine which rows to keep.
3. [ ] contains a logical statement, or list of column names to select from a data frame. If nothing is supplied, all columns will be returned.

Check Answer

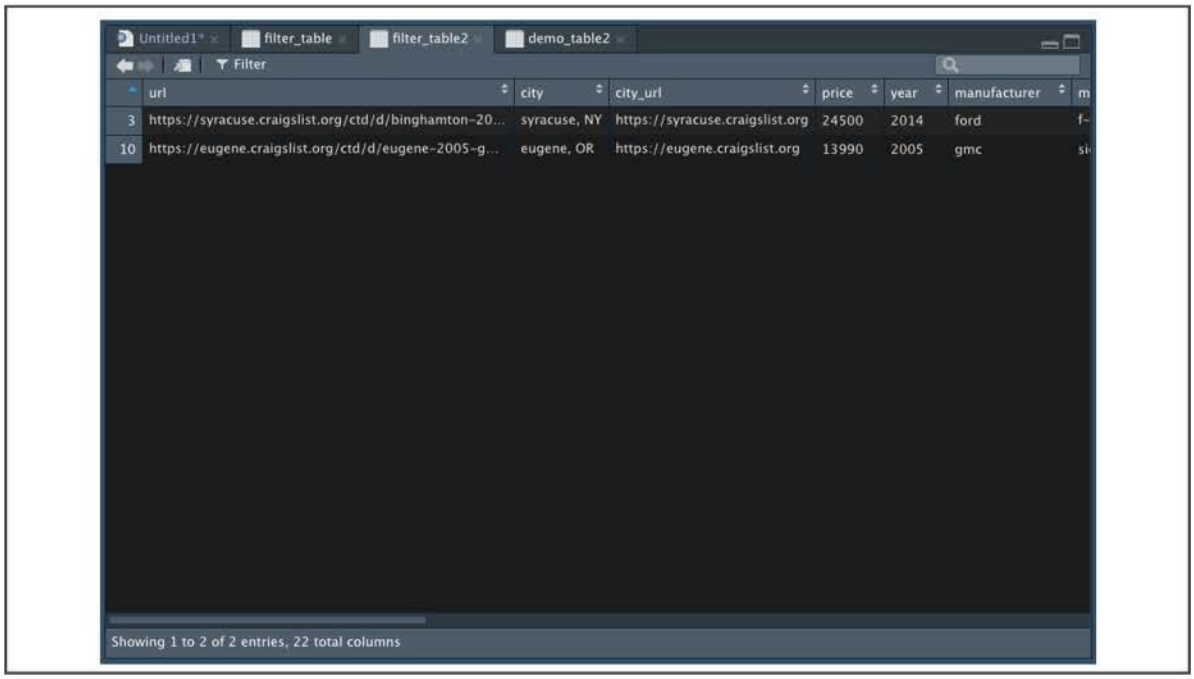
Finish ▸

For example, if we want to create a more elaborate filtered dataset from our used car data `demo_table2` where `price > 10000`, `drive == 4wd`, and `"clean" %in% title_status`, we would use the following statement:

```
> filter_table2 <- subset(demo_table2, price > 10000 & drive == "4wd" & '
```



Here's what the filtered data frame will look like:



> **IMPORTANT**
> When combining logical statements in R, use element-wise AND operator or element-wise OR operator.

The `subset()` function makes filtering and subsetting easier to read by assuming column names in the **subset** argument, which cuts down on statement length. In almost all cases, the bracket notation and `subset()` function are functionally equivalent (especially when using logical statements) and interchangeable.

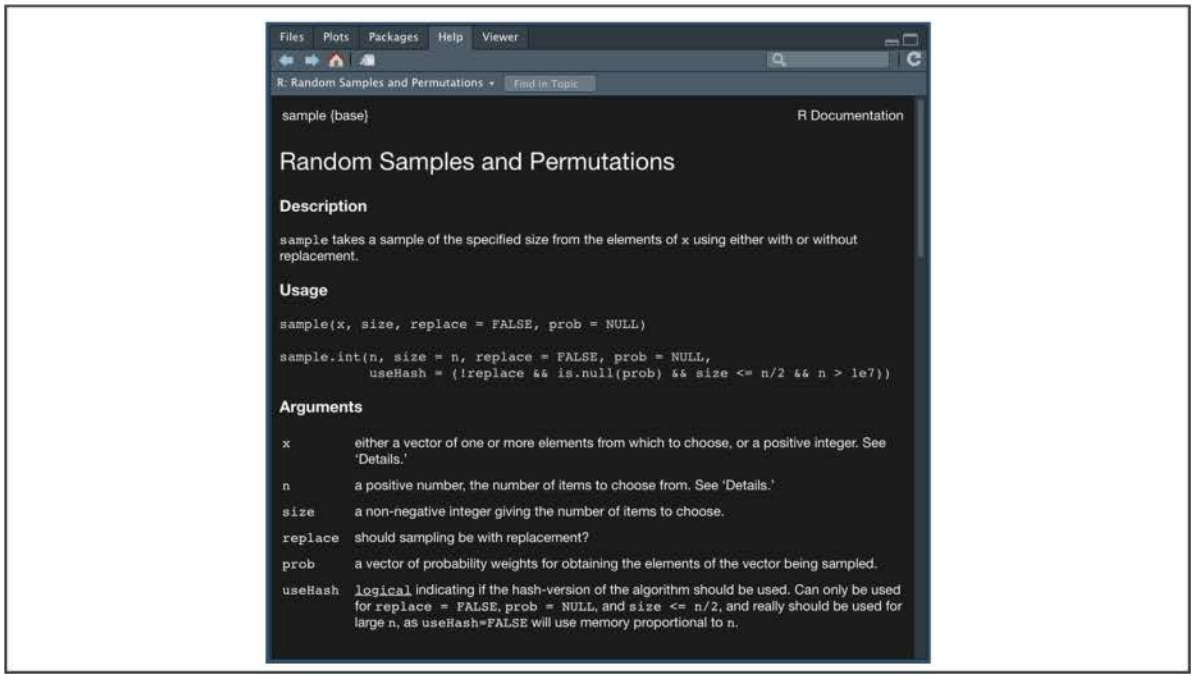
Sample Data in R

Often in data science, we need to generate a random sample of data points from a larger dataset. For example, some models might take too long to run on a massive dataset and require a smaller sample of the data.

Using filtering and subsetting methods may be appropriate for certain cases (such as looking at data within a specific timeframe), but usually we'll want to randomly sample our larger data to reduce bias. In these cases, we can use the built-in function `sample()`. Let's try it now.

Type the following code into the R console to look at the `sample()` documentation in the Help pane:

```
> ?sample()
```



The `sample()` function uses a few arguments to create a sampled vector from a larger vector:

- **x**
- **size**

- **replace**

Looking at the R documentation for the `sample()` function, fill in the name of the argument that corresponds to each of the following descriptions:

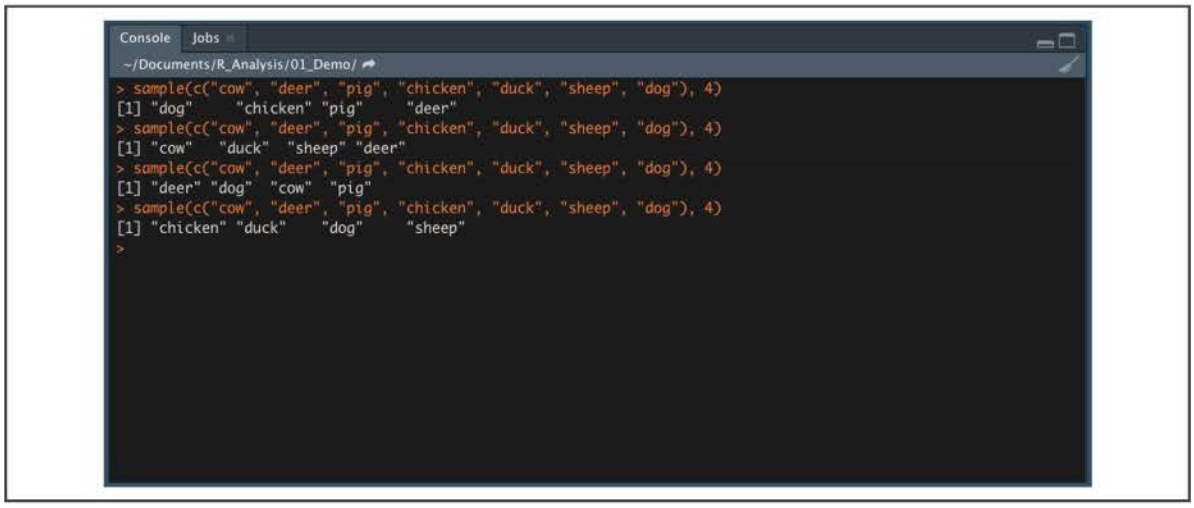
1. [ ] is the larger vector to select from.
2. [ ] is the number of sample data points to select from x.
3. [ ] is a flag to tell whether or not it is okay to select the same values. By default, it is set to FALSE, which means each selected value is unique.

Check Answer

Finish ▸

If we want to sample a large vector and create a smaller vector, we can set the vector to "x":

```
> sample(c("cow", "deer", "pig", "chicken", "duck", "sheep", "dog"), 4)
```

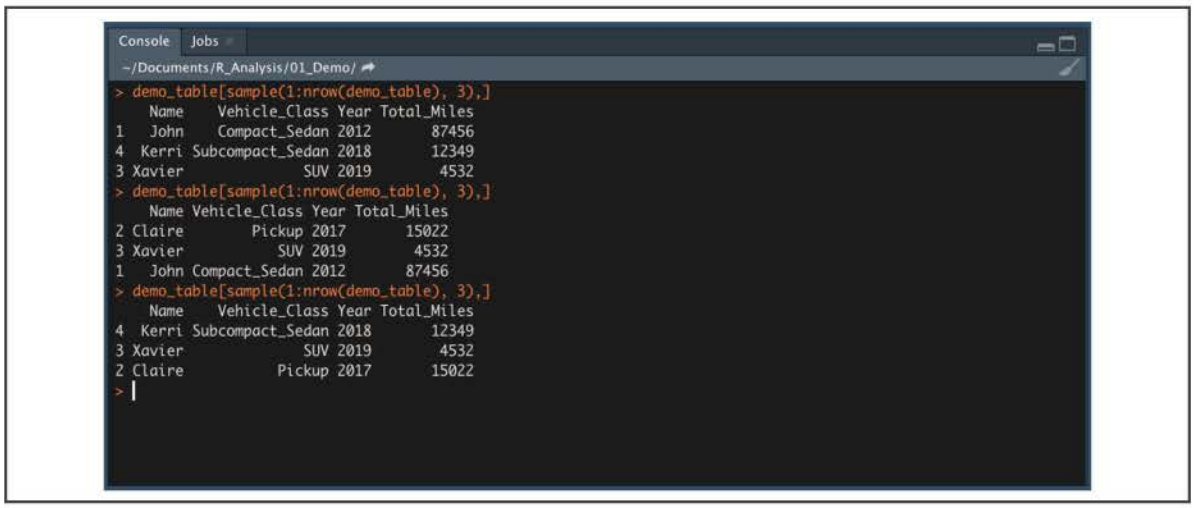


When it comes to sampling a two-dimensional data structure, we need to supply the index of each row we want to sample. This process can be completed in three steps:

1. Create a numerical vector that is the same length as the number of rows in the data frame using the colon (:) operator.
2. Use the `sample()` function to sample a list of indices from our first vector.
3. Use bracket notation to retrieve data frame rows from sample list.

If we want to combine all three steps in a single R statement, our code would be as follows:

```
> demo_table[sample(1:nrow(demo_table), 3),]
```



After we have successfully loaded in and selected our data, our next steps in analysis are to group, transform, and reshape our data as to prepare for visualizations and modeling.

15.2.5: Transform, Group, and Reshape Data Using the Tidyverse Package

Jeremy is really starting to get the hang of this R language! Colleen, Jeremy's go-to source of advice for all things R, suggests he explore the tidyverse package of libraries to help him transform, group, and reshape his data.

One of the most successful optimized packages for R is the **tidyverse (https://www.tidyverse.org/)** package. The tidyverse package contains libraries such as dplyr, tidyr, and ggplot2. These packages work together to help simplify the process of creating transformed data columns, grouping data using factors, reshaping our two-dimensional data structures, and visualizing our results using plots.

Throughout this module and beyond, you'll find more and more uses for the tidyverse libraries as you become more comfortable with R and begin generating more complex and robust RScripts. For now, we'll concentrate on leveraging the tidyverse libraries to help us transform, group, and reshape our R data frames.

Transform

Raw data is often insufficient in telling the full story. Usually when we are analyzing data, we want to perform calculations and incorporate the calculations

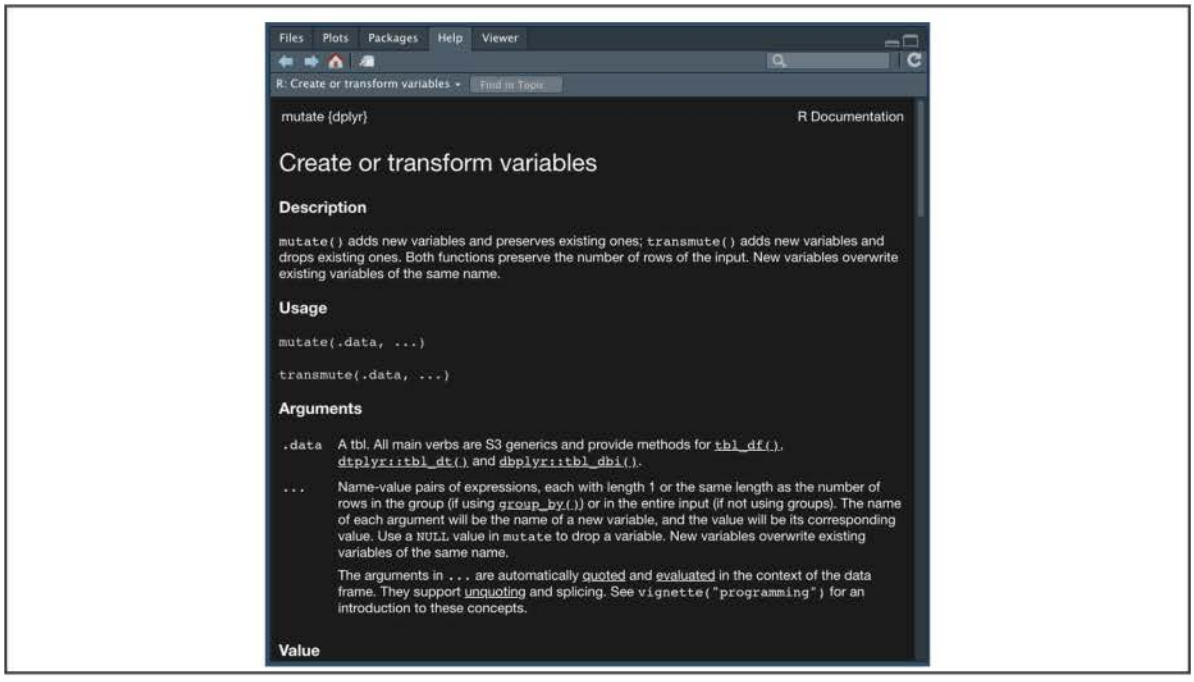
back into the raw data to ease in downstream analysis. Tidyverse's **dplyr (https://dplyr.tidyverse.org/)** library transforms R data.

The dplyr library contains a wide variety of functions that can be chained together to transform data quickly and easily. Using dplyr is slightly more complex than the simple assignment statement in R because dplyr allows the user to chain together functions in a single statement using their own pipe operator (`%>%`).

By chaining functions together, the user does not need to assign intermediate vectors and tables. Instead, all of the data transformation can be performed in a single assignment function that is easy to read and interpret. To transform a data frame and include new calculated data columns, we'll use the `mutate()` function.

Type the following code into the R console to look at the `mutate()` documentation in the Help pane:

```
> library(tidyverse)
> ?mutate()
```



The documentation for the `mutate()` function (generally any dplyr function) is a little obscure, but it makes more sense looking at the examples. We can think of the `mutate()` function as a series of smaller assignment statements that are

separated by commas. Each of the assigned names appears as a new column in our raw data frame.

For example, if we want to use our coworker vehicle data from the `demo_table` and add a column for the mileage per year, as well as label all vehicles as active, we would use the following statement:

```
> demo_table <- demo_table %>% mutate(Mileage_per_Year=Total_Miles/(2020-
```

	Name	Vehicle_Class	Year	Total_Miles	Mileage_per_Year	IsActive
1	John	Compact_Sedan	2012	87456	10932.000	TRUE
2	Claire	Pickup	2017	15022	5007.333	TRUE
3	Xavier	SUV	2019	4532	4532.000	TRUE
4	Kerri	Subcompact_Sedan	2018	12349	6174.500	TRUE

For each name and expression, a new column is generated within our returned data frame. This returned data frame is then assigned to our original data structure `demo_table`.

This process of mutating and transforming columns can be applied at any time (or even multiple times) within a dplyr pipe, and it can even reference columns that were generated from previous `mutate()` functions. These transformed data frames can be used in further downstream analysis, visualizations, and modeling. However, transformed data frames are not designed to summarize data. To summarize our data frames in R, we'll need to use a grouping function.

Group Data

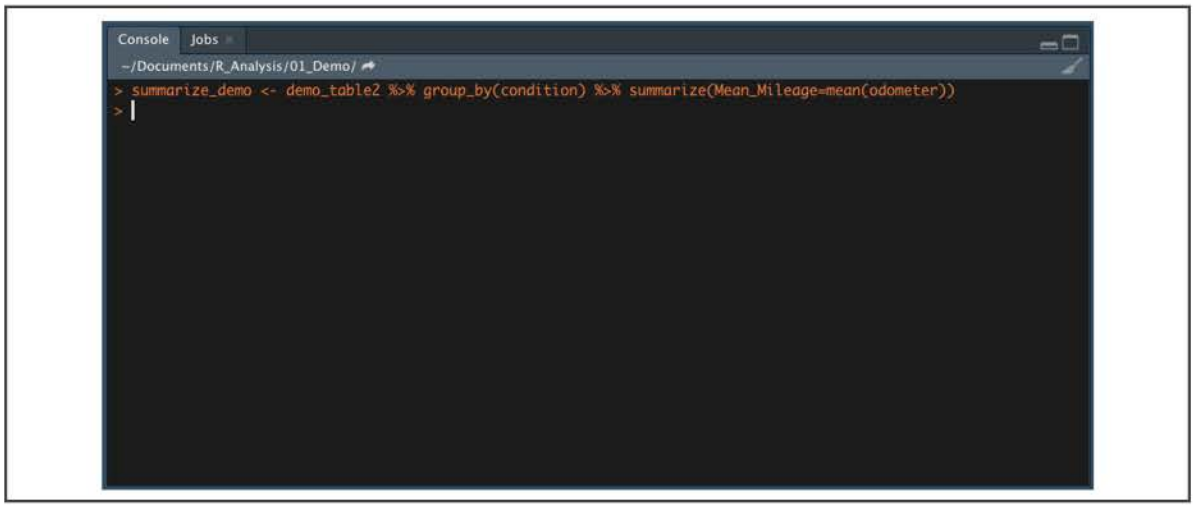
Just like in Python, grouping across a factor allows us to quickly summarize and characterize our dataset around a factor of interest (also known as a character vector in R, or list of strings in Python). The most straightforward way to perform a

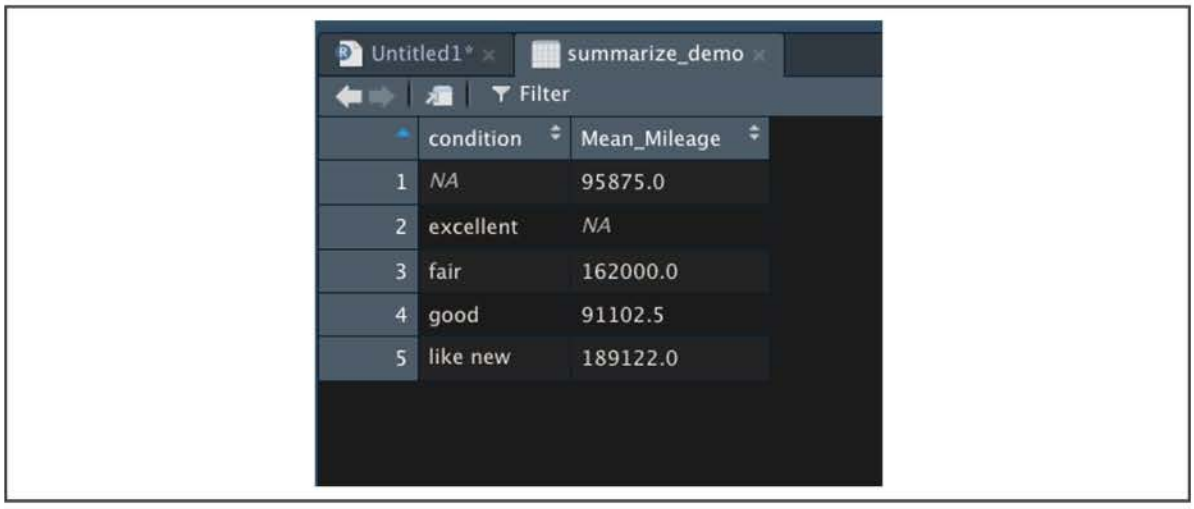
grouping on an R data frame is to use dplyr's `group_by()` function. The `group_by()` function tells dplyr which factor (or list of factors in order) to group our data frame by.

The behavior of `group_by()` is almost identical to that of the Pandas `DataFrame.group_by()` function, with the exception of using a value or vector instead of a list. Once we have our `group_by` data structure, we use the dplyr `summarize()` function to create columns in our summary data frame.

For example, if we want to group our used car data by the condition of the vehicle and determine the average mileage per condition, we would use the following dplyr statement:

```
> summarize_demo <- demo_table2 %>% group_by(condition) %>% summarize(Mea
```



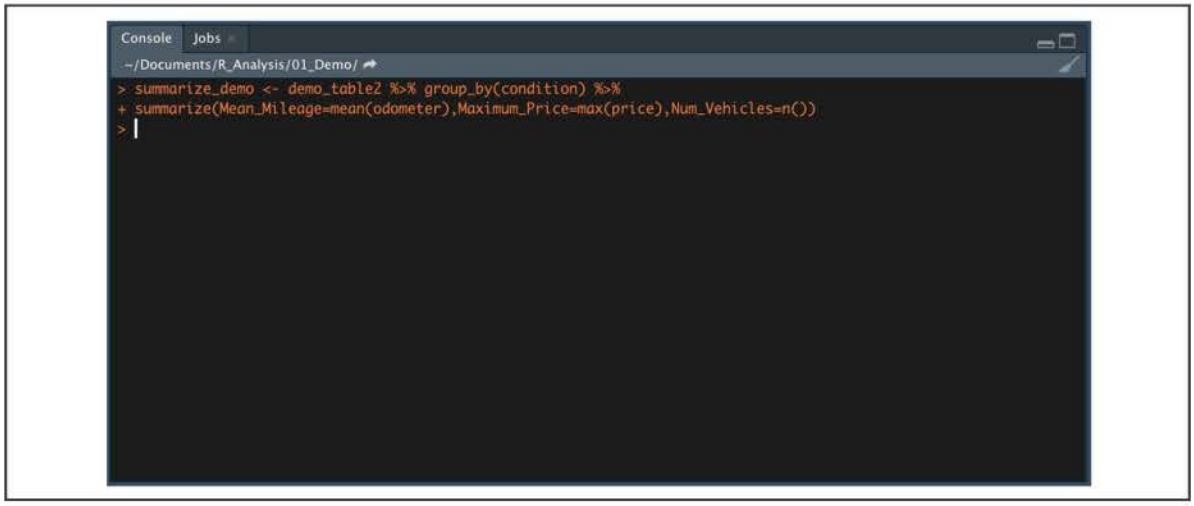


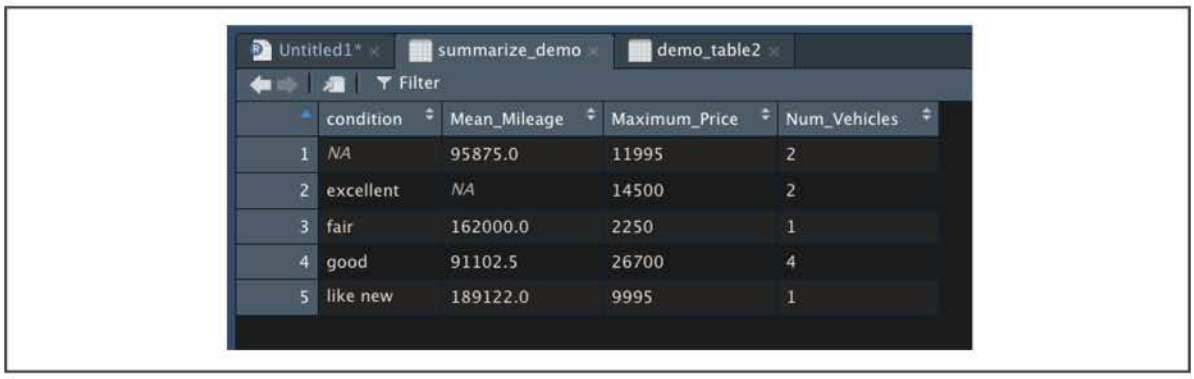
> **Note**
>
> Dplyr's pipe operator allows us to move our connected functions to a new line while computing everything in the same assignment function. This allows you to write RScripts that are not too wide for your RScript window.

Using the `summarize()` function is very similar to using the `mutate()` function on a raw data frame: for each name and summary function, we create a column in a summary data frame. Our simplest summary functions will use statistics summary functions such as `mean()`, `median()`, `sd()`, `min()`, `max()`, and `n()` (used to calculate the number of rows in each category).

However, the dplyr `summarize()` documentation provides a more comprehensive list of functions that can be used to summarize our data. For example, if in addition to our previous summary table we wanted to add the maximum price for each condition, as well as add the vehicles in each category, our statement would look as follows:

```
> summarize_demo <- demo_table2 %>% group_by(condition) %>% summarize(Mea
```





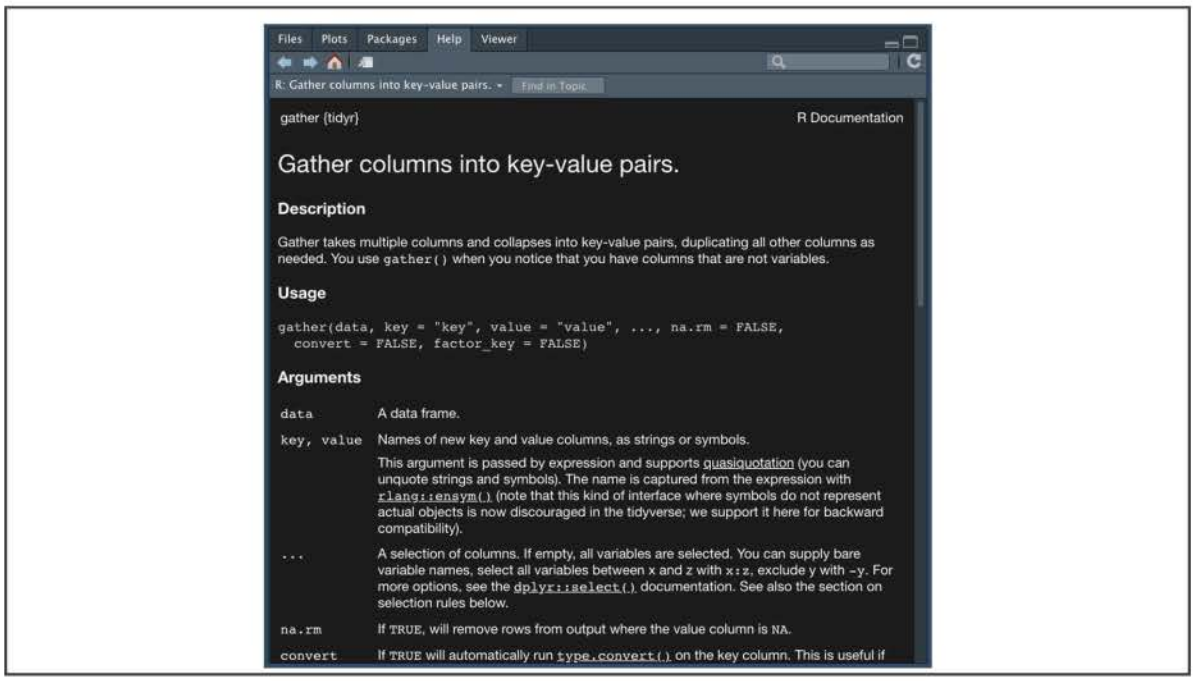
Reshape Data

When performing more involved data analytics and visualizations, there may be situations where the shape and design of our data frame is overcomplicated or incompatible with the libraries and functions we wish to use. For example, a "wide" data frame with few rows and many columns and factors may be difficult to visualize. Or a "long" data frame may be difficult to analyze if it contains multiple rows for a single subject.

Thankfully, the tidyr library from the tidyverse has the `gather()` and `spread()` functions to help reshape our data. The `gather()` function is used to transform a wide dataset into a long dataset.

Type the following code into the R console to look at the `gather()` documentation in the Help pane:

```
> ?gather()
```



To properly reshape an R data frame, the `gather()` function requires a few arguments:

- **data**
- **key**
- **value**
- **...**

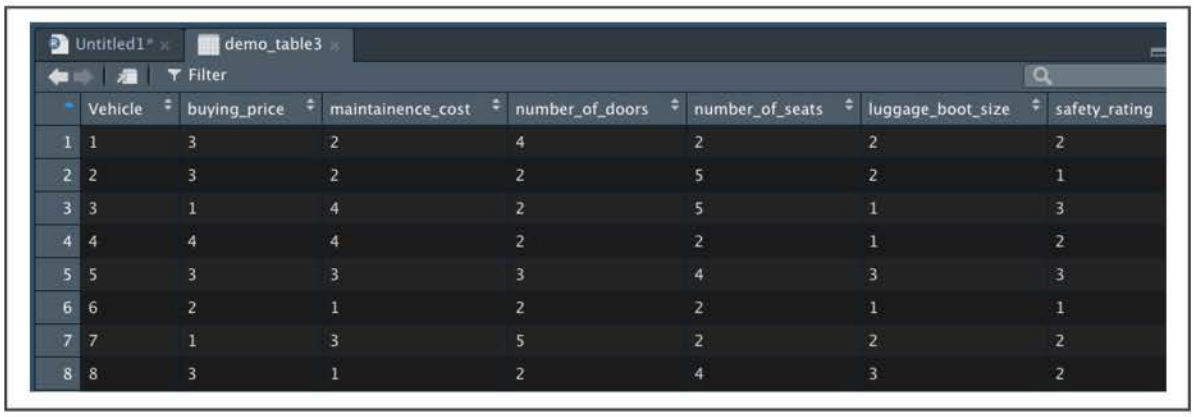
Looking at the R documentation for the `gather()` function, fill in the name of the argument that corresponds to each of the following descriptions:

1. [ ] is the data frame we wish to reshape. Instead of supplying the data frame object, we can alternatively use the pipe operator to create a dplyr/tidyr all-in-one statement.
2. [ ] is the name of the variable column that the original, wider data frame will collapse into.
3. [ ] is the name of the new value column derived from the original data frame results.
4. [ ] represents a list of columns to collapse into the key column. If we do not supply a list of columns, all columns of the original data frame will be collapsed. If the columns are in order, we can use the colon (:) operator to create our list of columns to collapse.

One of the easiest ways to learn how to reshape data is to practice with a simple dataset.

For this example, first **download the demo2.csv (https://courses.bootcampspot.com/courses/138/files/18463/download?wrap=1)**  **(https://courses.bootcampspot.com/courses/138/files/18463/download?wrap=1)** file, and put it in your active directory. Once you have put the file in your active directory, load the `demo2.csv` file into your R environment and look at the top of the data frame:

```
> demo_table3 <- read.csv('demo2.csv',check.names = F,stringsAsFactors =
```



Untitled1* | demo_table3

Filter

	Vehicle	buying_price	maintainence_cost	number_of_doors	number_of_seats	luggage_boot_size	safety_rating
1	1	3	2	4	2	2	2
2	2	3	2	2	5	2	1
3	3	1	4	2	5	1	3
4	4	4	4	2	2	1	2
5	5	3	3	3	4	3	3
6	6	2	1	2	2	1	1
7	7	1	3	5	2	2	2
8	8	3	1	2	4	3	2

For the following checkpoint **download Vehicle_Data.xlsx (https://courses.bootcampspot.com/courses/138/files/18486/download?wrap=1)**  **(https://courses.bootcampspot.com/courses/138/files/18486/download?wrap=1)** .

Using the file *Vehicle_data.xlsx*, answer the following questions.

	Long	Wide
1. Is the data in the “Vehicle Prices” tab in a long or wide format?	○	○
2. Is the data in the “Volkswagen 2020 Lineup” tab in a long or wide format?	○	○

Check Answer

Finish ▸

The `demo_table3` data frame contains survey results from 250 vehicles that were collected by a rental company. The rental company evaluated seven different metrics for each vehicle and rated each metric on a scale of 1 to 5, with 1 representing low and 5 representing high.

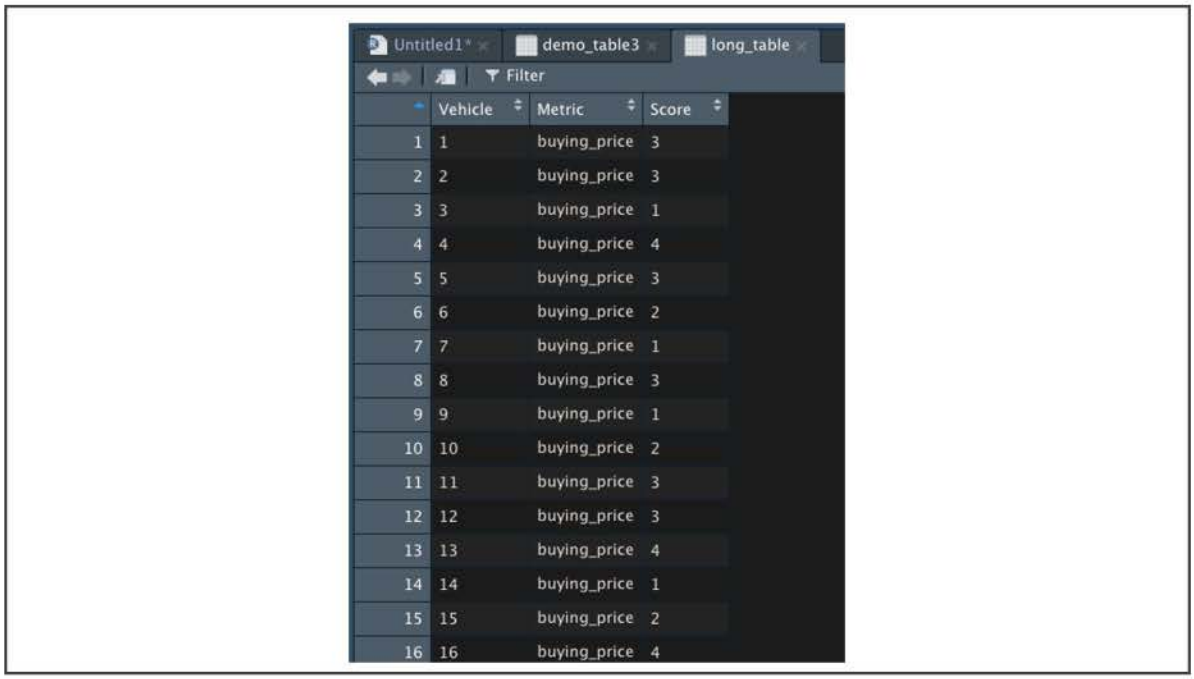
This data would be considered a wide format because different metrics were collected from a single vehicle, and each metric (also known as a variable in this case) was stored as a separate column. To change this dataset to a long format, we would use `gather()` to reshape this dataset.

Type the following function into the R console:

```
> long_table <- gather(demo_table3,key="Metric",value="Score",buying_pric
```

Alternatively, you may type the following function in the R console:

```
> long_table <- demo_table3 %>% gather(key="Metric",value="Score",buying_
```

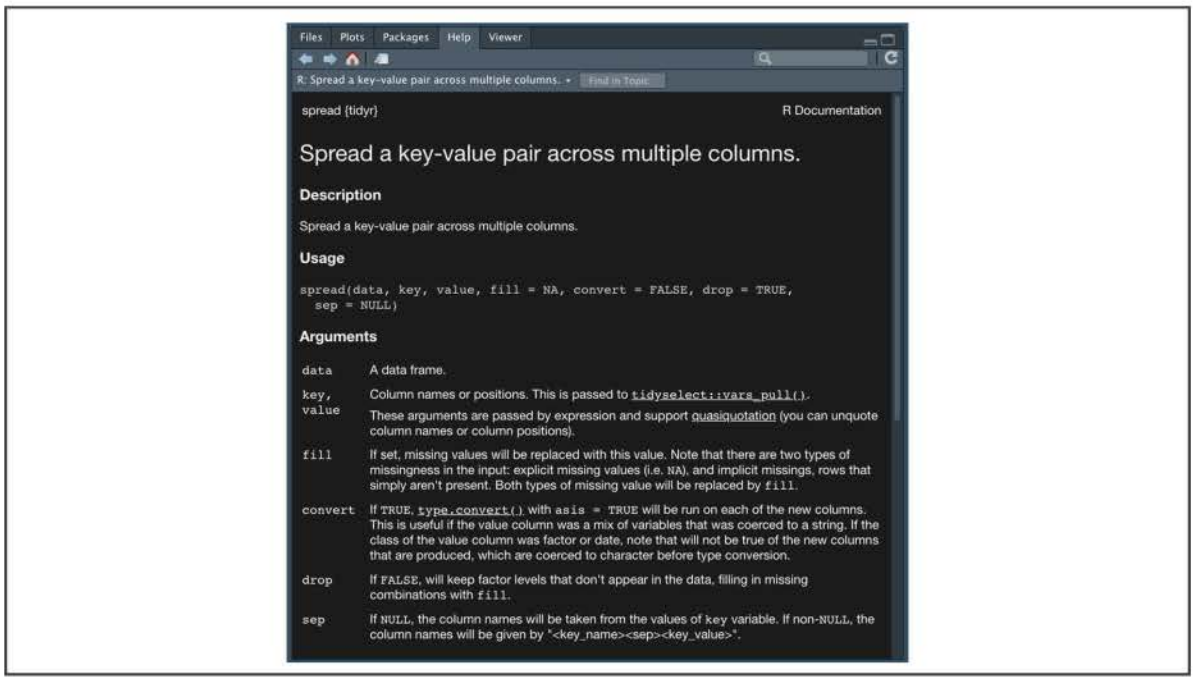


By using the `gather()` function, we have collapsed all of the survey metrics into one Metric column and all of the values into a Score column. Because the Vehicle column was not in the arguments, it was treated as an identifier column and added to each row as a unique identifier.

Alternatively, if we have data that was collected or obtained in a long format, we can use tidyr's `spread()` function to spread out a variable column of multiple measurements into columns for each variable.

Type the following code into the R console to look at the `spread()` documentation in the Help pane:

```
> ?spread()
```



To properly reshape an R data frame, the `spread()` function requires a few arguments:

- **data**
- **key**
- **value**
- **fill**

Looking at the R documentation for the `spread()` function, fill in the name of the argument that corresponds to each of the following descriptions:

1. [ ] is the name of the variable column that we wish to spread out.

2. [ ] is the data frame we wish to reshape. Instead of supplying the data frame object, we can alternatively use the pipe operator to create a dplyr/tidyr all-in-one statement.

3. [ ] is an optional argument that will set any empty rows in a new variable column with the fill value.

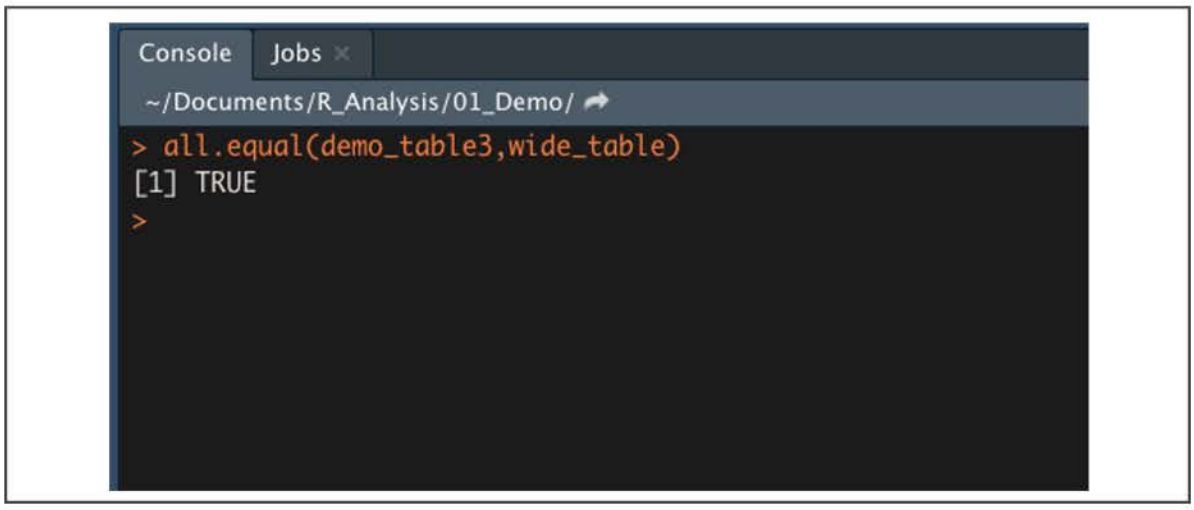
4. [ ] is the name of the value column that we wish to fill in our new variable columns with.

Check Answer

Therefore, if we want to spread out our previous long-format data frame back to its original format, we would use the following `spread()` statement:

```
> wide_table <- long_table %>% spread(key="Metric",value="Score")
```

And if we want to check if our newly created wide-format table is exactly the same as our original `demo_table3`, we can use R's `all.equal()` function:



IMPORTANT

If you ever compare two data frames that you expect to be equal, and the `all.equal()` function tells you they're not, try sorting the columns of both data frames. You can sort columns using the `order()` and `colnames()` functions and bracket notation:

```
> table <- table[,order(colnames(table))]
```

Now that we have learned about selecting and manipulating our data in R, it's time to learn how to use R for data analysis. We'll begin by visualizing our datasets using ggplot2.

15.3.1: Introduction to ggplot2

After taking some time to learn about loading, selecting, and reshaping data in R, Jeremy is eager to move forward. He wants to start producing some reusable code and contributing to the team's data analysis repository. Although he isn't quite ready to generate robust statistical analysis scripts for the executive team, Jeremy knows that there are plenty of smaller scripts he can generate to produce visualizations that can be reused for reports and presentations.

In the data world, there are few analytical and graphical tools more popular and recognizable than R's **ggplot2** (https://ggplot2.tidyverse.org/) library. Due to its adoption among research groups and government agencies, ggplot2 figures and visualizations can be found in almost every scientific paper written from 2005 to the present. ggplot2 is a favorite of many R and non-R users alike because it is easy to implement, read, and interpret, yet capable of performing a deep and complex analysis.

In this section, we'll learn about the ggplot library components and how to implement our basic plots, such as bar, line, and scatter plots. Once we have mastered the basics of plotting in ggplot2, we'll learn how to plot more advanced visualizations such as boxplots and heatmaps.

All figures in ggplot2 are created using the same three components:

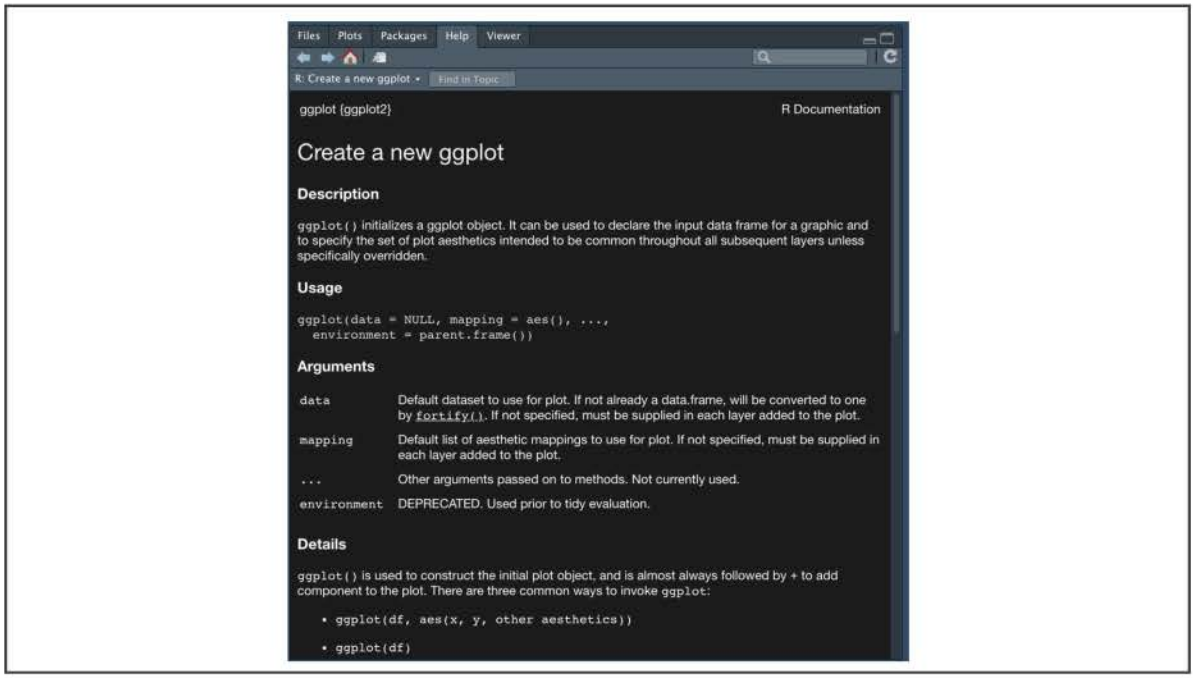
1. `ggplot` **function**—tells ggplot2 what variables to use

2. `geom` **function**—tells ggplot2 what plots to generate
3. **formatting or theme functions**—tells ggplot2 how to customize the plot

Similar to how we generate figures using Python's Matplotlib library, we build our ggplot2 visualizations by layering multiple plots and adding customized colors, scales, and labels to represent our data effectively. Also, similar to Matplotlib, the ggplot2 documentation is very clear with examples for each option and function, which makes ggplot2 far more approachable than many programming languages.

Before we start to build our basic figures, we need to declare our input data and variables using the `ggplot()` function. Type the following code into the R console to look at the `ggplot()` documentation in the Help pane:

```
> ?ggplot()
```



The `ggplot()` function only requires two arguments to declare the input data:

- **data**
- **mapping**

Looking at the R documentation for the `ggplot()` function, fill in the name of the argument that corresponds to each of the following descriptions:

1. _____ is our input data frame.
2. _____ uses the aes() aesthetic function to tell ggplot() what variables are assigned to the x (independent) and y (dependent) variables.

Check Answer

Finish ▸

IMPORTANT

There are a number of optional `aes()` arguments to assign such as color, fill, shape, and size to customize the plots. We'll cover these optional assignments in this module.

The return value of the `ggplot()` function is our `ggplot` object, which is used as the base to build our visualizations. Once we have established a base `ggplot` object, we can add any number of plotting and formatting functions using an addition (+) operator.

15.3.2: Build a Bar Plot in ggplot2

For his first plots, Jeremy has decided to visualize some information about fuel economy. His practice dataset will be from the U.S. Environmental Protection Agency (EPA) and dated 1999 through 2008. He knows Colleen already has this data on lock, so he'll be able to check his work with her—and it will help them both brainstorm for their team's first big presentation to the CEO.

Now that we are familiar with setting up the `ggplot()` function, let's build our first plot using the mpg (miles per gallon) dataset. First, we'll take a moment to familiarize ourselves with the mpg dataset. In the R console, type the following statement:

```
> head(mpg)
```

The mpg dataset contains fuel economy data from the EPA for vehicles manufactured between 1999 and 2008. The mpg dataset is built into R and is used throughout R documentation due to its availability, diversity of variables, and overall cleanliness of data. For our purposes, we'll use the mpg data to demonstrate how to implement each of our ggplot visualizations.

Which type of plot is primarily used to visualize categorical data, bar plots, or line plots?

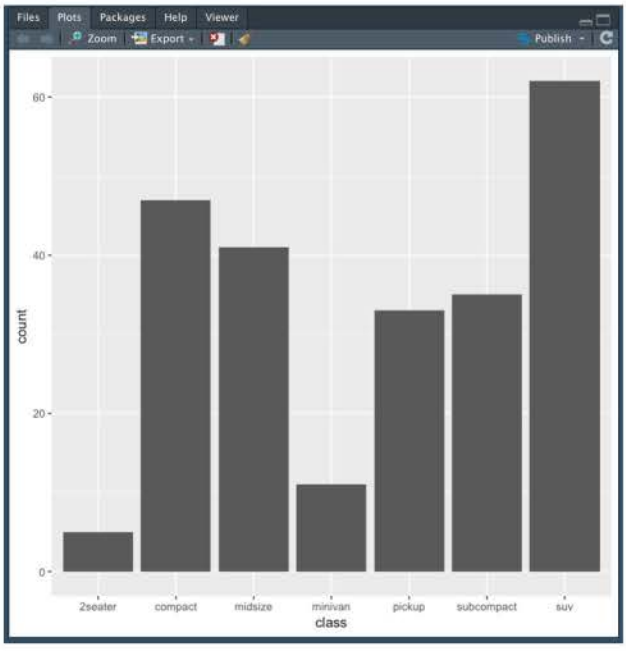
- Bar plot
- Line plot

Check Answer

Finish ▸

The first plots we'll generate using ggplot2 will be bar plots. Bar plots are used to visualize categorical data. They can be used to represent the frequency of each categorical value in a list of categorical data. For example, if we want to create a bar plot that represents the distribution of vehicle classes from the mpg dataset, we would use the following statements in R:

```
> plt <- ggplot(mpg,aes(x=class)) #import dataset into ggplot2
> plt + geom_bar() #plot a bar plot
```



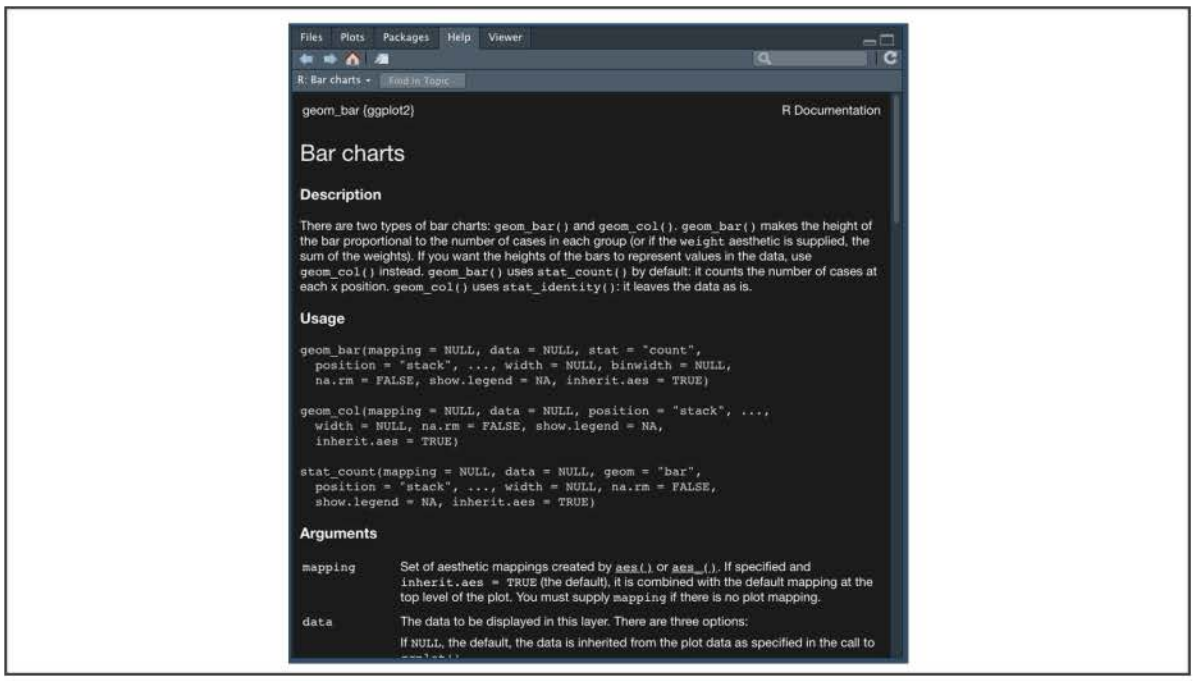
Note

> When you generate a plot in RStudio, the multi-tool pane will switch over to the Plot pane.

In this example, we're only trying to visualize univariate (single variable) data. Therefore, we only need to assign our `x` argument within the `aes()` function. After creating our `ggplot` object, we then generate a bar plot using `geom_bar()`.

Type the following code into the R console to look at the `geom_bar()` documentation in the Help pane:

```
> ?geom_bar()
```

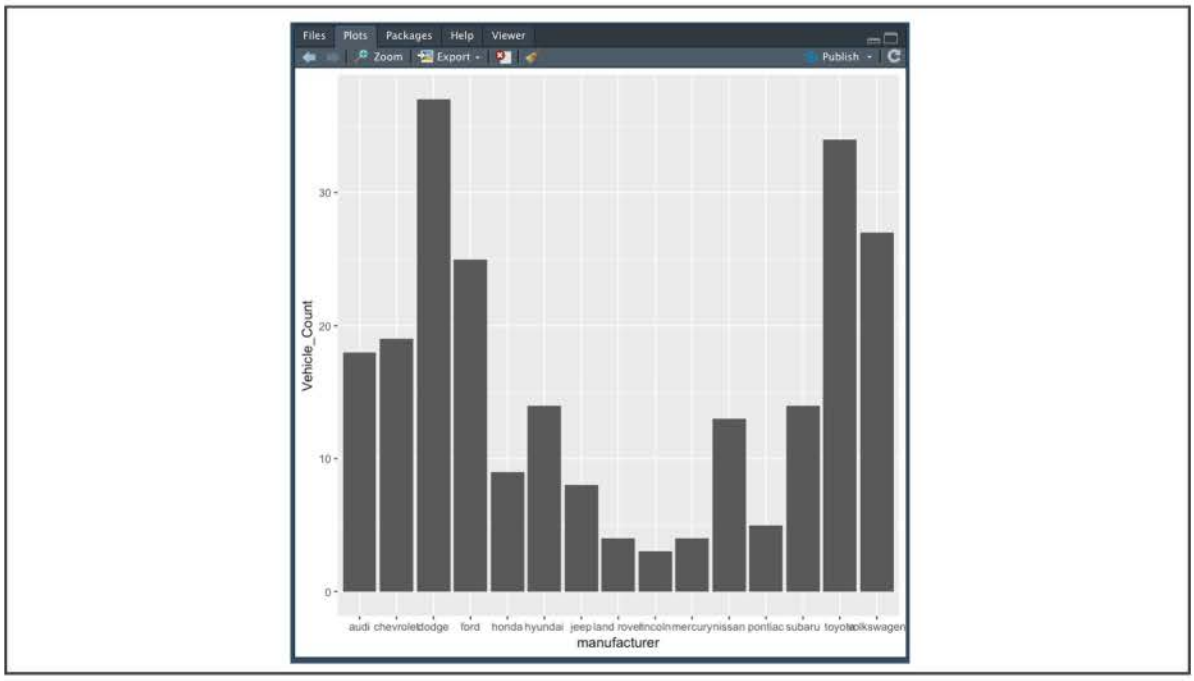


Unlike most of our previous R functions that we have explored, the `geom` functions from ggplot2 are very large. However, in most cases, we can leave all of the arguments alone and use the `geom ()` function by itself.

Another use for bar plots is to compare and contrast categorical results. For example, if we want to compare the number of vehicles from each manufacturer in the dataset, we can use dplyr's `summarize()` function to summarize the data, and ggplot2's `geom_col()` to visualize the results:

```
> mpg_summary <- mpg %>% group_by(manufacturer) %>% summarize(Vehicle_Cou
```

```
> plt <- ggplot(mpg_summary,aes(x=manufacturer,y=Vehicle_Count)) #import
> plt + geom_col() #plot a bar plot
```



As we practiced previously, creating a summary table for the manufacturer vehicles was done using dplyr's `group_by()` and `summarize()` functions. Our new summary table was then used as the input data for our `ggplot()` function.

In our first example, we only needed to assign one variable to our list of classes. In contrast, our second example required two variables—one for our categorical factors (assigned to x), and another for our calculated results (assigned to y). Once we generated our `ggplot` object, we then used an alternative method for creating a bar plot, `geom_col()`.

Functionally, both `geom_bar()` and `geom_col()` create bar plots; however, the two methods assume different inputs. `geom_bar()` expects one variable and generates frequency data, and `geom_col()` expects two variables where we provide the size of each category's bar.

> **Note**
>
> Many of ggplot2 visualizations have alternative methods that accommodate different use cases. Feel free to look at the **ggplot2**

> **documentation** (https://ggplot2.tidyverse.org/reference/index.html) if you have a specific use case in mind.

In its current state, our bar plot could be sufficient for personal use when drawing quick conclusions about the data. For instance, we can see from our bar plot that Dodge had the highest number of vehicles in the dataset and Lincoln had the fewest. However, our current bar plot would not be appropriate to use for an analytical report or for publishing. The two biggest issues with the current plot are:

- Our axis titles are not consistent and could be better formatted.
- Our x-axis labels are overlapping and run off the page.

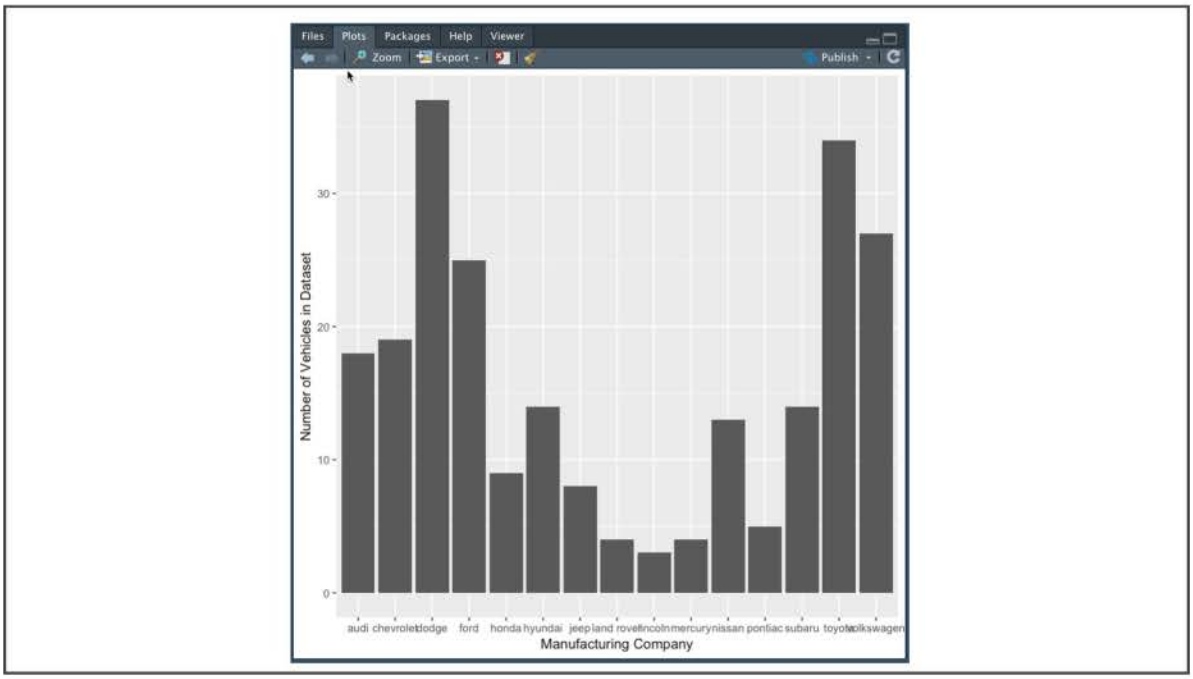
We'll fix this by adding formatting functions.

15.3.3: Add Formatting Functions

Jeremy needs to polish his bar plot before presenting to the CEO. First, he needs to format the axis titles so they're consistent, and then he needs to fix the axis labels so they don't run off the page or overlap. It's time for Jeremy to learn some formatting functions!

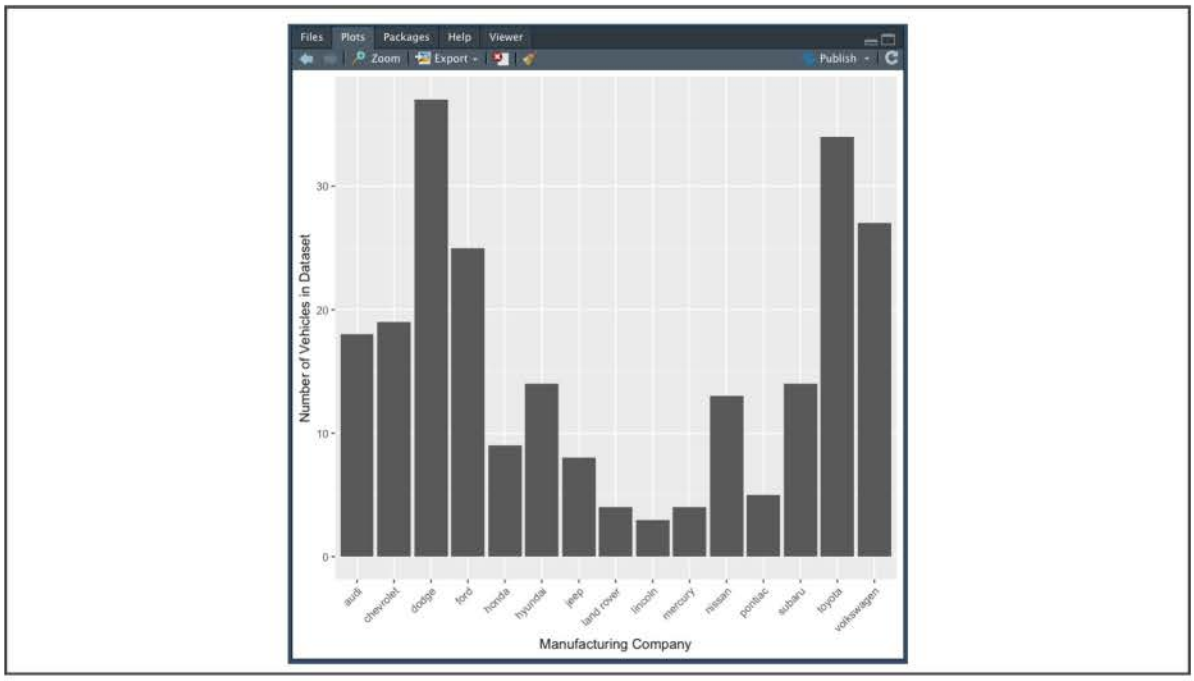
To address the issues with the plot, we'll need to add formatting functions to our plotting statement. To change the titles of our x-axis and y-axis, we can use the `xlab()` and `ylab()` functions, respectively:

```
> plt + geom_col() + xlab("Manufacturing Company") + ylab("Number of Vehi
```



For our figure, rotate the x-axis labels 45 degrees so they no longer overlap. Our new plotting statement would be as follows:

```
> plt + geom_col() + xlab("Manufacturing Company") + ylab("Number of Vehi
>theme(axis.text.x=element_text(angle=45,hjust=1)) #rotate the x-axis lab
```



> **Caution**
>
> Unfortunately, rotating and adjusting the axis labels in ggplot2 is not as straightforward as changing axis titles. Due to the amount of customizability in ggplot2, making small adjustments such as rotating text requires very specific values to be changed in nested functions. Thankfully, there is plenty of **external documentation (http://www.cookbook-r.com/Graphs/Axes_(ggplot2)/)** and **Stack Overflow support (https://stackoverflow.com/questions/1330989/rotating-and-spacing-axis-labels-in-ggplot2)** that addresses these exact use cases, so finding help on how to tweak your ggplot2 visualizations requires only a basic Google search.

In this case, we set the angle argument of our `element_text()` function to 45 degrees and our `hjust` argument to 1. The `hjust` argument tells `ggplot` that our

rotated labels should be aligned horizontally to our tick marks.

Similarly, if we want to adjust our y-axis labels, we would do so by using the `axis.text.y` argument of the `theme()` function. Now that we have adjusted our axis labels and titles, our figure is far easier to read and ready for print. Now it's time to generate our line plots!

15.3.4: Build a Line Plot in ggplot2

Jeremy is pretty comfortable with bar plots, but he knows he will want to show relationships between different types of data for this project—and that a simple bar plot might not cut it.

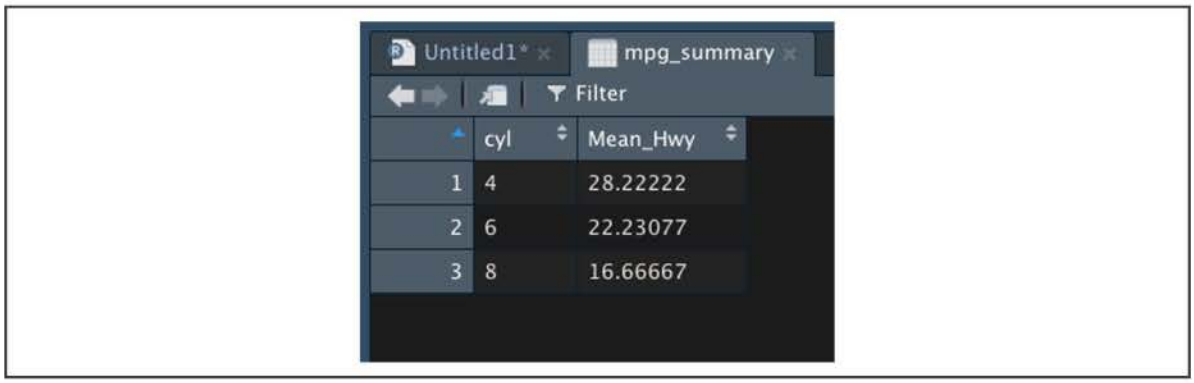
Colleen suggests he dig into line plots—after all, she uses them a lot when she needs to show the relationship between a categorical variable and a continuous variable.

Jeremy heads back to his desk, grateful for such a smart and supportive team. Suddenly, he realizes he didn't ask Colleen *how* line plots demonstrate the relationship between these two types of variables. Not to be deterred, he fires up his computer and gets learning.

Line plots are used to visualize the relationship between a categorical variable and a continuous numerical variable.

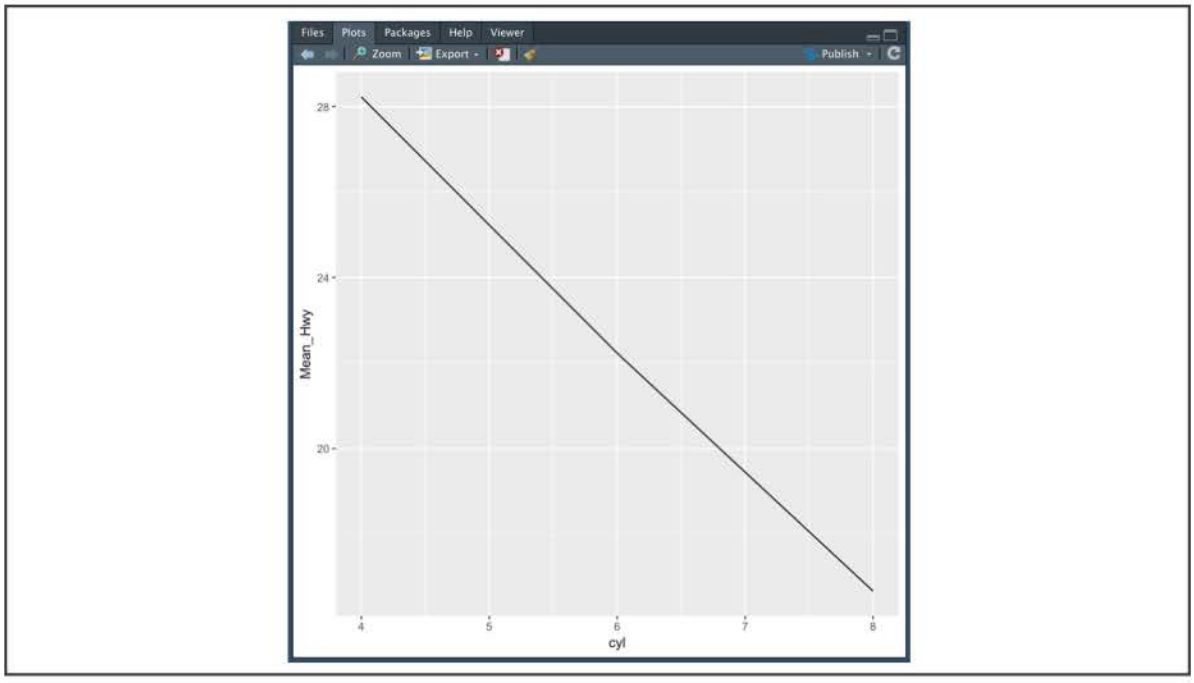
When creating the `ggplot` object for our line data, we need to set the categorical variable to the *x* value and our continuous variable to the *y* value within our `aes()` function. For example, if we want to compare the differences in average highway fuel economy (`hwy`) of Toyota vehicles as a function of the different cylinder sizes (`cyl`), our R code would look like the following:

```
> mpg_summary <- subset(mpg,manufacturer=="toyota") %>% group_by(cyl) %>%
> plt <- ggplot(mpg_summary,aes(x=cyl,y=Mean_Hwy)) #import dataset into g
```



Once we set up our `ggplot` object, we can generate our line plot using `geom_line()`:

```
> plt + geom_line()
```

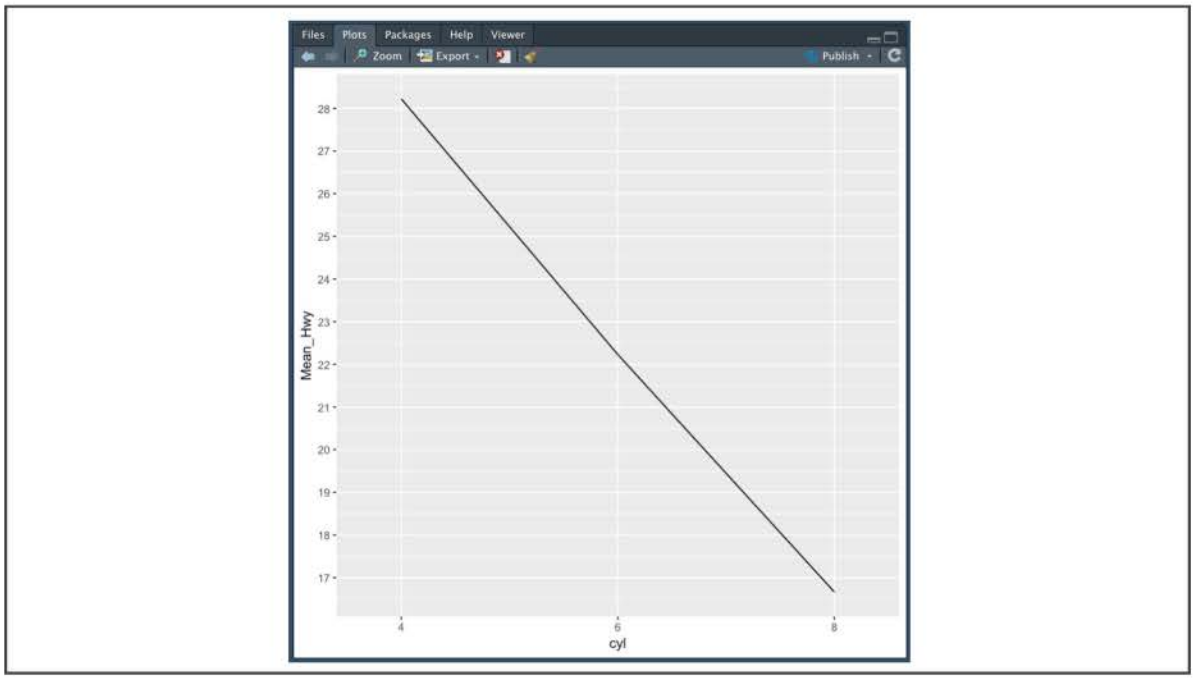


In this example, we can observe the general trend in the data: as the number of cylinders in Toyota vehicles increases, the average highway fuel economy decreases.

However, the default x-axis misrepresents the data because there are no five- and seven-cylinder vehicles. In addition, the default y-axis tick marks are too general and do not allow the reader to determine average fuel economy values.

To adjust the x-axis and y-axis tick values, we'll use the `scale_x_discrete()` and `scale_y_continuous()` functions:

```
> plt + geom_line() + scale_x_discrete(limits=c(4,6,8)) + scale_y_continu
```



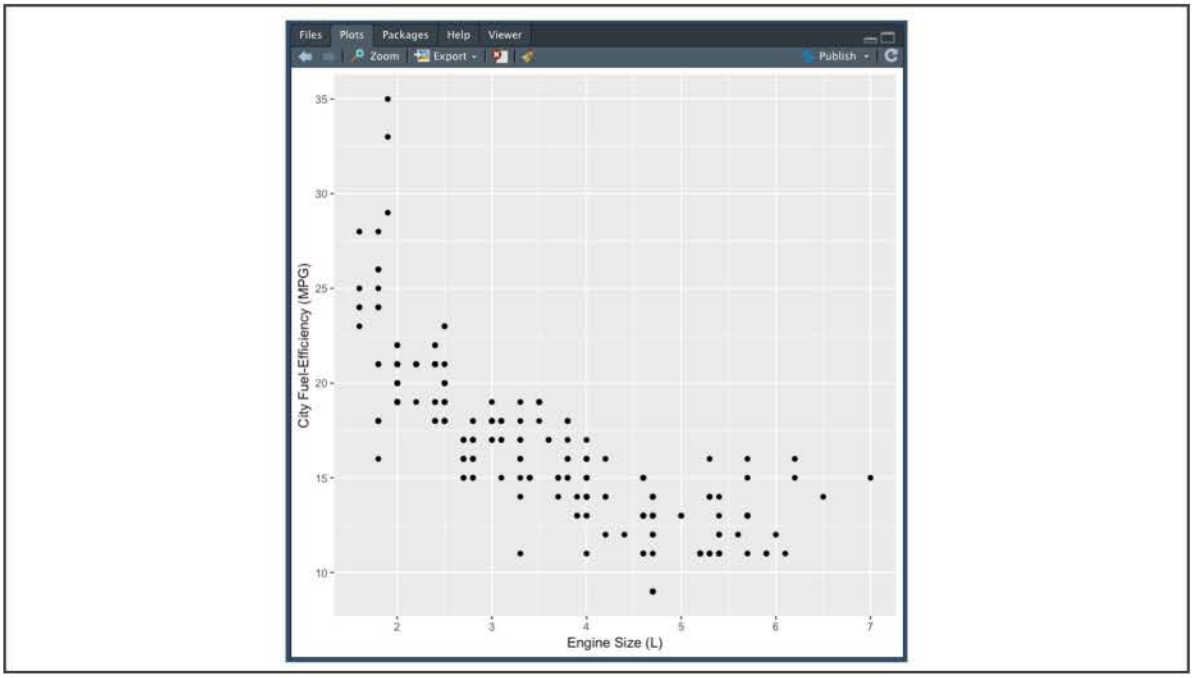
The `scale_x_discrete()` function tells ggplot to use explicit values for the x-axis ticks. In other words, the `scale_x_discrete()` function will generate x-axis ticks for each value in a list. In contrast, the `scale_y_continuous()` function tells ggplot to rescale the y-axis based on a defined range.

Implementing scatter plots in ggplot2 is just as easy as line plots. To set up our `ggplot` object for our scatter plot, we'll need to set the independent variable as our *x* value and the dependent variable as our *y* value within our `aes()` function. For example, if we want to create a scatter plot to visualize the relationship between the size of each car engine (`disp`) versus their city fuel efficiency (`cty`), we would create the following `ggplot` object:

```
> plt <- ggplot(mpg,aes(x=displ,y=cty)) #import dataset into ggplot2
```

Once we successfully create our `ggplot` object, we can generate our scatter plot using the `geom_point()` function:

```
> plt + geom_point() + xlab("Engine Size (L)") + ylab("City Fuel-Efficier
```



Although our default scatter plot visualizes the relationship between engine size and city fuel efficiency effectively, we can use scatter plots to visualize more than just two variables.

In data science, we're often interested in the relationship between two quantitative variables in regards to other categorical variables (often referred to as grouping factors). By customizing our data points with aesthetic changes, we can add additional context to our scatter plots to help convey more information within a single visualization. Let's see this in action.

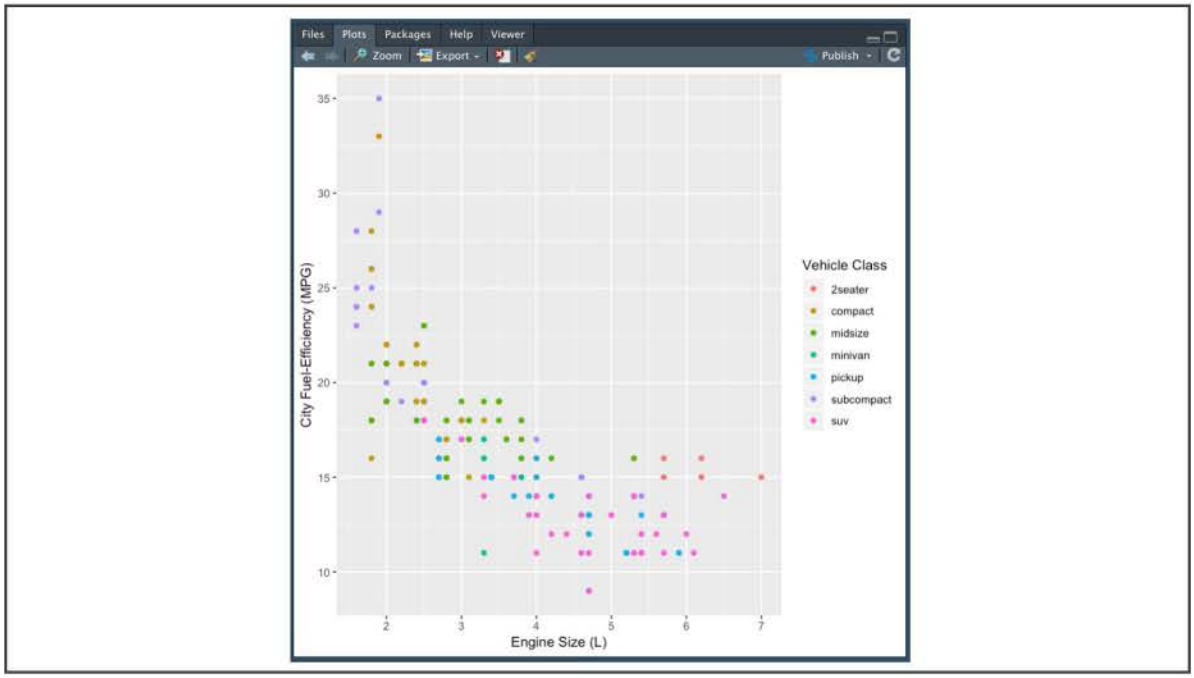
There are a number of customizing aesthetics we can add to our `aes()` function to change our scatter plot data points, such as:

- **alpha** changes the transparency of each data point

- **color** changes the color of each data point
- **shape** changes the shape of each data point
- **size** changes the size of each data point

If we apply these custom aesthetics to our previous example, we can use scatter plots to visualize the relationship between city fuel efficiency and engine size, while grouping by additional variables of interest:

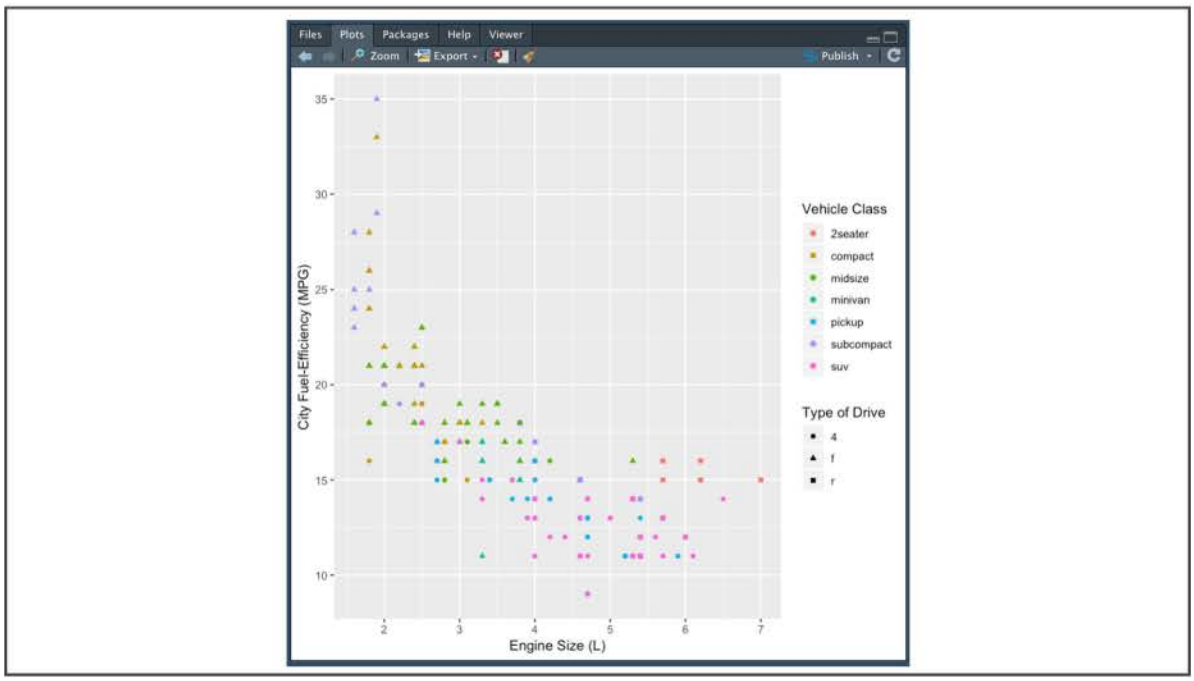
```
> plt <- ggplot(mpg,aes(x=displ,y=cty,color=class)) #import dataset into
> plt + geom_point() + labs(x="Engine Size (L)", y="City Fuel-Efficiency
```



> **Note**
> An alternative to `xlabs()` and `ylabs()` is the `labs()` function, which lets you customize your axis labels as well as any grouping variable labels.

By coloring each data point by its vehicle class, we can see that vehicle class data points are clustering together in regard to our engine size and city fuel efficiency. We're not limited to only adding one aesthetic either:

```
> plt <- ggplot(mpg,aes(x=displ,y=cty,color=class,shape=drv)) #import dat
> plt + geom_point() + labs(x="Engine Size (L)", y="City Fuel-Efficiency
```



Depending on the size of the plot, the number of data points, and the density of the data, some aesthetics work better than others. It's good practice to try building multiple versions of the same visualization with different aesthetics to determine which aesthetic most effectively conveys the results.

Skill Drill

Using the same dataset, create an additional visualization that uses City Fuel-Efficiency (MPG) to determine the size of the data point.

Hint: Use the **geom_point documentation (https://ggplot2.tidyverse.org/reference/geom_point.html#aesthetics)** for assistance.

Caution

Although there is no technical limit to the number of variables we can add to a ggplot figure, there are diminishing returns. A good rule of thumb is to

limit the number of variables displayed in a single figure to a maximum of 3 or 4.

IMPORTANT

Most of the customizations we've covered thus far can be used on any ggplot visualization, regardless of what type of plot we're using. Try to experiment adding custom aesthetics, labels, and axes to every plot. If you're ever uncertain how to customize, refer to the **ggplot2 documentation (https://ggplot2.tidyverse.org/reference/index.html)** .

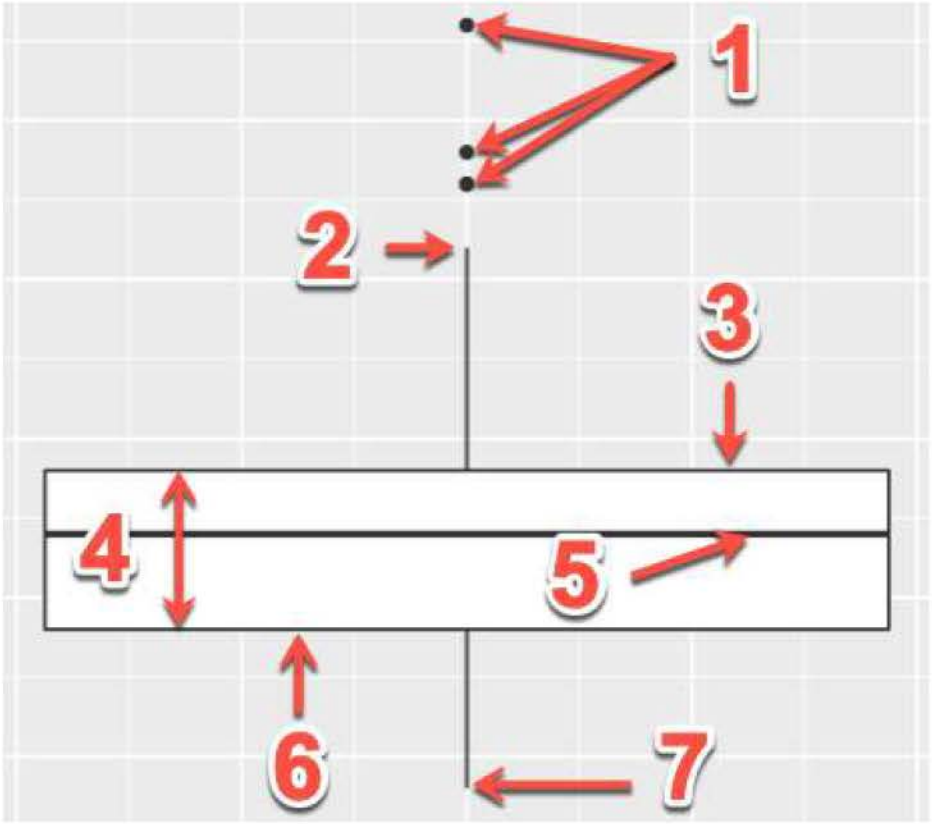
15.3.5: Create Advanced Boxplots in ggplot2

Jeremy and Colleen are excited about how their team is progressing. They take a break for lunch and start to brainstorm about how to really impress their CEO. Colleen suggests moving beyond basic charts to some more advanced visualizations, and Jeremy suggests a joint presentation to give Colleen a chance to shine. While Colleen practices her public speaking, Jeremy digs back into R.

Beyond the basic bar, line, and scatter plots, there are a number of more advanced ggplot2 visualizations that can be used to describe specific features of a dataset.

For instance, when performing statistical analysis, we may want to visualize summary statistics using boxplots, or unpack the relationship across multiple variables using a heatmap. Fortunately, ggplot2 has functions such as `geom_boxplot()` and `geom_tile()` that generate more advanced visualizations with ease.

The boxplot is also known as a box-and-whisker-plot, named for the lines extending from the boxes. It's used to visualize a variety of summary statistics for a continuous numerical vector. Boxplots are very common in data science due to the density of information contained within a single visualization, as well as the boxplot's ability to compare measurements across grouping factors.



Match the different statistical metrics to each boxplot element, as shown in the image above.

1	
2	
3	
4	
5	
6	
7	

Third quantile | Median | First quantile — 1.5 * IQR

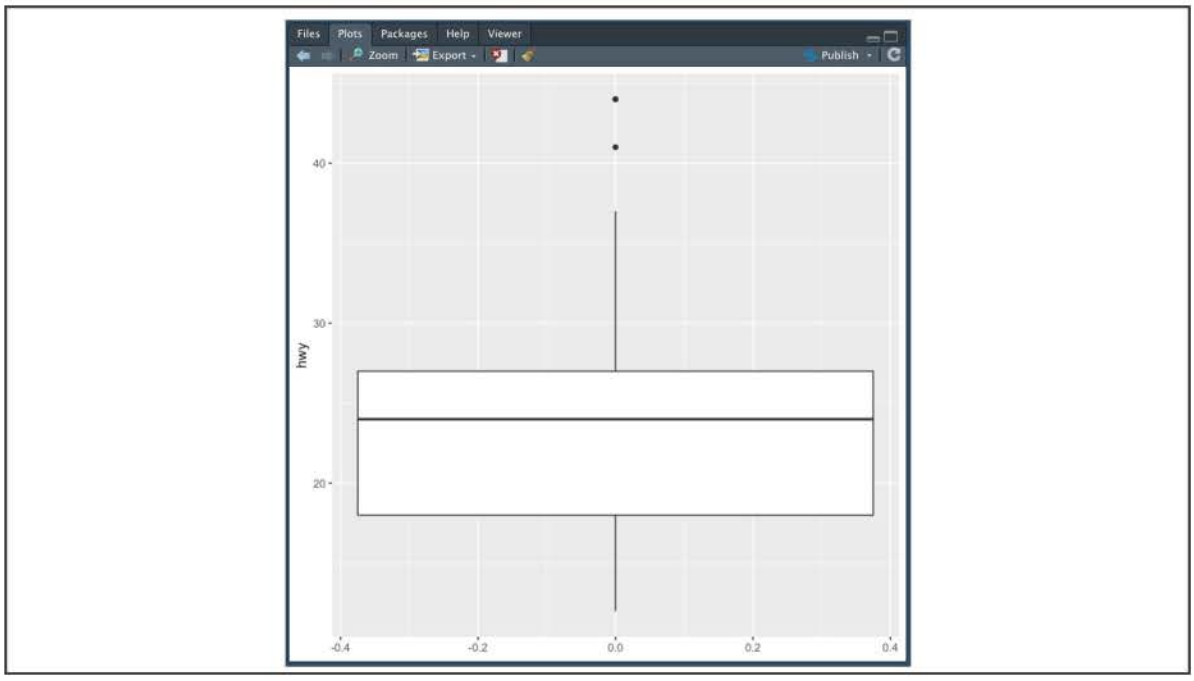
Potential outliers | Third quantile + 1.5 * interquartile range (IQR)

First quantile | IQR

Check Answer

To generate a boxplot in ggplot2, we must supply a vector of numeric values. For example, if we want to generate a boxplot to visualize the highway fuel efficiency of our mpg dataset, our R code would look as follows:

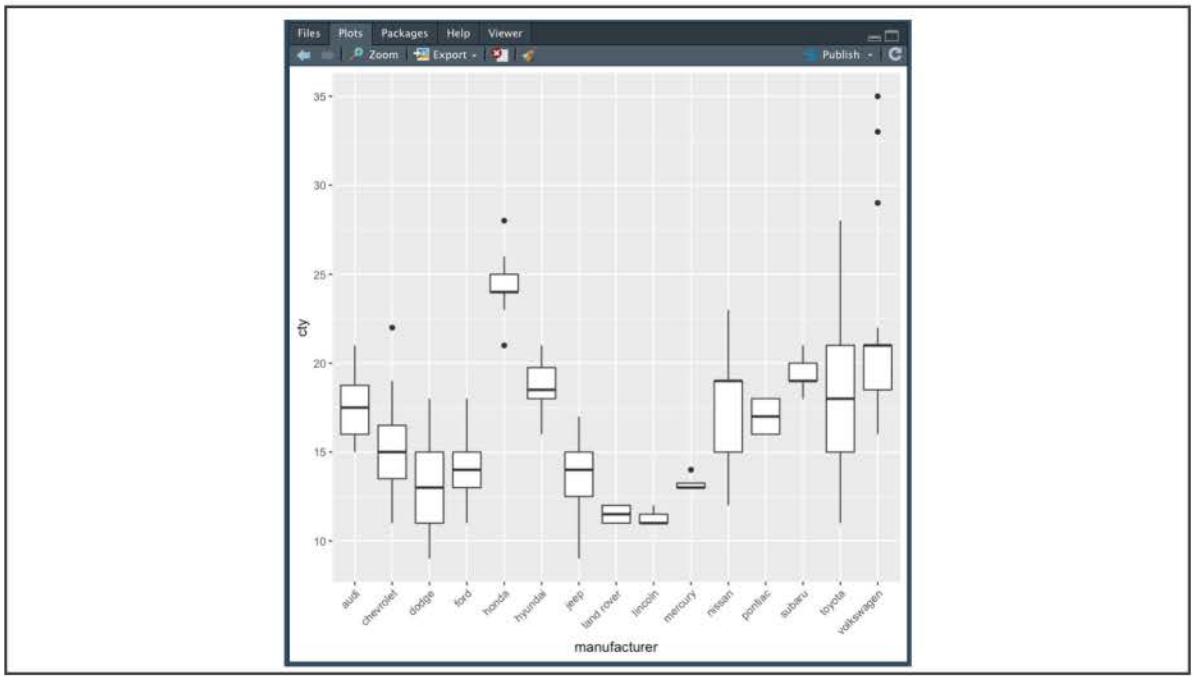
```
> plt <- ggplot(mpg,aes(y=hwy)) #import dataset into ggplot2
> plt + geom_boxplot() #add boxplot
```



Unlike with the previous `ggplot` objects, `geom_boxplot()` expects a numeric vector assigned to the y-value. This is due to the ggplot accounting for multiple boxplots in a single figure. If we were to supply our categorical grouping factor to x, we can create a boxplot that compares measurements from a variety of groups.

Expanding on our previous example, if we want to create a set of boxplots that compares highway fuel efficiency for each car manufacturer, our new R code would look as follows:

```
> plt <- ggplot(mpg,aes(x=manufacturer,y=hwy)) #import dataset into ggplo
> plt + geom_boxplot() + theme(axis.text.x=element_text(angle=45,hjust=1)
```



These grouped boxplots are fantastic to use in technical reports and presentations due to how easy they are to read and interpret as well as how much information can be conveyed.

> **Skill Drill**
>
> Customize the boxplot to be more aesthetic by adding some color and using dotted instead of solid lines.
>
> **Hint:** Reference the **geom_boxplot documentation (https://ggplot2.tidyverse.org/reference/geom_boxplot.html#aesthetics)** for assistance.

> **Note**
>
> You can apply numerous custom aesthetics to boxplots to improve visualizations so they are easier to read. Be sure to check out the **ggplot2**

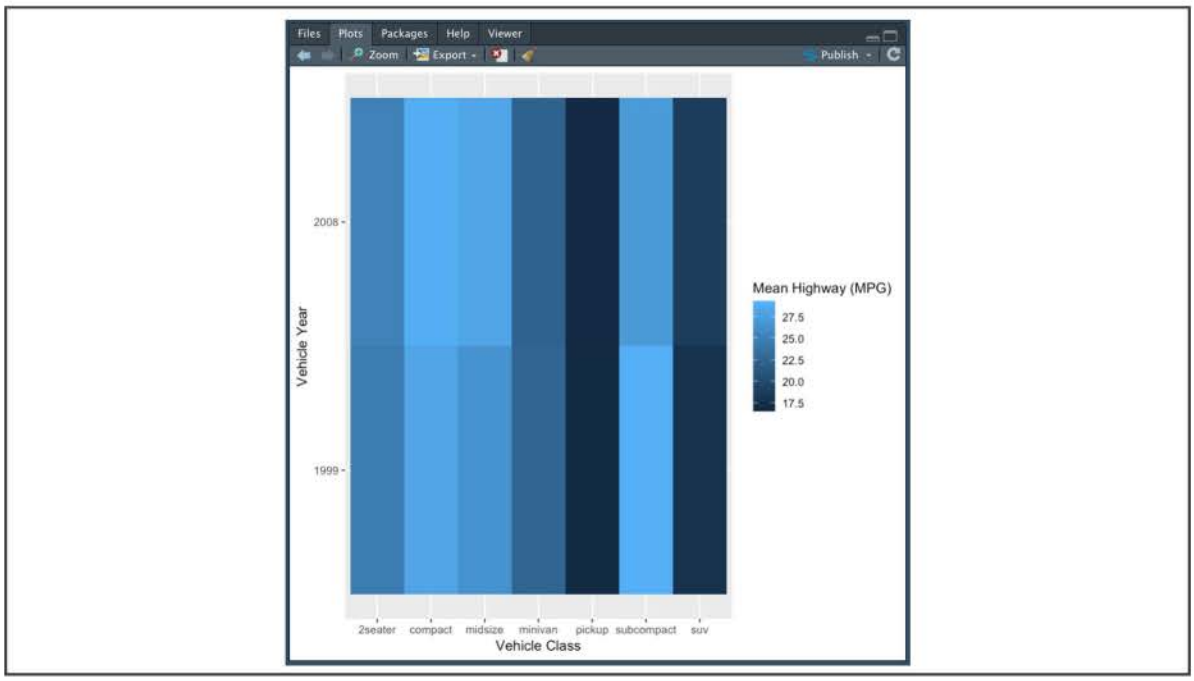
boxplot documentation (https://ggplot2.tidyverse.org/reference/geom_boxplot.html) .

15.3.6: Create Heatmap Plots

Jeremy had been looking at this fuel efficiency data for a bit when suddenly, inspiration struck: He could use a heatmap to visualize the average highway fuel efficiency across the type of vehicle class from 1999 to 2008. Heatmaps are a useful way to see intensity across time—and this type of chart could go over really well in a high-level presentation, as it's a good way to see a lot of information at once. Time to get coding!

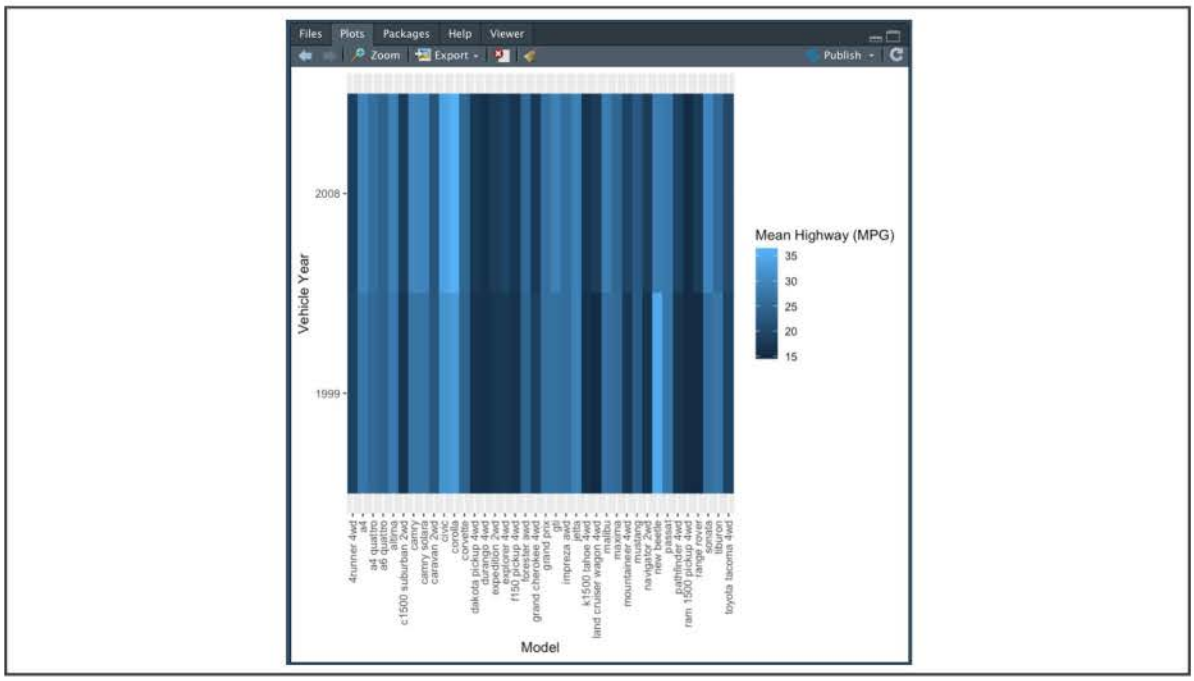
Heatmap plots help visualize the relationship between one continuous numerical variable and two other variables (categorical or numerical). Heatmaps display numerical values as colors on a two-dimensional grid so that value clusters and trends are readily identifiable. For example, if we want to visualize the average highway fuel efficiency across the type of vehicle class from 1999 to 2008, our R code would look as follows:

```
> mpg_summary <- mpg %>% group_by(class,year) %>% summarize(Mean_Hwy=mean
> plt <- ggplot(mpg_summary, aes(x=class,y=factor(year),fill=Mean_Hwy))
> plt + geom_tile() + labs(x="Vehicle Class",y="Vehicle
Year",fill="Mean Highway (MPG)") #create heatmap with labels
```



As our heatmap shows, nearly all vehicle classes experienced an average improvement in highway fuel efficiency from 1999 to 2008. Unlike our previous ggplot visualizations, heatmaps are used to look at large trends in a dataset. Therefore, we can use heatmaps to visualize variables with a large number of values/categories. For example, if we want to look at the difference in average highway fuel efficiency across each vehicle model from 1999 to 2008, our R code would look as follows:

```
> mpg_summary <- mpg %>% group_by(model,year) %>% summarize(Mean_Hwy=mear
> plt <- ggplot(mpg_summary, aes(x=model,y=factor(year),fill=Mean_Hwy)) #
> plt + geom_tile() + labs(x="Model",y="Vehicle Year",fill="Mean Highway
```



> **Note**
>
> When using variables with a number of different values (categories, levels, etc.), you may want to adjust the angle of your text from 45 to 90 degrees. If you angle your text, you may need to also adjust your `hjust` and `vjust` arguments to ensure your labels line up with your tick marks.

Although boxplots and heatmaps are two of the more common advanced visualizations used in data science, there are a number of more specific ggplot2 visualization functions that can be used on an individual basis.

The ggplot2 documentation sufficiently describes how to implement each function, but does not provide logic or guidance on how to select a visualization for your data. Thankfully, there are many cheat sheets available that provide guidance on what visualizations to use given the dimensions and data types you wish to use. One of the most popular is the **RStudio Data Visualization ggplot2 Cheat Sheet (https://rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf)**, which is used to help data scientists determine what functions to use to generate an appropriate visualization for their analysis.

15.3.7: Add Layers to Plots

Jeremy has mastered the boxplot and heat map. What about adding additional plots to his visualizations?

Often when we're building visualizations for data analysis, we'll want to produce layered plots or combine very similar plots into a single visualization (also referred to as faceting, which we'll cover later on).

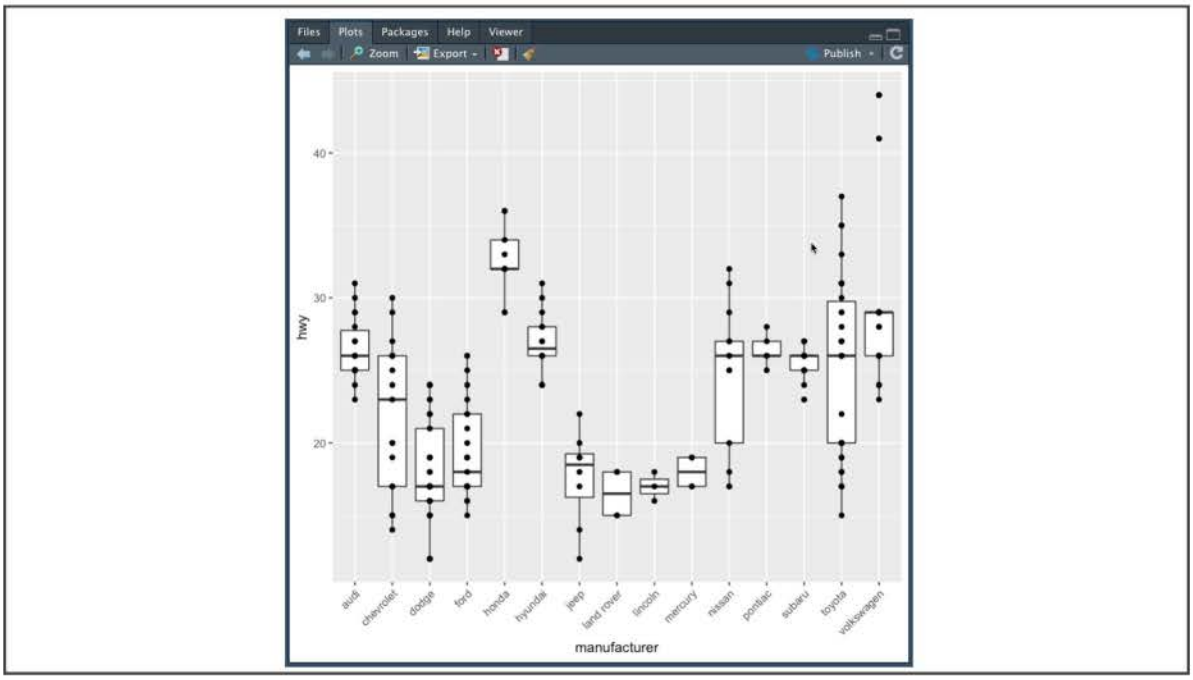
There are two types of plot layers:

1. Layering additional plots that use the **same variables and input data** as the original plot
2. Layering of additional plots that use **different but complementary data** to the original plot

We can add additional plots to our visualization by adding additional `geom` functions to our plotting statement. Layering plots that share input variables can be beneficial when you want to add context to your initial visualization.

For example, to recreate our previous boxplot example comparing the highway fuel efficiency across manufacturers, add our data points using the `geom_point()` function:

```
> plt <- ggplot(mpg,aes(x=manufacturer,y=hwy)) #import dataset into ggplo
> plt + geom_boxplot() + #add boxplot
> theme(axis.text.x=element_text(angle=45,hjust=1)) + #rotate x-axis labe
> geom_point() #overlay scatter plot on top
```



By layering our data points on top of our boxplot, we can see the general distribution of values within each box as well as the number of data points. This new information can provide the reader better context when comparing two manufacturers with similarly shaped boxplots.

Although layering plots with visualizations using the same input variables is a more common approach, there may be instances when we would want to add additional plotting layers with new and complementary data.

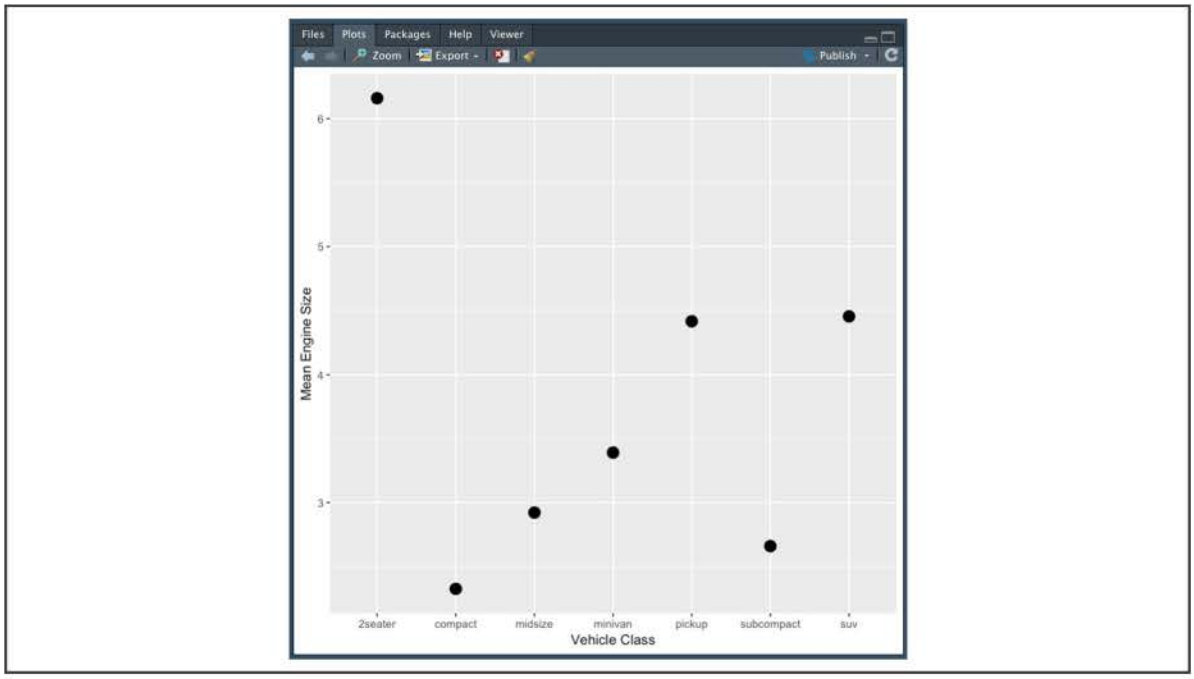
For example, what if we want to compare average engine size for each vehicle class? In this case, we would supply our new data and variables directly to our new `geom` function using the optional mapping and data arguments.

The **mapping argument** functions exactly the same as our `ggplot()` function, where our mapping argument uses the `aes()` function to identify the variables to use. Additionally, the **data argument** can be used to provide a new input data structure; otherwise, the mapping function will reference the data structure provided in the `ggplot` object.

Our R code would be as follows:

```
> mpg_summary <- mpg %>% group_by(class) %>% summarize(Mean_Engine=mean(
```

```
> plt <- ggplot(mpg_summary,aes(x=class,y=Mean_Engine)) #import dataset i
> plt + geom_point(size=4) + labs(x="Vehicle Class",y="Mean Engine Size")
```



True or False: Two datasets with equal means must have the same input data.

- True
- False

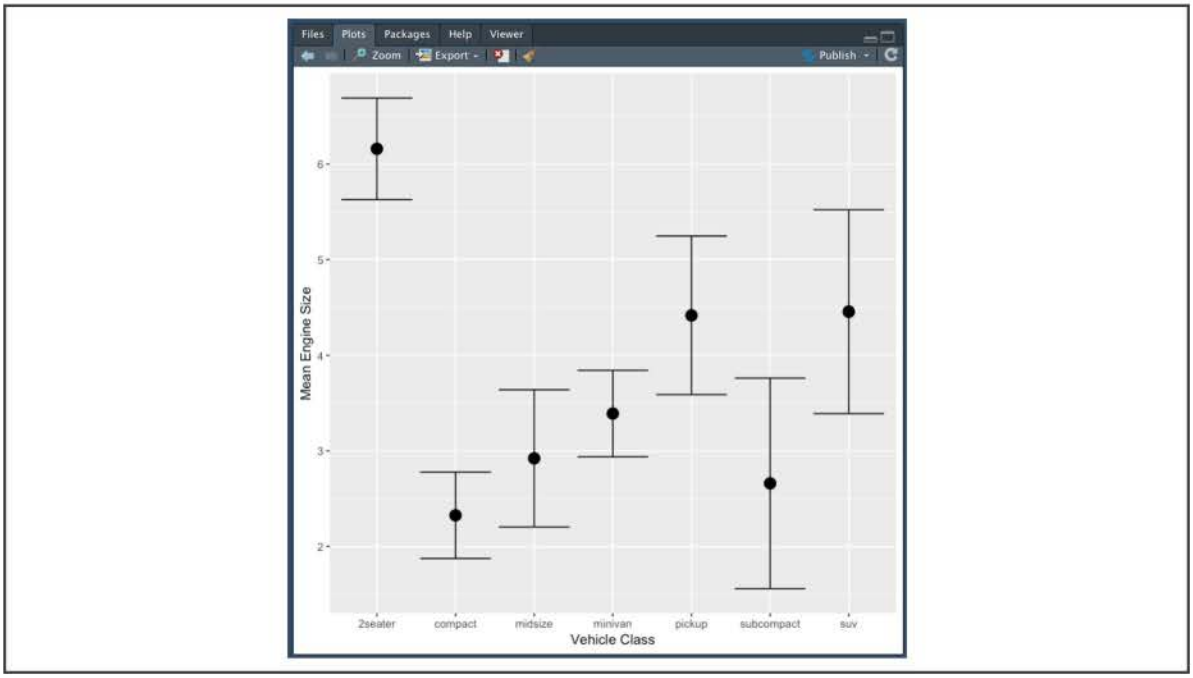
Check Answer

Finish ▸

Although this plot sufficiently visualizes the means, it's critical that we provide context around the standard deviation of the engine size for each vehicle class. If we compute the standard deviations in our dplyr `summarize()` function, we can layer the upper and lower standard deviation boundaries to our visualization using the `geom_errorbar()` function:

```
> mpg_summary <- mpg %>% group_by(class) %>% summarize(Mean_Engine=mean(d
> plt <- ggplot(mpg_summary,aes(x=class,y=Mean_Engine)) #import dataset i
> plt + geom_point(size=4) + labs(x="Vehicle Class",y="Mean Engine Size")
```

```
> geom_errorbar(aes(ymin=mean_engine-sd_engine,ymax=mean_engine+sd_engine
```



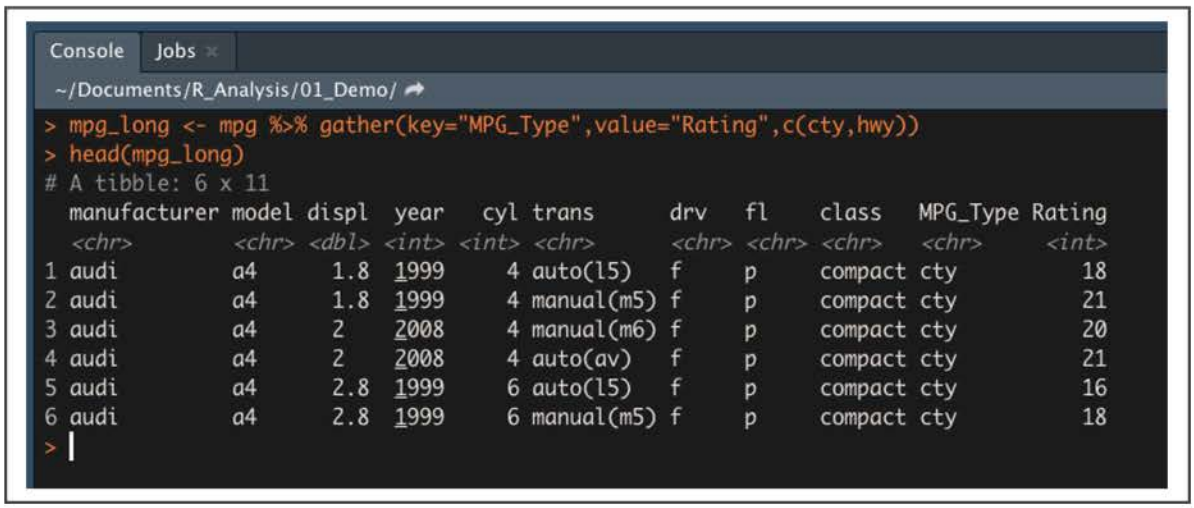
Layering plots can be very helpful visualizing wide-format data or summary data when there are multiple variables and metrics used to describe a single subject. As the number of subjects increases, or if the input data is in a long format, layering might not be as effective.

> **Caution**
> Not all visualizations will benefit from layering plots. Before adding layers of information, ask yourself if the new layer will add context without distracting or taking away from the original plot.

Often when our data is in a long format, we want to avoid visualizing all data within a single plot. Rather, we want to plot all our measurements but keep each level (or category) of our grouping variable separate. This process of separating out plots for each level is known as faceting in ggplot2.

Faceting is performed by adding a `facet()` function to the end of our plotting statement. Consider, if instead of the wide format, our mpg dataset was obtained where city and highway fuel efficiency data was provided in a long format:

```
> mpg_long <- mpg %>% gather(key="MPG_Type",value="Rating",c(cty,hwy)) #c
> head(mpg_long)
```



Fill in the missing function arguments below to rotate the x-axis labels 90 degrees.

Hint: Refer to the ggplot2 theme documentation and Stack Overflow responses.

```
theme(  )
```

How would you change the previous answer to rotate the x-axis labels 45 degrees?

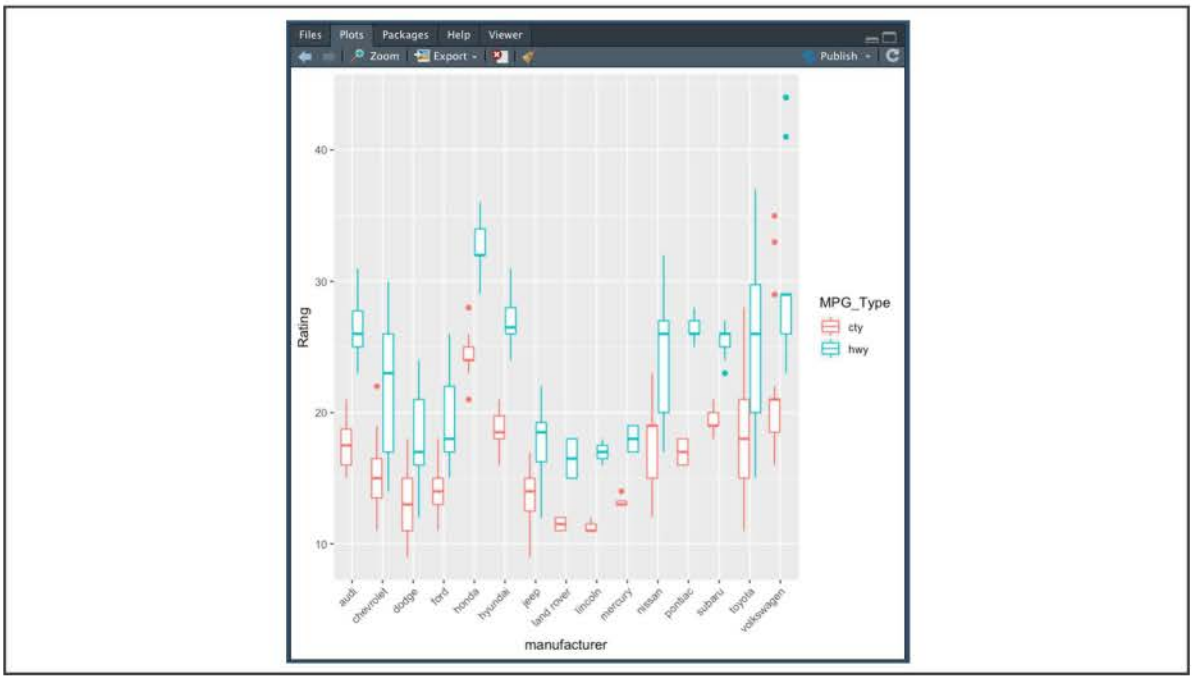
```
theme(  )
```

Check Answer

Finish ▸

If we want to visualize the different vehicle fuel efficiency ratings by manufacturer, our R code would be as follows:

```
> plt <- ggplot(mpg_long,aes(x=manufacturer,y=Rating,color=MPG_Type)) #im
> plt + geom_boxplot() + theme(axis.text.x=element_text(angle=45,hjust=1)
```



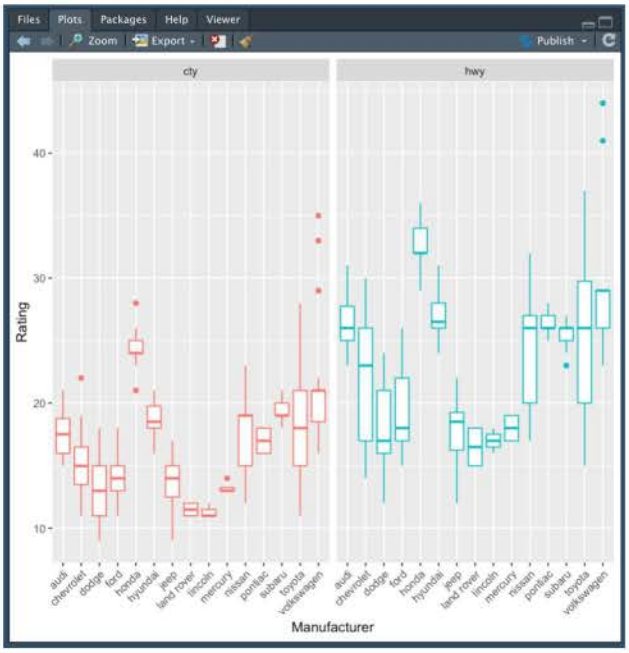
The produced boxplot is optimal for comparing the city versus highway fuel efficiency for each manufacturer, but it is more difficult to compare all of the city fuel efficiency across manufacturers. One solution would be to facet the different types of fuel efficiency within the visualization using the `facet_wrap()` function.

Type the following code into the R console to look at the `facet_wrap()` documentation in the Help pane:

```
> ?facet_wrap()
```

Similar to any of ggplot2's `geom` functions, the `facet_wrap()` function has many optional variables to tweak the direction and type of faceting. However, the most basic use cases for faceting only require us to provide the annotation for the **facets** argument. The facets argument expects a list of grouping variables to facet by using the `vars()` function. Therefore, to facet our previous example by the fuel-efficiency type, our R code could be as follows:

```
> plt <- ggplot(mpg_long,aes(x=manufacturer,y=Rating,color=MPG_Type)) #in
> plt + geom_boxplot() + facet_wrap(vars(MPG_Type)) + #create multiple bo
> theme(axis.text.x=element_text(angle=45,hjust=1),legend.position = "nor
```



By faceting our boxplots by fuel-efficiency type, it's easier to make comparisons across manufacturers. In this example, we faceted two levels/groups, but more complicated long-format datasets may contain measurements for multiple levels. Using faceting can help make data exploration of these complex datasets easier or can help isolate factors of interest for our audience.

Caution

Although there is no hard limit to the number of faceted plots that can be generated, too many faceted plots can render a visualization useless. Generally speaking, the more axis ticks you need to convey your data, the fewer facets you should use.

Skill Drill

1. Create one or two additional plots using a different variable for the `facet_wrap()`.
2. Create another plot using two or more variables for the `facet_wrap()`. With this data, does adding more variables make the chart easier or harder to understand?

> **Note**
>
> There are many ways to implement the `facet_wrap()` function, and the documentation can be fairly involved. Thankfully, there are plenty of very simple examples online that can help you customize your faceting.

Now that we have practiced using R to load in a dataset, perform transformation functions, and visualize data, we are ready to analyze our datasets using R.

15.4.1: Identifying Statistical Test Types

It's been a few days since Jeremy began teaching himself to learn R, and he's beginning to feel more confident. He knows there's still a lot to learn about data analysis, statistics, and reporting, but he feels ready to start cutting his teeth on some real data.

Jeremy asked Colleen to show him some of the datasets they were working on, but he didn't expect each file to be so different! Although Jeremy has worked with a variety of datasets during his time at AutosRUs, these aggregated data tables are an entirely new format. Fortunately, Jeremy knows that with some time and patience, he can use R to explore and demystify these data tables and prepare himself for some statistical testing!

When presented with a new or foreign dataset, Jeremy knows it's good practice to familiarize yourself with each data column, the dimensions of the data, and the overall characteristics of the dataset. He's going to start this section of his analysis by exploring the different characteristics to look out for in a dataset, and what each characteristic means when it comes to analysis.

For the remainder of this module, we'll focus on data analytics using hypothesis testing and statistics. Although the discipline of statistics has many specializations and nearly limitless applications, there are only a few concepts required to get started. In this introduction, we'll cover some core statistical concepts such as:

- mathematical data types
- null and alternative hypothesis
- p-values and hypothesis testing
- t-test of the means
- correlation and linear regression tests
- comparing frequency distribution using chi-squared test

Due to the structured nature of statistical testing, if we can determine any two components of a statistical analysis (input variables, analytical question, or statistical test), we can infer the third. To simplify the process of inferring what component is needed, you can use a statistical test lookup table such as the following:

Statistical Test	**Input Variable Type**				**Analytical Question**
	Independent		**Dependent**		
	No. of Variables	**Data Type**	**No. of Variables**	**Data Type**	
One-sample t-test	1	Dichotomous (Population or Sample)	1	Continuous	Is there a statistical difference between

					the mean of the sample distribution and the mean of the population distribution?
Two-sample t-test	1	Dichotomous (Sample A versus Sample B)	1	Continuous	Is there a statistical difference between the distribution means from two samples?
ANOVA	1+	Categorical	1	Continuous	Is there a statistical difference between the distribution

					means from multiple samples ?
Simple linear regression	1	Continuous	1	Continuous	Can we predict values for a dependent variable using a linear model and values from the independent variable?
Multiple linear regression	2+	Continuous	1	Continuous	How much variance in the dependent variable is accounted for in a linear combinat

					ion of independent variables?
Chi-squared	1	Categorical	1+	Categorical	Is there a difference in categorical frequencies between groups?

There are several helpful lookup tables that can reinforce statistical concepts. For your convenience, we've aggregated many of these lookup tables into a single **cheat sheet (https://courses.bootcampspot.com/courses/138/files/18479/download?wrap=1)**  **(https://courses.bootcampspot.com/courses/138/files/18479/download?wrap=1)** you can use for the remainder of the module.

In addition, the remaining sections will provide supplementary notes on the in-depth statistical concepts. These will focus on common statistical interview questions, analytical "gotchas," and other pertinent context to bolster your understanding of each statistical concept. If this is your first time learning statistical concepts, or if you're still brushing off the mental cobwebs, we suggest you delay studying these optional notes until you're comfortable with this module's core statistical concepts. Take your time with each section, and refer to the statistics cheat sheet for help.

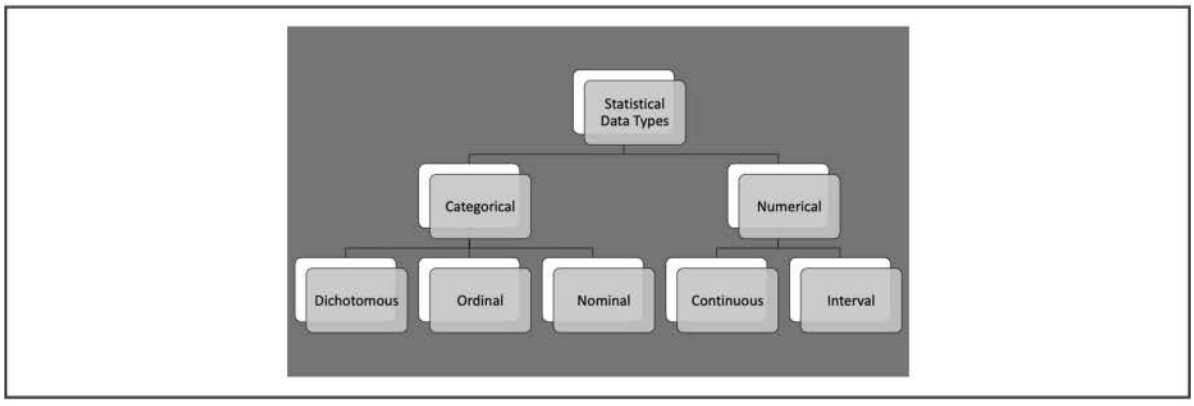
> **IMPORTANT**
> Once you're more familiar with statistical analysis, start searching for more elaborate statistical lookup tables with non-normal, generalized statistical

tests. As you practice characterizing datasets, asking questions, determining a hypothesis, and testing using statistics, the process will become easier.

15.4.2: Identify Different Data Types

Now that Jeremy feels comfortable with R, he's excited to jump into statistics. Colleen has been wanting to brush up on her statistics knowledge as well, so she suggests that their new team start an internal study group. Their first session is about different data types.

There are two major data types in statistics: categorical and numerical. Within these types are several subtypes, each with its own use cases. In this section, we'll describe these types and explain how to analyze data effectively. First, take a look at the following diagram to visualize how data is categorized.



Categorical Data

Categorical data represents data characteristics or qualitative descriptions. Generally, categorical data is any data that is not measured, also known as qualitative data. Categorical data can be collected in the form of strings,

true/false Boolean values, or even encoded numbers as categories (such as one for red, two for blue, three for green, etc.). Several statistical tests use categorical data to inform which groups to compare. Categorical data has three subtypes: dichotomous, ordinal, and nominal

---

Dichotomous Data

Dichotomous data is collected from either one of two categories. For example, an online survey might collect member/non-member or demographic information. Dichotomous data can be collected in the form of true/false Boolean values, 0 or 1 binary values, or two strings. Later in the module, we'll use dichotomous data to help perform many of our comparative statistical tests.

Ordinal Data

Ordinal data has a ranked order. Although ordinal data has a sequence, we don't necessarily know the value between each ordinal data point. Data that is collected on a value scale (e.g., movie rankings, survey results, and the **Likert scale (https://en.wikipedia.org/wiki/Likert_scale)**) are common forms of ordinal data. Ordinal data combines the qualitative properties of labels to the quantitative properties of scale to allow for comparative analyses. Ordinal data is very popular with research and survey groups because it allows for quantitative analysis without the need of machinery and tools to obtain measurements.

> **Note**
>
> There are statistical tests such as the Mann-Whitney U test and the Kruskal-Wallis H test that compare ordinal datasets. These statistical tests are more advanced versions of basic comparative statistics tests and are outside the scope of this course. However, once you master the basics of statistical

> testing, it is not difficult to apply more advanced statistical models based on your specific data needs. Remember—Google is your best friend!

Nominal Data

Nominal data is data used as labels or names for other measures. Nominal data can be as individual as an identification number or can be as general as a list of three options. Unlike ordinal data, nominal data has no ranking. Therefore, nominal data is often used with a more quantitative data type to perform an analysis. Often nominal data will be transformed using a grouping function to decrease the complexity of the data.

Numerical Data

Typically, **numerical data** is obtained by taking a measurement from an instrument (such as a ruler, measuring scale, sensor, etc.) or by counting. In statistics, numerical data is used to perform quantitative analysis that can produce the probability of an outcome or quantify the relationship between categories. Within numerical data there are two primary data types to consider: continuous and interval.

Continuous Data

Continuous data can be subdivided infinitely. For example, if you want to describe the thickness of window glass, you could measure it in x number of centimeters, millimeters, nanometers, picometers, and so on. Continuous data is typically recorded with decimal places to match the precision of the measurement. Almost all statistical tests and models use continuous data to generate precise results.

Interval Data

Interval data is spaced out evenly on a scale. Also known as integer data, interval data does not use decimal places and can't be subdivided. Interval data also can't be multiplied or divided. Because interval data is spaced out evenly, it can be grouped together or bucketed easily. For example, a set of integers 15, 4, 18, 10, 3 , and 5 could be collected as a group that is less than 20. Due to this property, interval data can be treated as a numerical data type or transformed into a nominal data type.

Additionally, interval data can be generated through rounding continuous data at the cost of losing precision of the measurement. Therefore, interval data can be used by most statistical models as either a quantitative or qualitative variable, depending on the use case.

Now that we understand the different statistical data types and how to identify each by its characteristics, we should be able to classify any tabular dataset.

Getting Oriented with Data

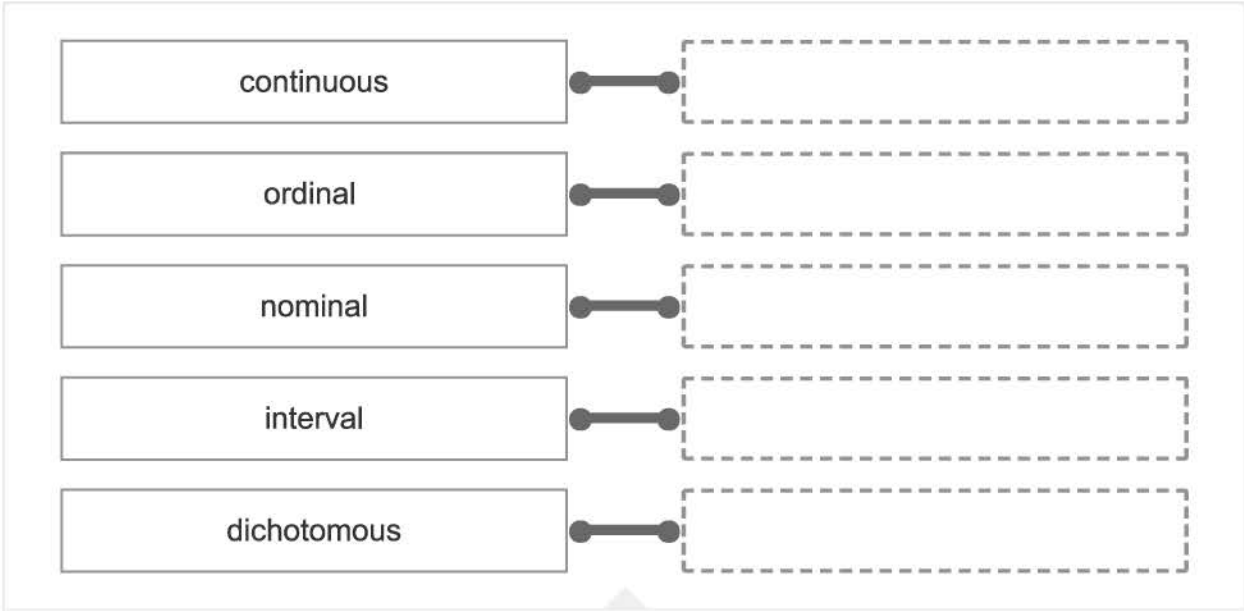
The easiest means of orienting ourselves on a new dataset in R is to use the `head()` function, which shows us the first few rows of our data frame. At any point when looking at the first few rows, we can use bracket notation (or the `$` operator) to select an individual column to explore.

Alternatively, if we're using RStudio, we can explore any data frame by clicking on it in our environment pane. By navigating through each column and classifying each data type, we can determine which columns provide measurement results, and which columns provide characteristics about our subjects.

If we're fortunate to have context provided for a given dataset via documentation or from the data collector, we should be able to identify columns and metrics of

interest. However, we have not finished characterizing our data just yet—we still need to understand how values in our data are distributed.

Match the following data types to their definitions.



Check Answer

15.4.3: Dive Into Distributions

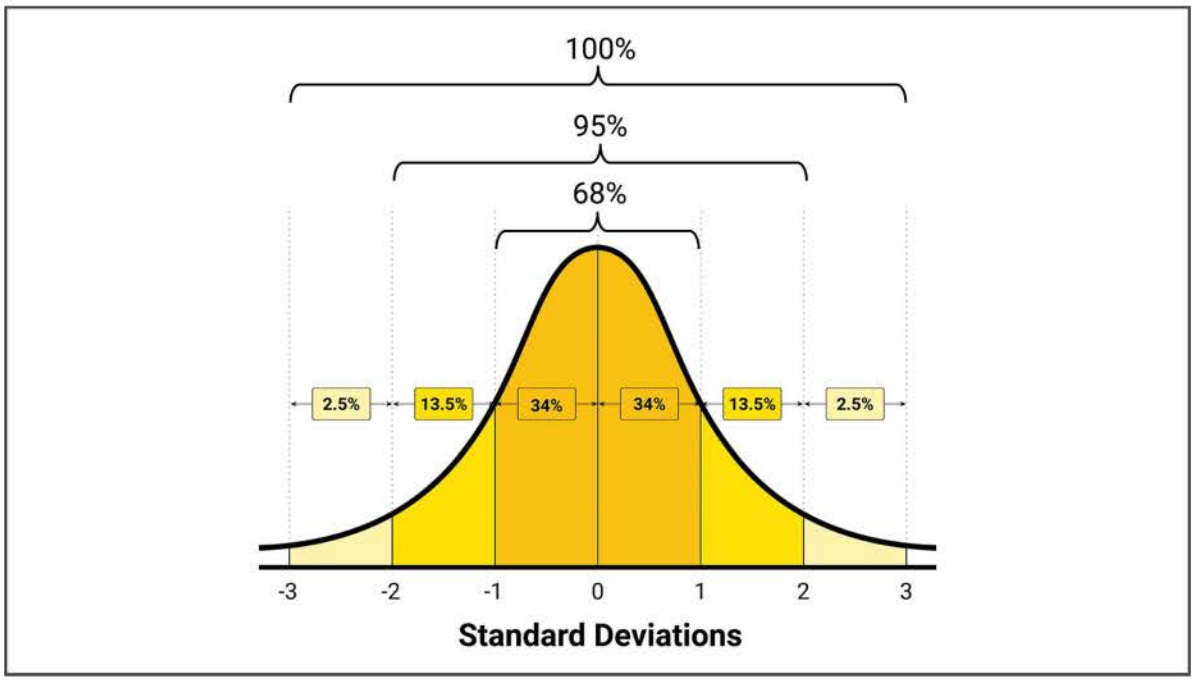
Now that the team feels comfortable with different data types, Jeremy wants to do a quick refresher on data distributions. In other words, Jeremy wants to understand the shape of his data before performing any analysis.

When it comes to data analysis, characterizing the distribution of numerical data is as important as characterizing the different data types. If we get this wrong, our results could be meaningless. This is because the majority of basic statistical tests assume that each numerical metric follows an approximate normal distribution. In other words, we must confirm our data is normal before we can use a statistical test. If the data does not follow an approximate normal distribution, we would need to implement a more generalized (and oftentimes more complicated), non-normal statistical function.

Fortunately, there are qualitative and quantitative tests we can use to test our data for normality to avoid using these more generalized functions.

What Is Normal Distribution?

Normal distribution, or normality, is commonly referred to as "the bell curve," and describes a dataset where values farther from its mean occur less frequently than values closer to its mean.



When numerical data is considered to be normally distributed, the probability of any data point follows the **68-95-99.7** rule, stating that 68.27%, 95.45%, and 99.73% (effectively 100%) of the values lie within one, two, and three standard deviations of the mean, respectively.

In statistics, the **central limit theorem** is a key concept that states if you take a sufficiently large sample of data from a dataset with mean μ and standard deviation σ, then the distribution will approximate normal distribution. Therefore, if we are using relatively large sample sizes, we should expect data to become more normally distributed.

15.4.4: Test for Normality

Jeremy had forgotten how important distribution is! After brushing up on distribution, he asks Colleen to give him a quick recap on how to actually test for normality.

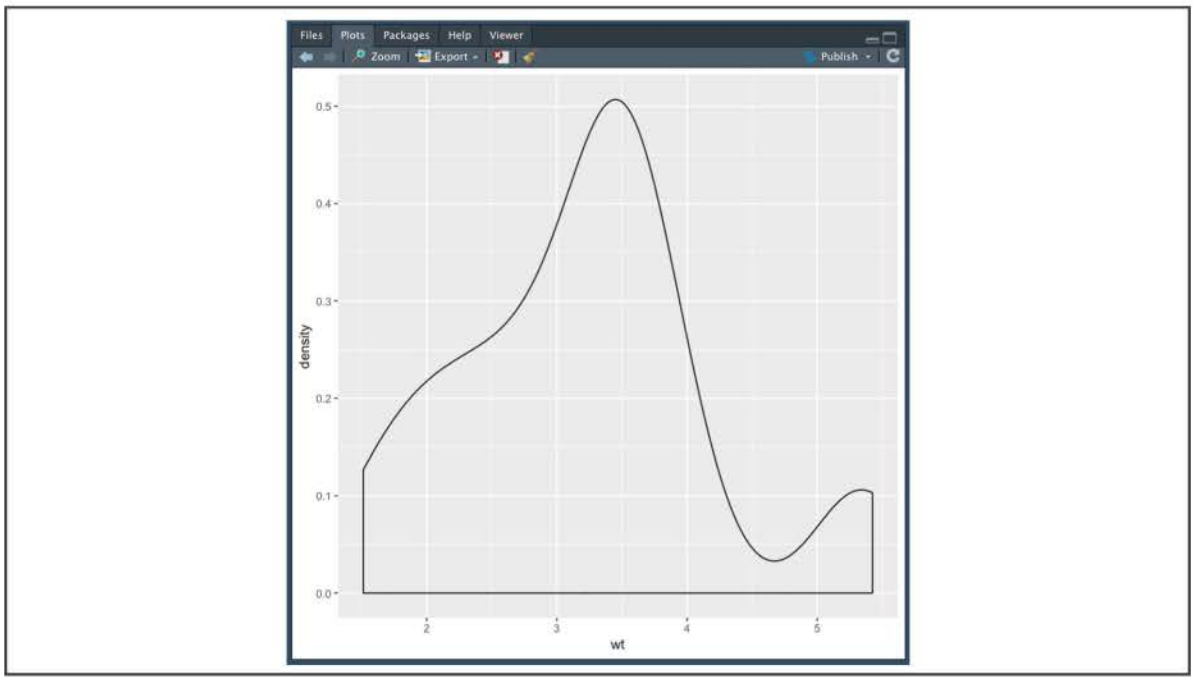
You can test for normality during data analysis by performing a qualitative test or a quantitative test.

Qualitative Test for Normality

The **qualitative test for normality** is a visual assessment of the distribution of data, which looks for the characteristic bell curve shape across the distribution. In R, we would use ggplot2 to plot the distribution using the `geom_density()` function.

For example, if we want to test the distribution of vehicle weights from the built-in `mtcars` dataset, our R code would be as follows:

```
> ggplot(mtcars,aes(x=wt)) + geom_density() #visualize distribution using
```



The `geom_density()` function plots a numerical vector by creating buckets of similar values and calculating the density (number of bucket data points/total number of data points) for each bucket.

The results of each bucket density calculation are plotted, connected, and smoothed out to create our distribution plot. Although our data distribution does not perfectly match the normal bell curve shape, the distribution does approximate a normal distribution and could be used for further analysis.

But what if our data distribution is noisy—meaning that the dataset contains uncharacteristically large or small values at high frequency—or we need to make more informed, quantitative decisions? In these cases, we would want to perform our quantitative test.

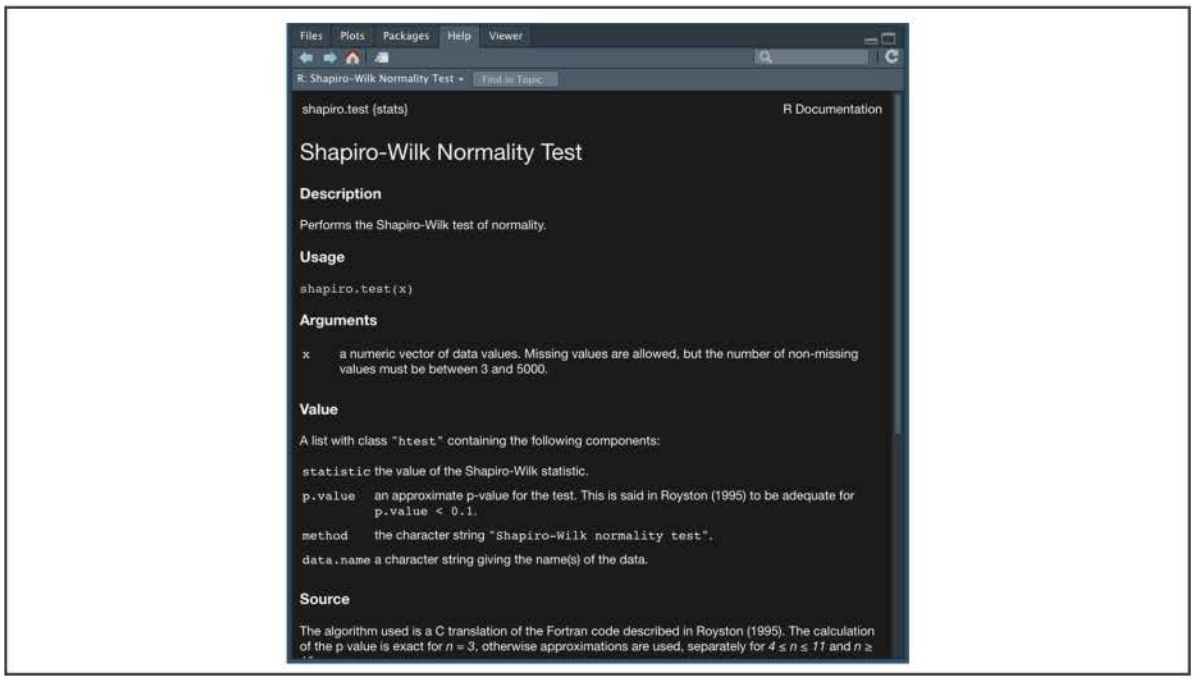
Quantitative Test for Normality

The **quantitative test for normality** uses a statistical test to quantify the probability of whether or not the test data came from a normally distributed dataset.

In most cases, data scientists will use the Shapiro-Wilk test for normality, though there are many other statistical tests available. In R, we can use the built-in stats library to perform our quantitative test with the `shapiro.test()` function.

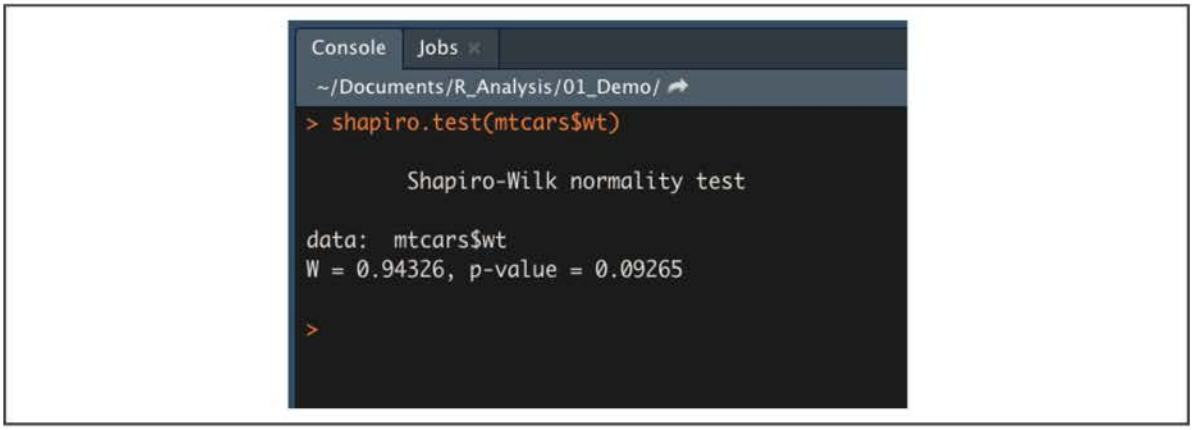
Type the following code into the R console to look at the `shapiro.test()` documentation in the Help pane:

```
>?shapiro.test()
```



The `shapiro.test()` function only requires the numeric vector of values you wish to test. Therefore, if we want to perform a quantitative Shapiro-Wilk test on our previous example, our R code would look as follows:

```
> shapiro.test(mtcars$wt)
```



Later we'll discuss what a p-value is and how it is used in statistics. For our purposes, you just need to know that if the p-value is greater than 0.05, the data is considered normally distributed.

Remember that most basic statistical tests assume an **approximate normal distribution**. Therefore, if our p-value is around 0.05 or more, we would say that our input data meets this assumption. But what happens if our data distribution does not look like a bell curve, or the p-value of the Shapiro-Wilk tests is too small?

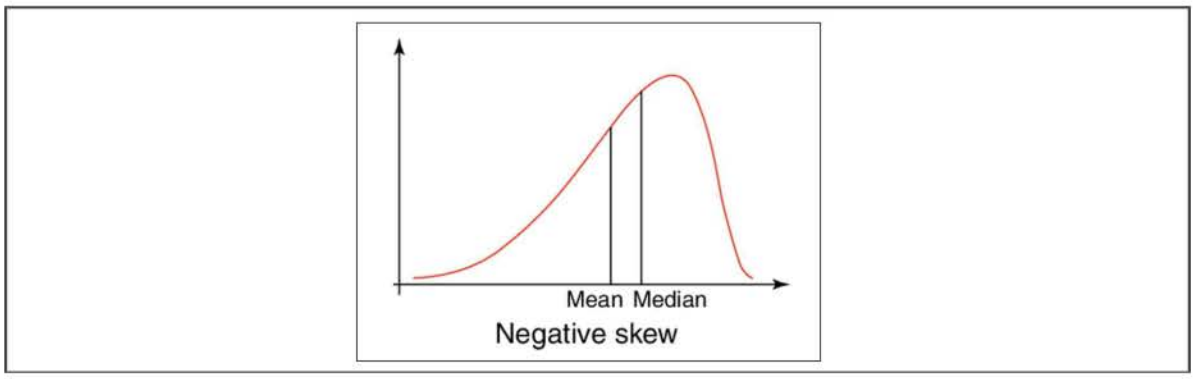
15.4.5: Understand Skew

Now that the team can test for normality, it's time to learn how to deal with datasets that aren't normal. Thankfully, Colleen knows how to do this and is using these mini lessons as a chance to practice for that big presentation for the CEO! So, on their next lunch break, they dig into the concept of skew.

When dealing with relatively smaller sample sizes, our data distributions are often asymmetrical. Compared to the normal distribution, where each tail of the distribution (on either side of the mean μ) mirrors one another, the asymmetrical distribution has one distribution tail that is longer than the other. This asymmetrical distribution is commonly referred to as a **skewed distribution** and there are two types—left skew and right skew.

Left Skew

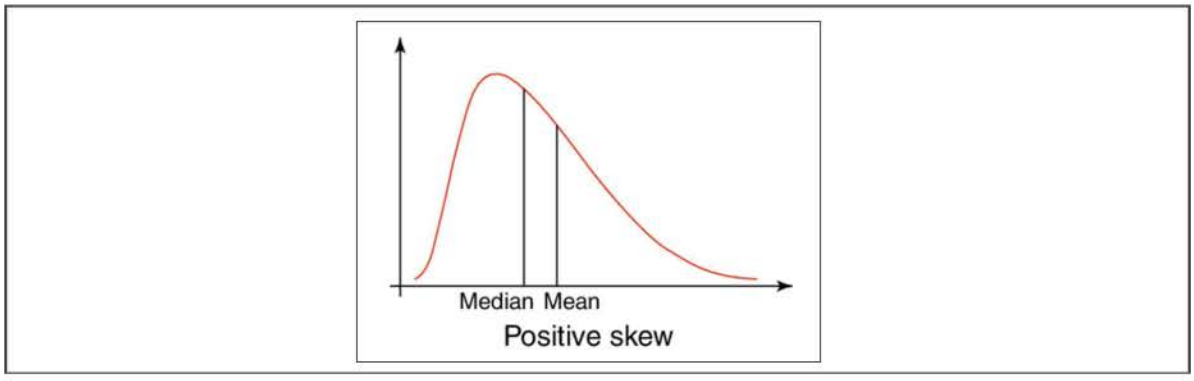
A data distribution is considered to be **left skewed,** or negative skewed, if the left tail is longer than the right, as shown below.



When data is skewed left, from the center of the distribution curve, there is a higher probability that extreme negative values exist within our dataset. When this occurs, the mean may no longer accurately reflect the central tendency of the data. Instead, we would use the median to describe the central tendency of the data. This skew is called negative skewed.

Right Skew

A data distribution is considered to be **right skewed**, or positive skewed, if the right tail is longer than the left, as shown below.



When data is skewed right, from the center of the distribution curve, there is a higher probability that extreme positive values exist within our dataset. Once again, if this occurs, we would use the median to describe the central tendency of the data. This skew is called positive skewed.

Manage Skewness

As with most problems in data analytics, we must approach skewness on a case-by-case basis. Depending on the severity of the skewness and the size of the dataset, there are multiple means of dealing (or not dealing) with skewness.

If our dataset is large, or the skewness is very subtle, we would simply point out that our data distribution shows signs of skew during reporting or presentation. In these cases, our mean and median will be roughly the same value, and there should be minimal impact to any downstream analysis.

If our dataset is smaller, or the skewness does impact the overall shape of our distribution, more action is needed. There are a few different things we can try:

- If possible, add more data points to our dataset to alleviate the effect of skew. However, this might not be possible or might not improve the distribution.
- Resample or regenerate data if we think that the data might not be representative of the original conditions or dataset.
- Transform our data values by normalization, using another numerical variable, or by transforming the data using an operator. The concept of transforming skewed data is very popular with scientists who deal with datasets where values can differ by orders of magnitude. One of the easiest means of transforming data is using a log-transform, where each value in the numeric dataset is transformed taking either natural log, or $\log_{10}$. By using a log-transformation, the effects of extreme values are reduced, and this transformation can help make each distribution tail more symmetrical.

> **Important**
>
> No matter what approach is used to help reduce the skewness of a dataset, it's good practice to disclose this information in a report or to use annotations on your results. This will help the reader understand the context surrounding any results, and it will make your analysis more credible.

15.5.1: Practice Hypothesis Testing

Now that Jeremy feels comfortable characterizing data from his colleagues, he wants to start practicing some basic statistics. He remembers a bit from his time in college, but it's been such a long time that Jeremy figures it's good to familiarize himself with some of the fundamentals before he tries to implement any statistical tests in R.

One of the largest and most critical concepts in statistics is hypothesis testing. In data science, we use statistical hypothesis testing to determine the probability of an event (or set of observations) under particular assumptions. In other words, we use statistical hypothesis testing to determine which of our hypotheses are most likely to be true. There are two types of statistical hypothesis:

- The **null hypothesis** is also known as H_0 and is generally the hypothesis that can be explained by random chance.
- The **alternate hypothesis** is also known as H_a and is generally the hypothesis that is influenced by non-random events.

By the end of this section, we should be able to generate a set of hypotheses and interpret the outcome of a statistical test. These concepts are universal and will apply to any statistical test, dataset, or analytical result.

The Importance of Hypothesis Testing

Although data collection and research are important, the backbone of the scientific method is **hypothesis testing**. Hypotheses are utilized by the scientific method to help narrow the scope of research and testing as well as provide a clear outcome of our results. Without generating a set of hypotheses, it becomes exponentially more difficult to quantify results and provide measurable outcomes to our analyses. As data analysts, it's our job to match a set of hypotheses to an appropriate statistical test to ensure that results are interpreted correctly.

Hypothesis Testing in Five Steps

Regardless of the complexity of the dataset or the proposed question, hypothesis testing uses the same five steps:

1. Generate a null hypothesis, its corresponding alternate hypothesis, and the significance level.
2. Identify a statistical analysis to assess the truth of the null hypothesis.
3. Compute the p-value using statistical analysis.
4. Compare p-value to the significance level.
5. Reject (or fail to reject) the null hypothesis and generate the conclusion.

Keep in mind that the null and alternate hypotheses are used to explain one of two outcomes from our proposed question, and both are mutually exclusive and exhaustive. In other words, no matter what, one of these statements must be used to explain our analysis results.

For example, perhaps we wanted to solve the question: "Is flipping a specific coin fair and balanced?" Given this question, our null hypothesis could be that the likelihood of flipping heads is the same as flipping tails. In other words, the likelihood of heads or tails can be totally explained by random chance. Our

alternative hypothesis might be that the likelihood of flipping heads is not the same as flipping tails. If we were to represent our hypotheses using mathematical symbols, it would be expressed as:

$H_0 : P_H = 0.5$

$H_a : P_H \neq 0.5$

Where P_H represents the probability of flipping heads.

> **Important**
> Notice that our null hypothesis represents the scenario that our results can be explained by random chance without any outside influence. In contrast, our alternate hypothesis represents any other scenario that our results could yield.

Once we established our null and alternate hypothesis, we would then need to identify a statistical analysis to assess if our null hypothesis is true. In this example, we could test our null hypothesis by flipping our own coin 50 times and then by calculating the probability of flipping heads.

> **Important**
> Depending on the complexity of the question and null hypothesis, calculating probability might be insufficient, and we might need to rely on more traditional statistical tests. Fret not, we'll cover many of the most popular statistical tests later in this module.

After determining which statistical analysis is most appropriate and analyzing our data, we must quantify our statistical results using probability. In our example, we are calculating probability directly; however, most statistical tests will produce probability in the form of a p-value.

The **p-value**, or probability value, tells us the likelihood that we would see similar results if we tested our data again, if the null hypothesis is true. Therefore, we use

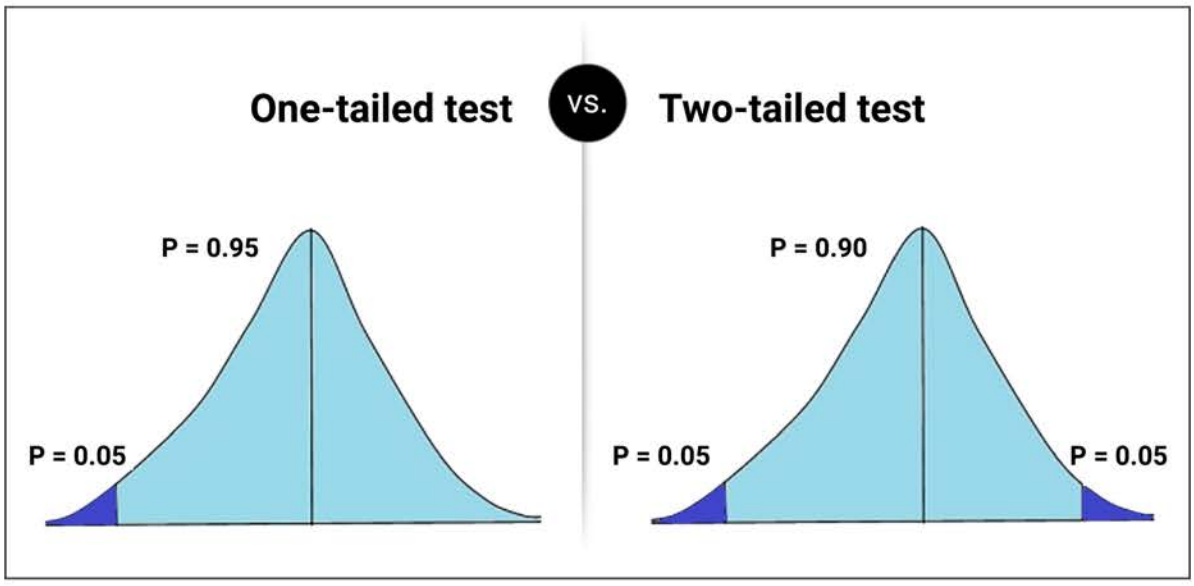
the p-value to provide quantitative evidence as to which of our hypotheses are true.

To determine which hypothesis is most likely to be true, we compare the p-value against a significance level. A **significance level** (also denoted as alpha or α) is a predetermined cutoff for our hypothesis test. When designing our hypothesis, we would determine the significance level based on the importance of our findings.

In most cases, a significance level of 0.05 is sufficient, but if our hypotheses are being used for critical decision-making (such as the performance of a drug or the durability of a helmet), we might want to use smaller cutoffs such as 0.01 or 0.001. Regardless of what significance level we select, we want to predetermine our cutoff prior to computing the p-value as to not introduce bias into our results. Refer to the following chart:

Importance of Findings	Significance Level	Probability of Being Wrong
Low	0.1	1 in 10
Normal	0.05	5 in 100
High	0.01	1 in 100
Very high	0.001	1 in 1,000
Extreme	0.0001	1 in 10,000

In addition to determining a significance level based on the importance of our findings, we must ensure our hypotheses and statistical tests are either one-tailed or two-tailed. The tails of our hypotheses or statistical tests are referring to the distribution of measurements or observations used in the analysis.



When it comes to hypothesis testing, a **one-tailed hypothesis** is only describing one side of the distribution. One-tailed hypotheses use descriptions such as “x is greater than y” or “x is less than or equal to y.” Alternatively, **two-tailed hypotheses** describe both sides of the distribution and use descriptions such as “equal to” or “not equal to.”

When it comes to checking our statistical tests, the documentation will tell us if our statistical test is one-tailed or two-tailed. Once we have determined the number of tails considered for both our hypotheses and statistical test, we can determine if we need to adjust our p-value:

- If our hypotheses and statistical test are both two-tailed, use the statistical test p-value as is.
- If our hypotheses are one-tailed, but our statistical test is two-tailed, divide the statistical test p-value by 2.

Once we have determined the significance level and the calculated p-value, we can complete our statistical analysis. If our calculated p-value is smaller than our significance level, we would state that there is sufficient statistical evidence that our null hypothesis is not true, and therefore we would reject our null hypothesis. Alternatively, if our calculated p-value is larger than our significance level, we would state that we do not have sufficient evidence to reject our null hypothesis, and therefore we fail to reject our null hypothesis.

> **Note**
>
> Outside of statistics, you may see others refer to this process as "accepting or rejecting" the null hypothesis. While this is not entirely untrue, many statisticians believe that there is no way of making a definitive choice between either hypothesis without an infinitely large dataset. Therefore, we use p-values and significance levels to determine *the probability* that our observations were obtained assuming the null hypothesis.

After we have determined which hypothesis is most likely to be true, we must conclude our statistical findings by relating our results back to the initial dataset or proposed question.

15.5.2: Assess Error in Hypothesis Testing

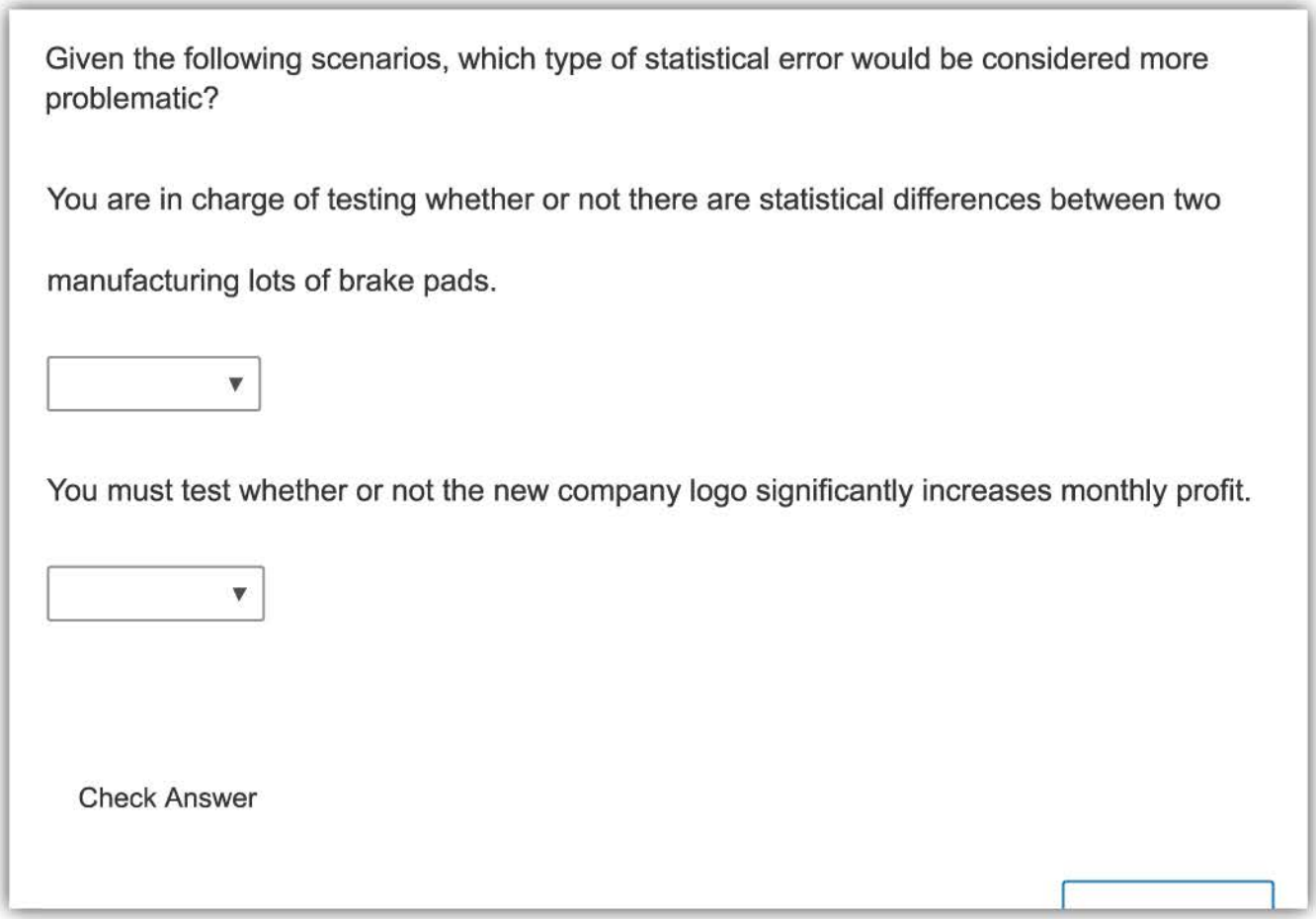
Jeremy is curious: what type of errors exist in hypothesis testing? He knows that the CEO will want to know all of the potential risks, so it's critical to get a handle on what errors could occur during this project phase.

In an ideal world, we would be able to definitively decide if our null hypothesis or alternative hypothesis was true by collecting all possible measurements or data points. But since this is often impossible, we must approximate truth using a subset of data.

In most cases, our approximations and hypothesis selection will be correct and represent the true real-world results. But due to the variability of data, at some point our hypothesis selection will be incorrect. Our incorrect hypothesis selection can fall into two categories:

- **Type I error** (also known as a **false positive** error)—an error in which we reject the null hypothesis when it is actually true. In other words, the observations and measurements used in our statistical test should have been attributed to random chance, but we attributed them to something else.
- **Type II error** (also known as a **false negative** error)—an error in which we fail to reject the null hypothesis when it is actually false. In other words, our analysis demonstrates that the observations were due to random chance, but they were not. The observations and measurements used in our statistical test failed to reflect an external force or influence to our problem.

While selecting the wrong hypothesis is never ideal, depending on our field of research or the importance of our proposed problem, one error type may be more problematic than the other.



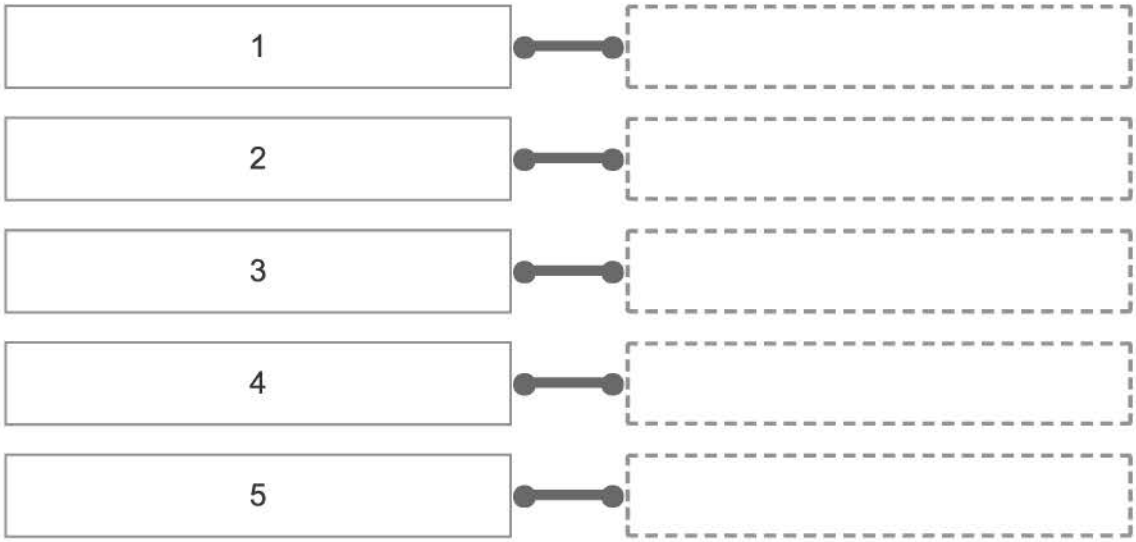
When it comes to limiting our type I and type II errors, there are two basic methods:

- A **type I error** can be limited by making your significance level smaller. A smaller significance level makes it harder to accidentally reject the null hypothesis when the data was truly random. This is also why our significance level (alpha or α) is sometimes referred to as our false positive rate.
- A **type II error** can be limited by increasing the power of the analysis. Although performing a power analysis is outside the scope of the course, power can be increased by adding additional measurements or observations to our analysis.

Now that we understand what a hypothesis test is, how to generate our hypothesis, and how to use the p-value to provide results and interpretations, we're

ready to dive into our first statistical test.

Put the hypothesis testing steps in the correct order.



- Compute the p-value using statistical analysis.
- Compare p-value to the significance level.
- Generate a null hypothesis, its corresponding alternate hypothesis, and the significance level.
- Reject (or fail to reject) the null hypothesis and generate the conclusion.
- Identify a statistical analysis to assess the truth of the null hypothesis.

Check Answer

15.6.1: Sample Versus Population Dataset

After brushing up on his statistics fundamentals, Jeremy is finally ready to start combining statistics with R. As part of his new job on the data analytics team, he'll have to perform retrospective analysis of historical vehicle data. This means Jeremy will have to know how to compare results and metrics across different groups and factors. Therefore, he needs to learn some statistical tests that will allow him to compare numerical variables.

In data analysis and statistics, an ideal dataset is one that contains measurements and results from every possible outcome, condition, or consideration. These datasets are known as a **population dataset** and contain all possible elements of a study or experiment.

Often, such an exhaustive dataset is prohibitively expensive or logistically impossible to generate. In this case, we must use a **sample** or subset of the population dataset, where not all elements of a study or experiment are collected or measured.

> **Caution**
> In data science, the concept of sample versus population does not strictly apply to people or animals. Any comprehensive dataset is considered a population, and any dataset that is a subset of a larger dataset is considered a sample.

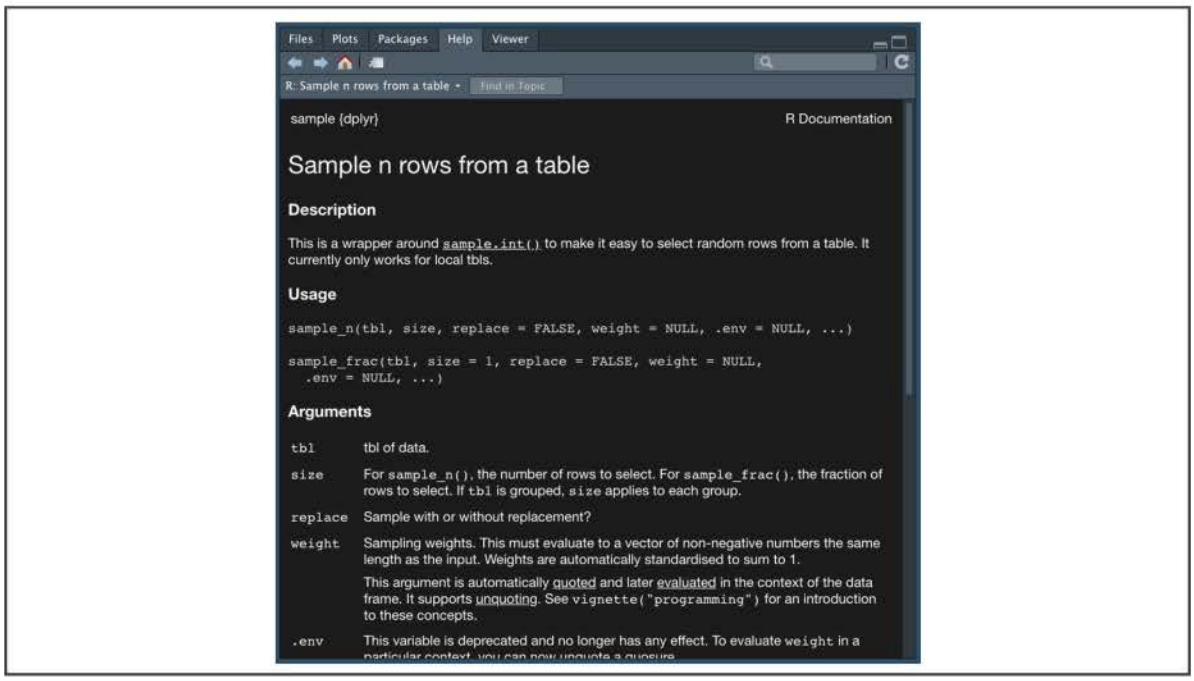
Since a sample dataset is just that—a sample—we must be clear about how a sample dataset represents the corresponding population data. One of the most straightforward ways to characterize a sample versus its population data is to compare the mean and standard deviation of both datasets. Ideally, a sample dataset will have a similar distribution to the population data, and therefore the mean and standard deviation would be about equal.

To produce a sample dataset that has a similar distribution to the population data, most statisticians suggest using random sampling. **Random sampling** is a technique in data science in which every subject or data point has an equal chance of being included in the sample. This technique increases the likelihood that even a small sample size will include individuals from each "group" within the population.

If performed using functions such as the built-in `sample()` function in R, or `sample_n()` function from dplyr, the resulting sample distributions should be similar to the input population data. When selecting sample data from a numerical vector, we should use the built-in `sample()` function. However, in most cases we will want to use the `sample_n()` function to select sample data from a two-dimensional data object.

Type the following code into the R console to look at the `sample_n()` documentation in the Help pane, listed under the subhead "Usage" in the image below:

```
>?sample_n()
```

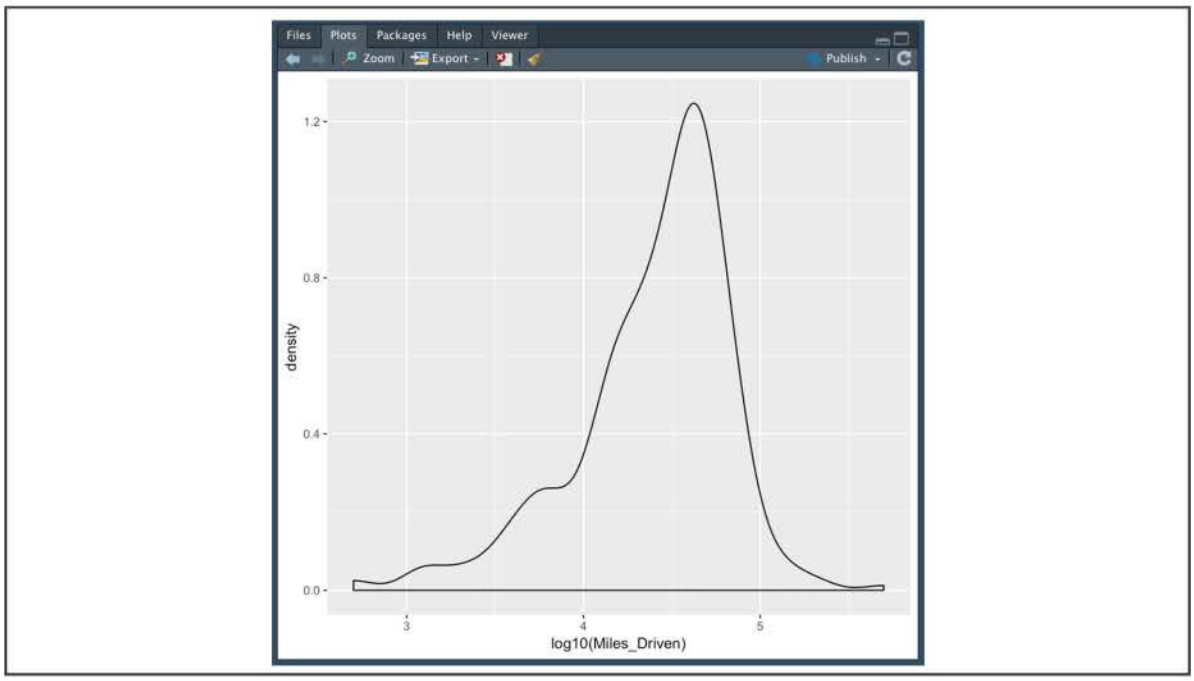


Using the `sample_n()` function only requires two arguments:

- **tbl** is the name of the input table, which is typically the name of a data frame. Optionally, we can use a dplyr pipe (%>%) to provide the data frame object directly, in which case, this argument is optional.
- **size** is the number of rows to return. As noted in the documentation, if we are providing a data frame that was grouped using the `group_by()` function, the **size** argument is the number of groups to return.

To practice generating samples using random sampling, download the **used vehicle dataset (https://courses.bootcampspot.com/courses/138/files/18457/download?wrap=1)** (https://courses.bootcampspot.com/courses/138/files/18457/download?wrap=1) and place it in your working directory. This dataset contains market data on more than 300 used vehicles. If we want to visualize the distribution of driven miles for our entire population dataset, we can use the `geom_density()` function from `ggplot2`:

```
> population_table <- read.csv('used_car_data.csv',check.names = F,string
> plt <- ggplot(population_table,aes(x=log10(Miles_Driven))) #import data
> plt + geom_density() #visualize distribution using density plot
```

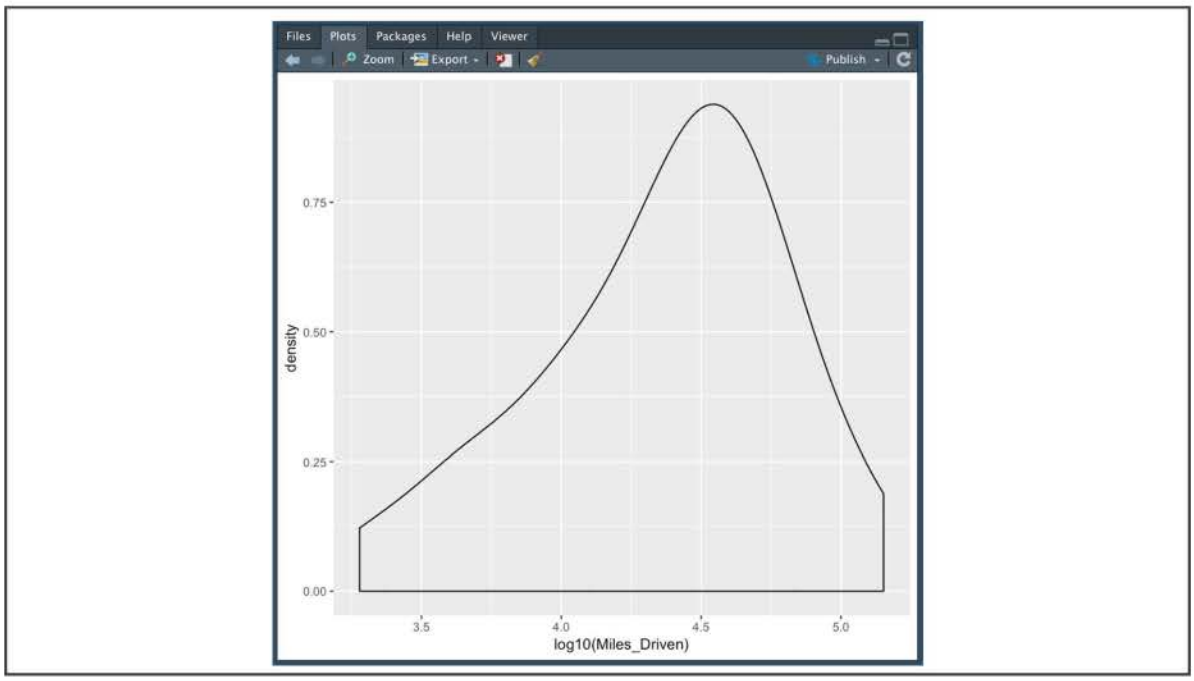


IMPORTANT

In this example, we want to transform our miles driven using a `log10` transformation. This is because the distribution of raw mileage is right skewed—a few used vehicles have more than 100,000 miles, while the majority of used vehicles have less than 50,000 miles. The `log10` transformation makes our mileage data more normal.

Now that we characterized our population data using our density plot, we'll create a sample dataset using dplyr's `sample_n()` function. Type the following code in the R console:

```
> sample_table <- population_table %>% sample_n(50) #randomly sample 50 c
> plt <- ggplot(sample_table,aes(x=log10(Miles_Driven))) #import dataset
> plt + geom_density() #visualize distribution using density plot
```



By using dplyr's `sample_n()` function, we can create a random sample dataset from our population data that contains minimal bias and (ideally) represents the population data.

Depending on the size of the population data, we may need to also adjust the size argument in our `sample_n()` function to ensure that our sample data is representative of the underlying population data. There are two basic ways to check that our sample data is representative of the underlying population: a qualitative assessment of each density plot or a quantitative statistical test such as the one-sample t-test.

15.6.2: Use the One-Sample t-Test

Jeremy is excited to dig into t-tests. He knows this is one of the most popular statistical tests in the world, and the CEO will definitely expect him to be able to do it!

The **Student's t-test** (most commonly referred to as **t-test**) is one of the most basic and popular statistical tests in the world. In statistics, we use a t-test to compare the mean of one dataset to another under a few assumptions.

There are two main forms of the t-test that we use: the **one-sample t-test** and the **two-sample t-test**. The one-sample t-test is used to assert if there is a statistical difference between the means of a sample dataset and hypothesized, potential population dataset. In other words, a one-sample t-test is used to test the following hypotheses:

- H_0 : There is **no statistical difference** between the observed sample mean and its presumed population mean.
- H_a : There is **a statistical difference** between the observed sample mean and its presumed population mean.

> **Note**
> We can also use a one-sided t-test by changing our alternative hypothesis to state that our sample mean is **significantly less** or **significantly more** than our presumed population mean.

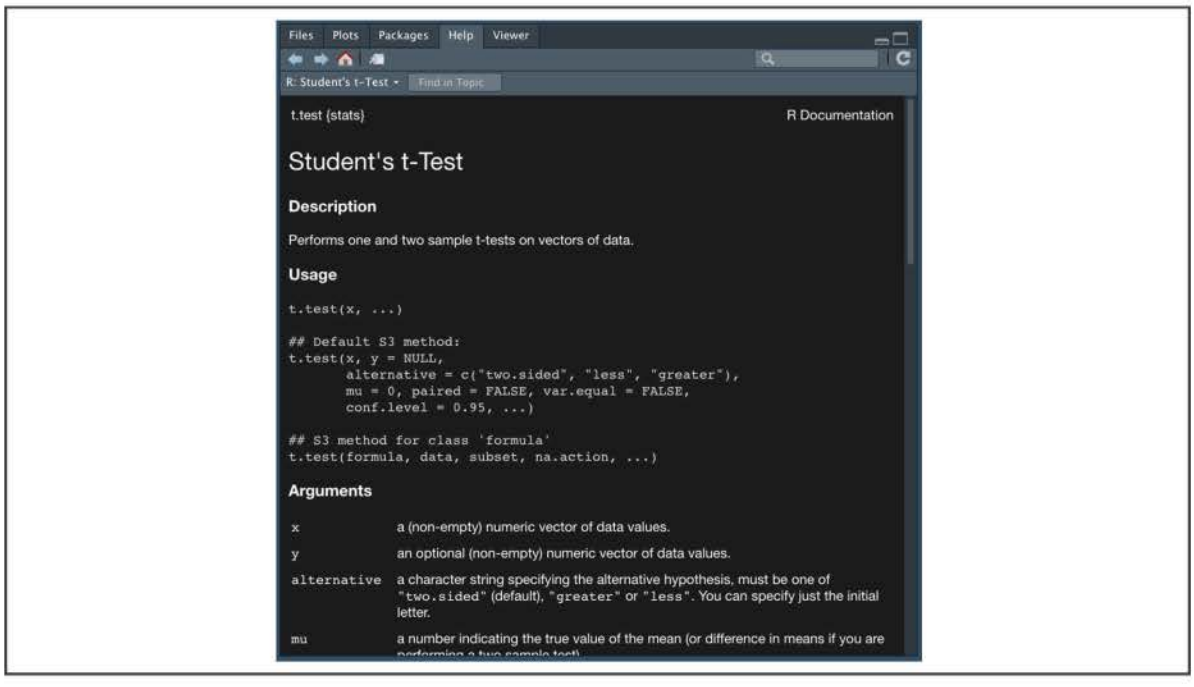
Before we can apply any statistical test to our data, we must check if there are any assumptions regarding our input dataset. When it comes to our one-sample t-test there are five assumptions about our input data:

1. The input data is numerical and continuous. This is because we are testing the distribution of two datasets.
2. The sample data was selected randomly from its population data.
3. The input data is considered to be normally distributed.
4. The sample size is reasonably large. Generally speaking, this means that the sample data distribution should be similar to its population data distribution.
5. The variance of the input data should be very similar.

As long as our input data satisfies (or mostly satisfies) the above assumptions, we can use the one-sample t-test to assert the similarities or differences in our data.

In R, we can implement a one-sample t-test using the built-in stats package `t.test()` function. Type the following code into the R console to look at the `t.test()` documentation in the Help pane:

```
> ?t.test()
```



To use the `t.test()` function to perform our one-sample t-test, we have to use a few arguments:

- **x**
- **mu**
- **alternative**

Looking at the R documentation for the `t.test()` function, fill in the name of the argument that corresponds to each of the following descriptions:

1. [ ] is the numeric vector of sample data.
2. [ ] is the calculated mean of the population data.
3. [ ] tells the `t.test()` function if the hypothesis is one-sided (one-tailed) or two-sided (two-tailed). The options for the alternative argument are "two.sided," "less," or "greater." By default, the `t.test()` function assumes a two-sided t-test.

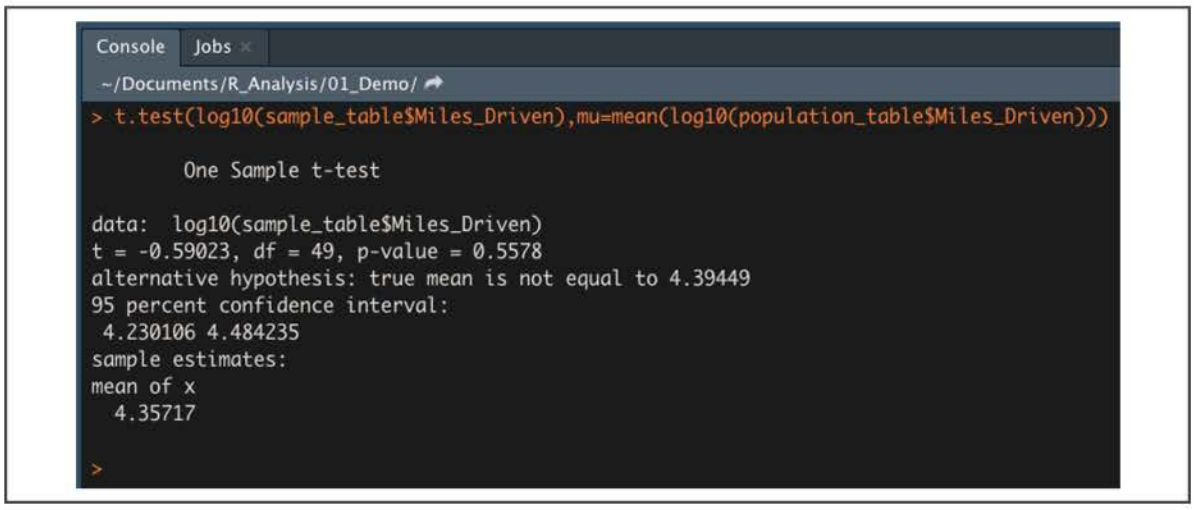
Check Answer

Finish ▸

By setting all three of these arguments, the `t.test()` function should produce our test statistic "t" along with our p-value, which we can use to evaluate our null hypothesis.

For example, if we want to test if the miles driven from our previous sample dataset is statistically different from the miles driven in our population data, we would use our `t.test()` function as follows:

```
>t.test(log10(sample_table$Miles_Driven),mu=mean(log10(population_table$M
```



There are a number of metrics produced from the `t.test()` function, but for now we will only concern ourselves with the calculated p-value. Assuming our significance level was the common 0.05 percent, our p-value is above our significance level. Therefore, we do not have sufficient evidence to reject the null hypothesis, and we would state that the two means are statistically similar.

> **Important**
>
> Due to random sampling, your sample dataset may differ from our example and thus your calculations may be different. Therefore, you'll need to compare your calculated p-value to your own significance level. If your p-value is lower than the significance level, you would have sufficient evidence to reject the null hypothesis and state that the two means are statistically different.

15.6.3: Use the Two-Sample t-Test

Now that Jeremy has conquered the one-sample t-Test, he's ready for the two-sample t-Test.

The second main form of the t-Test is a two-sample t-Test. Instead of testing whether a sample mean is statistically different from its population mean, the two-sample t-Test determines whether the means of two samples are statistically different. In other words, a two-sample t-Test is used to test the following hypotheses:

- H_0 : There is **no statistical difference** between the two observed sample means.
- H_a : There is **a statistical difference** between the two observed sample means.

There are also five assumptions regarding our input data when using the two-sample t-Test, which are the same as the one-sample t-Test:

1. The input data is numerical and continuous.
2. Each sample data was selected randomly from the population data.
3. The input data is considered to be normally distributed.
4. Each sample size is reasonably large. Generally speaking, this means that the sample data distribution should be similar to its population data distribution.
5. The variance of the input data should be very similar.

In R, we use the same `t.test()` function to calculate both a one-sample t-Test and two-sample t-Test. However, the two-sample t-Test arguments are slightly different:

- *x* is the first numeric vector of sample data.
- *y* is the second numeric vector of sample data.
- **alternative** tells the `t.test()` function if the hypothesis is one-sided (one-tailed) or two-sided (two-tailed). The options for the alternative argument are "two.sided,""less," or "greater." By default, the `t.test()` function assumes a two-sided t-Test.

Once we have provided the necessary numeric vectors for each sample, the `t.test()` function will calculate our two-sample t-Test and return the same output as before. As practice, let's test whether the mean miles driven of two samples from our used car dataset are statistically different.

First, we produce our two samples using the following R statements:

```
> sample_table <- population_table %>% sample_n(50) #generate 50 randomly
> sample_table2 <- population_table %>% sample_n(50) #generate another 50
```

True or False: Using sampling functions such as sample_n() function should not introduce bias.

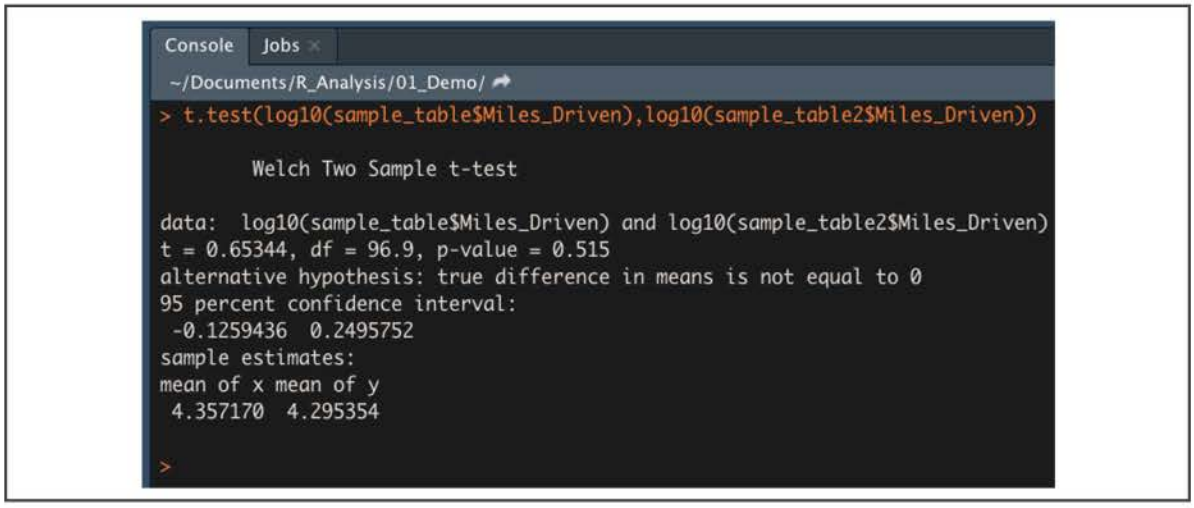
- True
- False

Check Answer

Finish ▸

Because our samples should not contain bias, we would expect our null hypothesis to be true—our samples should not be statistically different. To confirm, we'll use the `t.test()` function as follows:

```
> t.test(log10(sample_table$Miles_Driven),log10(sample_table2$Miles_Drive
```



Assuming a significance level of 0.05 percent. Is there sufficient evidence to reject the null hypothesis of the two-sample t-test?

- Reject the null hypothesis.
- Fail to reject the null hypothesis.

Check Answer

Finish ▸

15.6.4: Use the Two-Sample t-Test to Compare Samples

Jeremy can already tell that the two-sample t-test will be a standard part of his analytical procedure—especially when Colleen stops by and tells him you can actually use this test to compare samples from different populations!

In many cases, the two-sample t-test will be used to compare two samples from a single population dataset. However, two-sample t-tests are flexible and can be used for another purpose: to compare two samples from two different populations. This is known as a **pair t-test**, because we pair observations in one dataset with observations in another. We use the pair t-test when:

- Comparing measurements on the same subjects across a single span of time (e.g., fuel efficiency before and after an oil change)
- Comparing different methods of measurement (e.g., testing tire pressure using two different tire pressure gauges)

The biggest difference between paired and unpaired t-tests is how the means are calculated. In an unpaired t-test, the means are calculated by adding up all observations in a dataset, and dividing by the number of data points. In a paired t-test, the means are determined from the difference between each paired observation. As a result of the new mean calculations, our paired t-test hypotheses will be slightly different:

- H_0 : The **difference** between our paired observations (the true mean difference, or "μ_d") is **equal to zero**.

- H_a : The **difference** between our paired observations (the true mean difference, or "μ_d") is **not equal to zero**.

When it comes to implementing a paired t-test in R, we'll use the `t.test()` function.

Take a moment to look at the `t.test()` documentation in RStudio by typing `?t.test()` in the R console. Which arguments differ between an unpaired versus a paired t-test?

- [ ] x
- [ ] y
- [ ] paired
- [ ] alternative

Check Answer

Finish ▸

The required arguments are slightly different from the unpaired two-sample t-test:

- **x** is the first numeric vector of sample data.
- **y** is the second numeric vector of sample data.
- **paired** tells the `t.test()` function to perform a paired t-test. This value must be set to TRUE.
- **alternative** tells the `t.test()` function if the hypothesis is one-sided (one-tailed) or two-sided (two-tailed). The options for the alternative argument are "two.sided," "less," or "greater." By default, the `t.test()` function assumes a two-sided t-test.

To practice calculating a paired t-test in R, download the modified **mpg dataset (https://courses.bootcampspot.com/courses/138/files/18493/download?wrap=1)**  **(https://courses.bootcampspot.com/courses/138/files/18493/download?wrap=1)** and place it in your working directory. This data file contains a modified version of R's

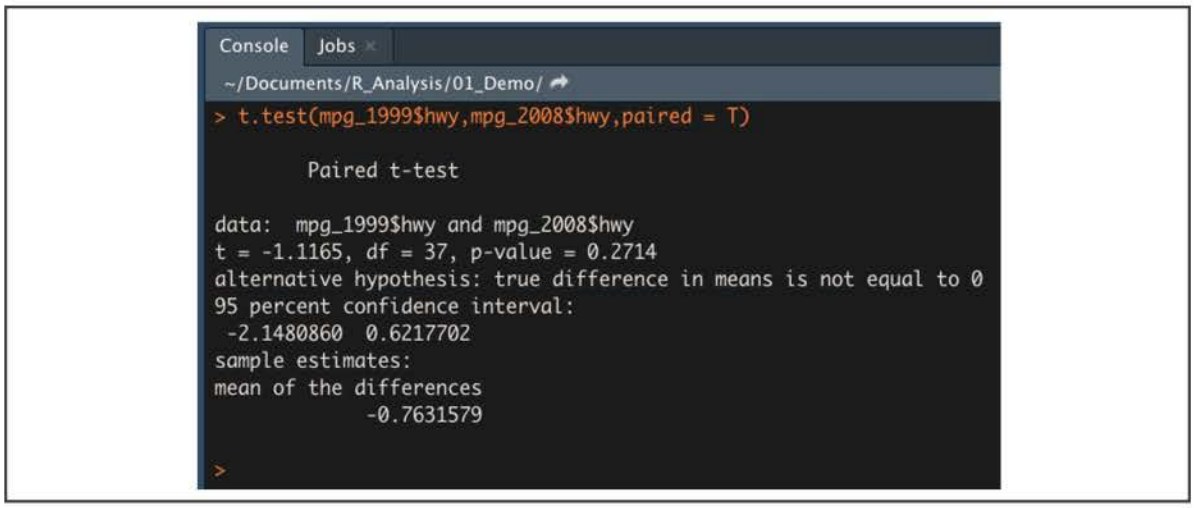
built-in mpg dataset, where each 1999 vehicle was paired with a corresponding 2008 vehicle.

First, let's generate our two data samples using the following code:

```
> mpg_data <- read.csv('mpg_modified.csv') #import dataset
> mpg_1999 <- mpg_data %>% filter(year==1999) #select only data points wh
> mpg_2008 <- mpg_data %>% filter(year==2008) #select only data points wh
```

Now that we have our paired datasets, we can use a paired t-test to determine if there is a statistical difference in overall highway fuel efficiency between vehicles manufactured in 1999 versus 2008. In other words, we are testing our null hypothesis—that the overall difference is zero. Using our `t.test()` function in R, our code would be as follows:

```
> t.test(mpg_1999$hwy,mpg_2008$hwy,paired = T) #compare the mean differer
```



Assuming a significance level of 0.05 percent. Is there sufficient evidence to reject the null hypothesis of the paired t-test?

- Reject the null hypothesis.
- Fail to reject the null hypothesis.

Check Answer

Finish ▸

15.6.5: Use the ANOVA Test

Woah! Two-sample t-tests can get complex pretty fast. Thankfully, Jeremy has one more trick up his sleeve—the analysis of variance (ANOVA) test.

When dealing with large real-world numerical data, we're often interested in comparing the means across more than two samples or groups. The most straightforward way to do this is to use the **analysis of variance (ANOVA) test,** which is used to compare the means of a continuous numerical variable across a number of groups (or factors in R).

Depending on your dataset and questions you wish to answer, an ANOVA can be used in multiple ways. For the purposes of this course, we'll concentrate on two different types of ANOVA tests:

- A **one-way ANOVA** is used to test the means of a single dependent variable across a single independent variable with multiple groups. (e.g., fuel efficiency of different cars based on vehicle class).
- A **two-way ANOVA** does the same thing, but for two different independent variables (e.g., vehicle braking distance based on weather conditions and transmission type).

Regardless of whichever type of ANOVA test we use, the statistical hypotheses of an ANOVA test are the same:

H_0 : The means of all groups are equal, or $\mu_1 = \mu_2 = \ldots = \mu_n$.

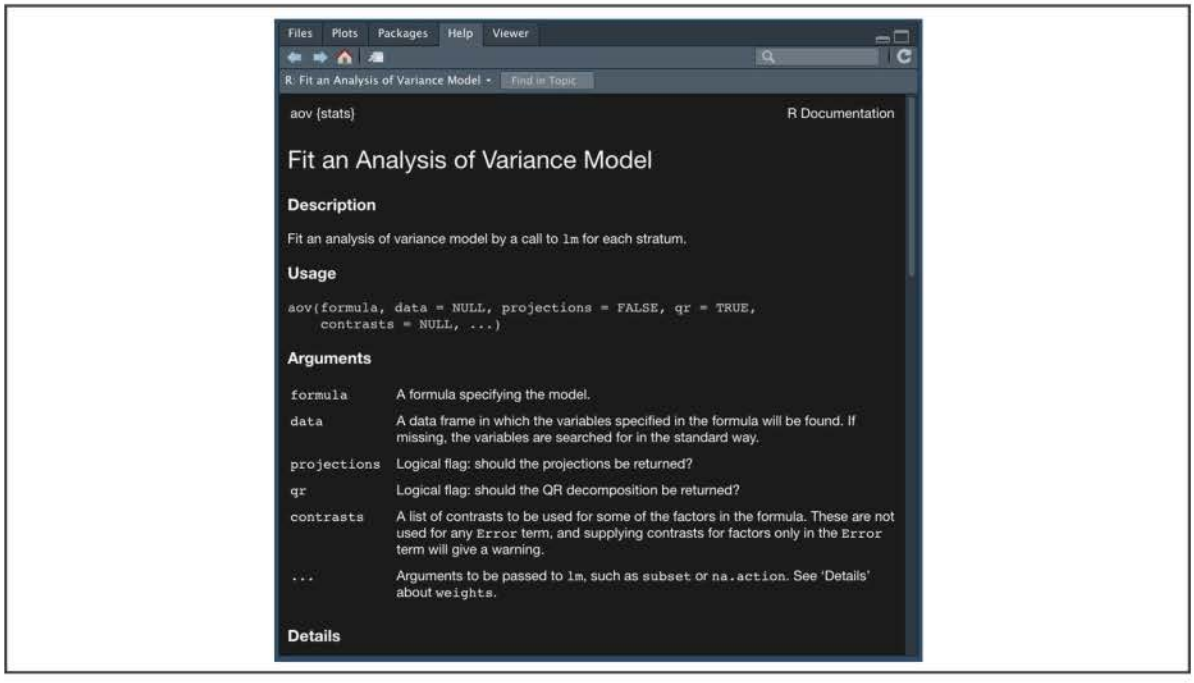
H_a : At least one of the means is different from all other groups.

Additionally, both ANOVA tests have assumptions about the input data that must be validated prior to using the statistical test:

1. The dependent variable is numerical and continuous, and the independent variables are categorical.
2. The dependent variable is considered to be normally distributed.
3. The variance among each group should be very similar.

In R, we can use the `aov()` function to perform both the one-way and two-way ANOVA test. Type the following code into the R console to look at the `aov()` documentation in the Help pane:

```
>?aov()
```



To perform an ANOVA test in R, we have to provide the `aov()` function two arguments:

- **formula**
- **data**

Looking at the R documentation for the aov() function, fill in the name of the argument that corresponds to each of the following descriptions:

1. [ ] is a special statement in R that tells the aov() function how to interpret the different variables and factors. In most cases, we'll use the formula Y ~ A or Y ~ A + B where Y is the column name of the dependent variable, and A and B are the column names of the independent variables.
2. [ ] is the name of our input data frame. The data frame should contain columns for each variable.

Check Answer

Unlike the `t.test()` function, where each group was a separate numeric vector, the `aov()` function expects that all of the observations and grouping information are contained within a single data frame. Once we have our cleaned and labeled data frame, we're ready to perform our ANOVA test using the `aov()` function.

To practice our one-way ANOVA, return to the mtcars dataset. For this statistical test, we'll answer the question, "Is there any statistical difference in the horsepower of a vehicle based on its engine type?"

In this case, we will use the "hp" and "cyl" columns from our mtcars dataset:

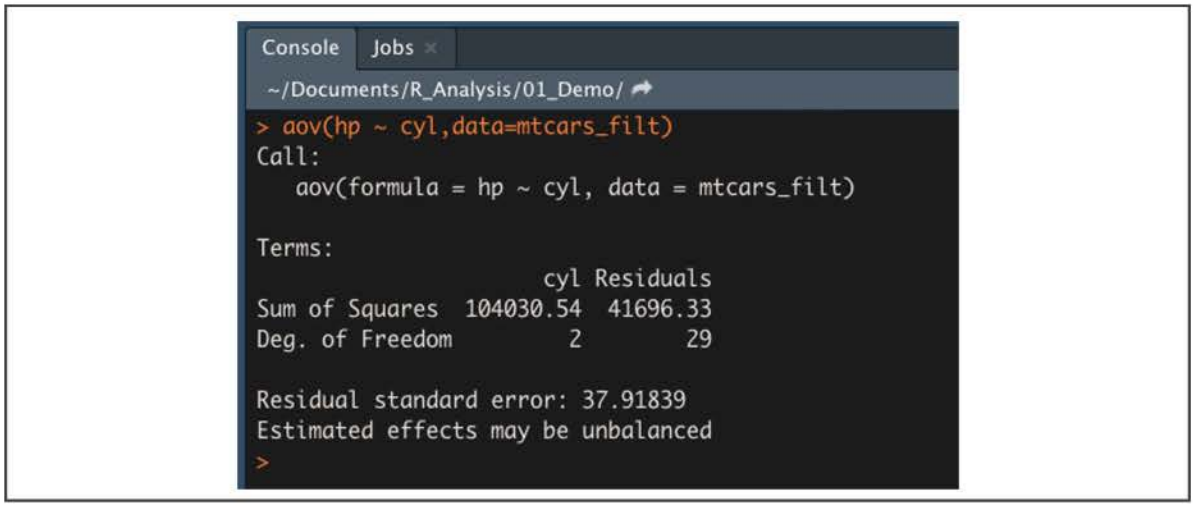
- horsepower (the "hp" column) will be our dependent, measured variable
- number of cylinders (the "cyl" column) will be our independent, categorical variable.

However, in the mtcars dataset, the `cyl` is considered a numerical interval vector, not a categorical vector. Therefore, we must clean our data before we begin, using the following code:

```
> mtcars_filt <- mtcars[,c("hp","cyl")] #filter columns from mtcars datas
> mtcars_filt$cyl <- factor(mtcars_filt$cyl) #convert numeric column to f
```

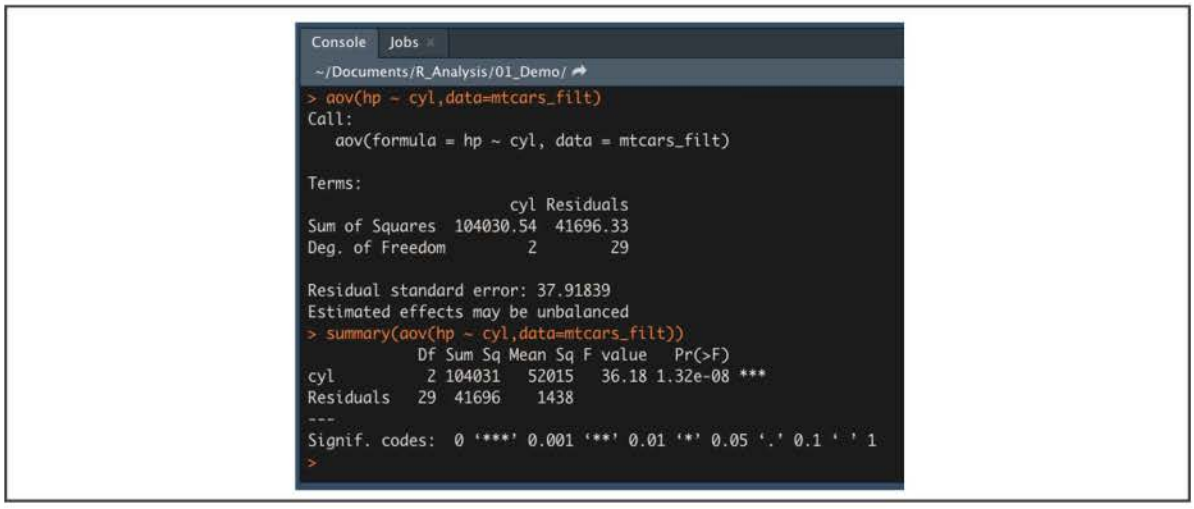
Now that we have our cleaned dataset, we can use our `aov()` function as follows:

```
> aov(hp ~ cyl,data=mtcars_filt) #compare means across multiple levels
```



Due to the fact that the ANOVA model is used in many forms, the initial output of our `aov()` function does not contain our p-values. To retrieve our p-values, we have to wrap our `aov()` function in a `summary()` function as follows:

```
> summary(aov(hp ~ cyl,data=mtcars_filt))
```



When using the formula statement, each independent variable will be shown as a separate row, with an additional “Residuals” row that tells us what the residual error is for our ANOVA model. For our purposes, we are only concerned with the “Pr(>F)” column, which is the same as our p-value statistic.

Depending on how small our p-value is, there may be symbols on the right side that indicate which significance level the p-value is below. In this case, our p-value is 1.32×10^{-8}, which is much smaller than our assumed 0.05 percent significance level. Therefore, we would state that there is sufficient evidence to reject the null hypothesis and accept that there is a significant difference in horsepower between at least one engine type and the others.

Now that you have learned the differences between our t-tests and ANOVA tests, you're ready to analyze data and perform statistical tests when comparing means. Feel free to explore more datasets and practice implementing analysis of means on your own.

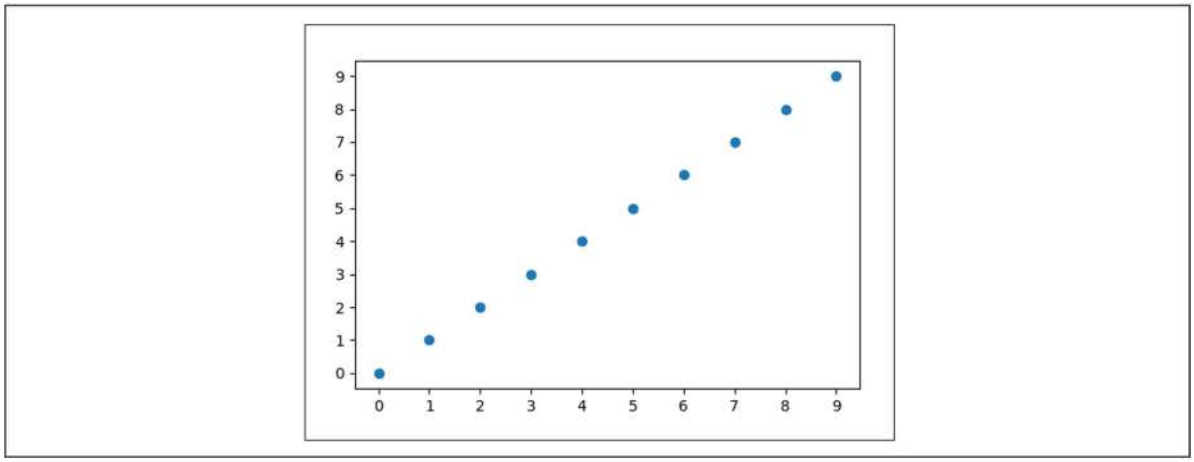
15.7.1: The Correlation Conundrum

Jeremy has finally started to make the connections between his programming experience with some statistical concepts. But this is only the beginning; comparing and contrasting data is only one statistical concept. Another big component to his new job will be to identify patterns in data and generate predictive models. Jeremy has a little experience in generating trendlines in plots, but he has no way to quantify how well these trend lines will perform when it comes time for decision making. Jeremy realizes that he must go back and learn more statistical tests that will help him quantify the patterns and models in his data.

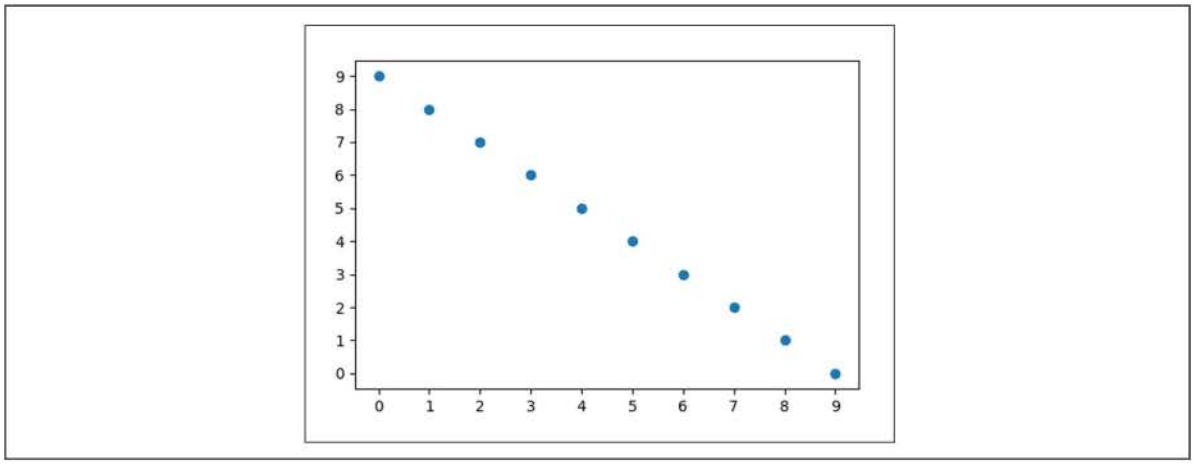
In data analytics, we'll often ask the question “is there any relationship between variable A and variable B?” This concept is known in statistics as correlation. **Correlation analysis** is a statistical technique that identifies how strongly (or weakly) two variables are related.

Correlation is quantified by calculating a **correlation coefficient**, and the most common correlation coefficient is the Pearson correlation coefficient. The **Pearson correlation coefficient** is denoted as “r” in mathematics and is used to quantify a linear relationship between two numeric variables. The Pearson correlation coefficient ranges between -1 and 1, depending on the direction of the linear relationship.

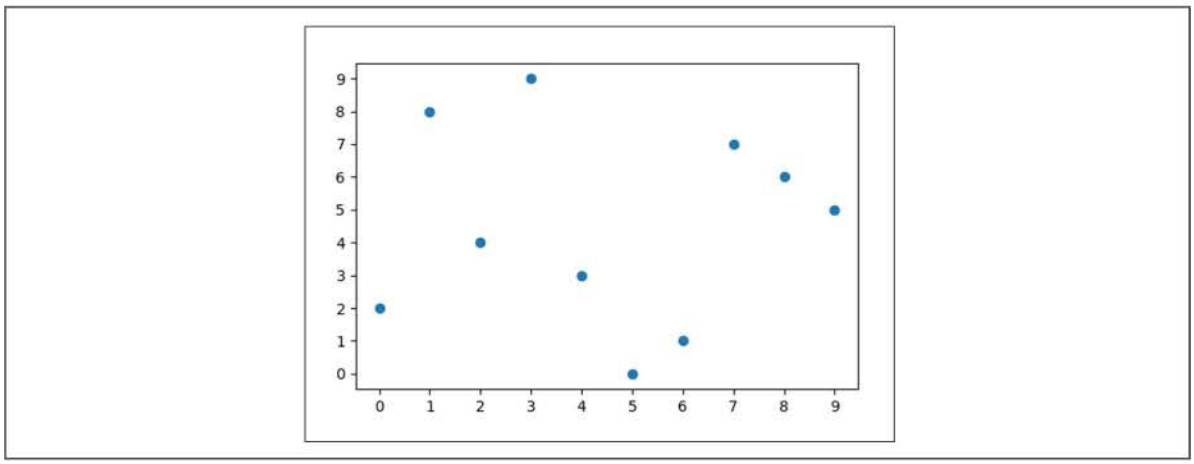
The following image is an example of an **ideal positive correlation** where r = 1. When two variables are positively correlated, they move in the same direction. In other words, when the variable on the x-axis increases, the variable on the y-axis increases as well:



The following image is an example of an **ideal negative correlation** where r = -1. When two variables are negatively correlated, they move in opposite directions. In other words, when the variable on the x-axis increases, the variable on the y-axis decreases.



The following image is an example of two variables with **no correlation** where r ≈ 0. When two variables are not correlated, their values are completely independent between one another.

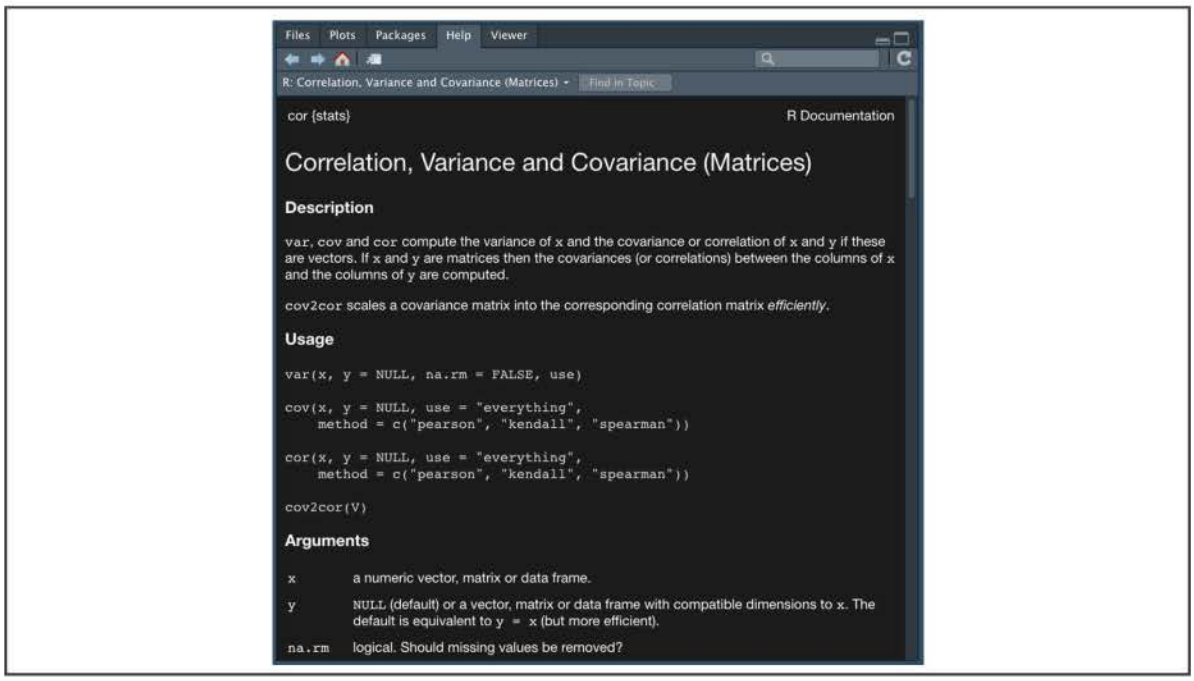


For real-world data, it can be very difficult to determine if two variables are correlated, so we must use the Pearson correlation coefficient to calculate the correlation strength. Refer to the table below.

Absolute Value of r	Strength of Correlation
$r < 0.3$	None or very weak
$0.3 \leq r < 0.5$	Weak
$0.5 \leq r < 0.7$	Moderate
$r \geq 0.7$	Strong

In R, we can use our `geom_point()` plotting function combined with the `cor()` function to quantify the correlation between variables. Type the following code into the R console to look at the `cor()` documentation in the Help pane:

```
>?cor()
```

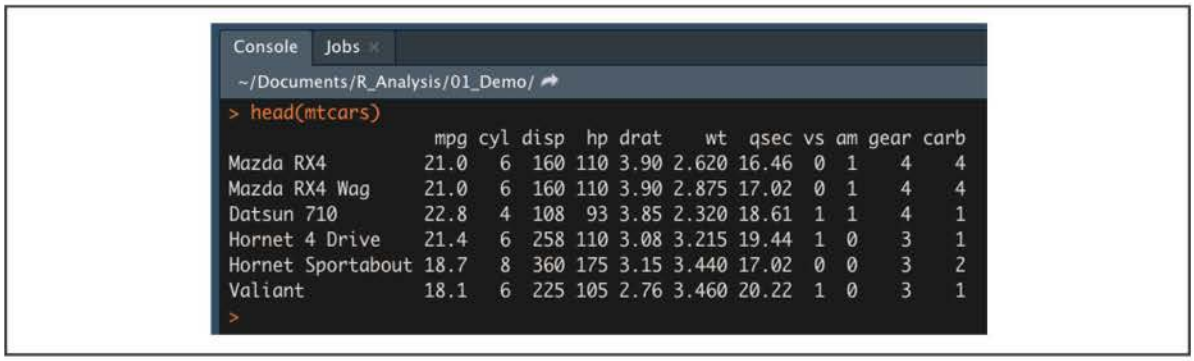


To use the `cor()` function to perform a correlation analysis between two numeric variables, we need to provide the following arguments:

- **x** is the first variable, which would be plotted on the x-axis.
- **y** is the second variable, which would be plotted on the y-axis.

As long as we are using two numeric variables, there are no other assumptions regarding our input data. To practice calculating the Pearson correlation coefficient, we'll use the mtcars dataset. Type the following in the R console:

```
> head(mtcars)
```

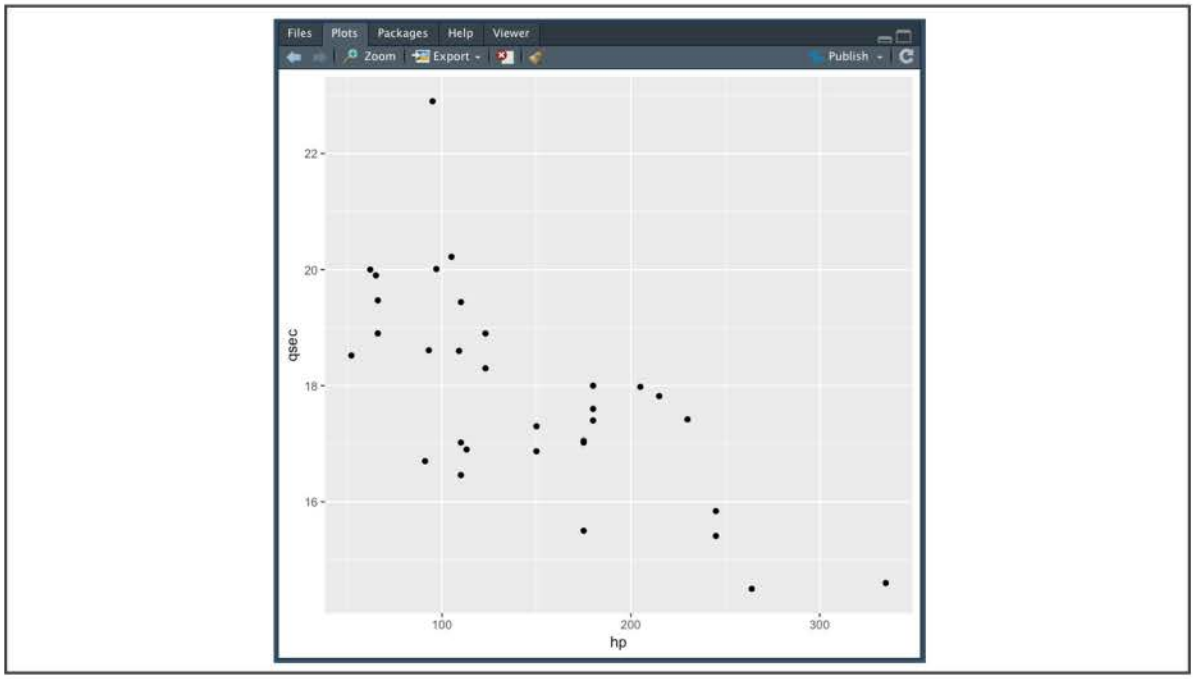


In the mtcars dataset, there are a number of numeric columns that we can use to test for correlation such as `mpg`, `disp`, `hp`, `drat`, `wt`, and `qsec`. For our

example, we'll test whether or not horsepower (`hp`) is correlated with quarter-mile race time (`qsec`).

First, let's plot our two variables using the `geom_point()` function as follows:

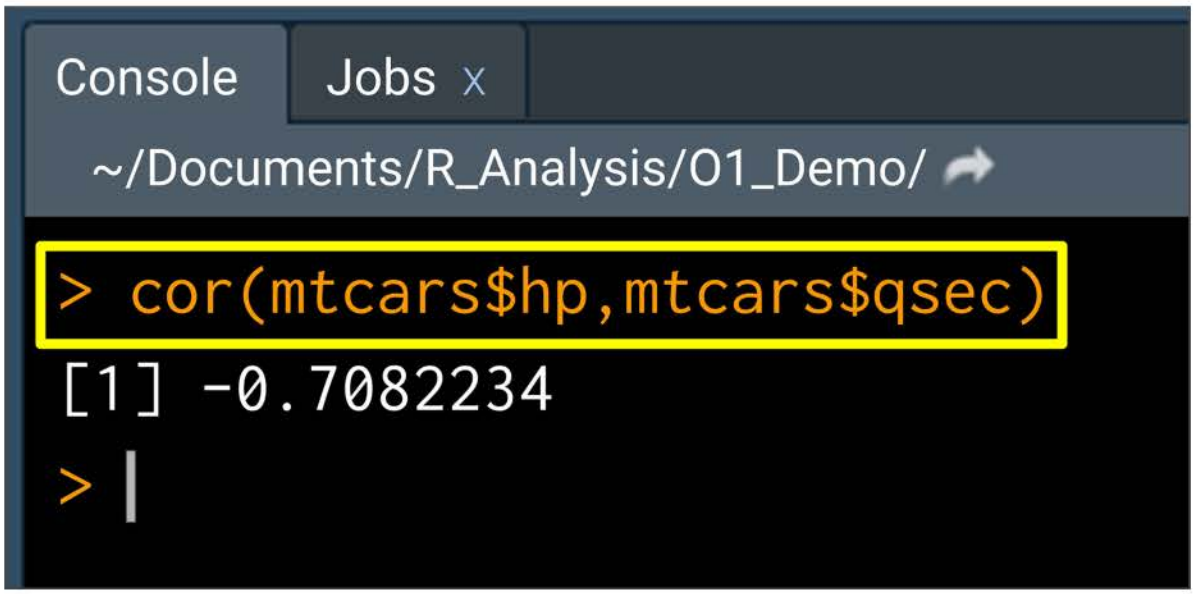
```
> plt <- ggplot(mtcars,aes(x=hp,y=qsec)) #import dataset into ggplot2
> plt + geom_point() #create scatter plot
```



Looking at our plot, it appears that the quarter-mile time is negatively correlated with horsepower. In other words, as vehicle horsepower increases, vehicle quarter-mile time decreases.

Next, we'll use our `cor()` function to quantify the strength of the correlation between our two variables:

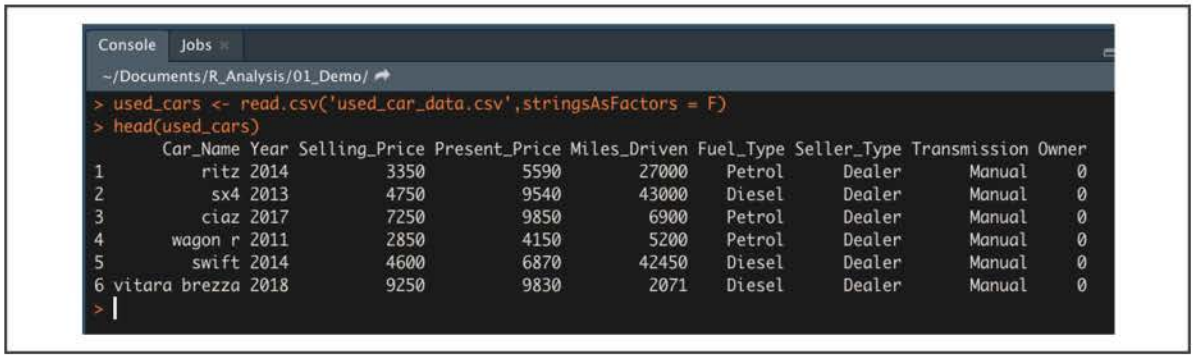
```
> cor(mtcars$hp,mtcars$qsec) #calculate correlation coefficient
```



From our correlation analysis, we have determined that the r-value between horsepower and quarter-mile time is -0.71, which is a strong negative correlation.

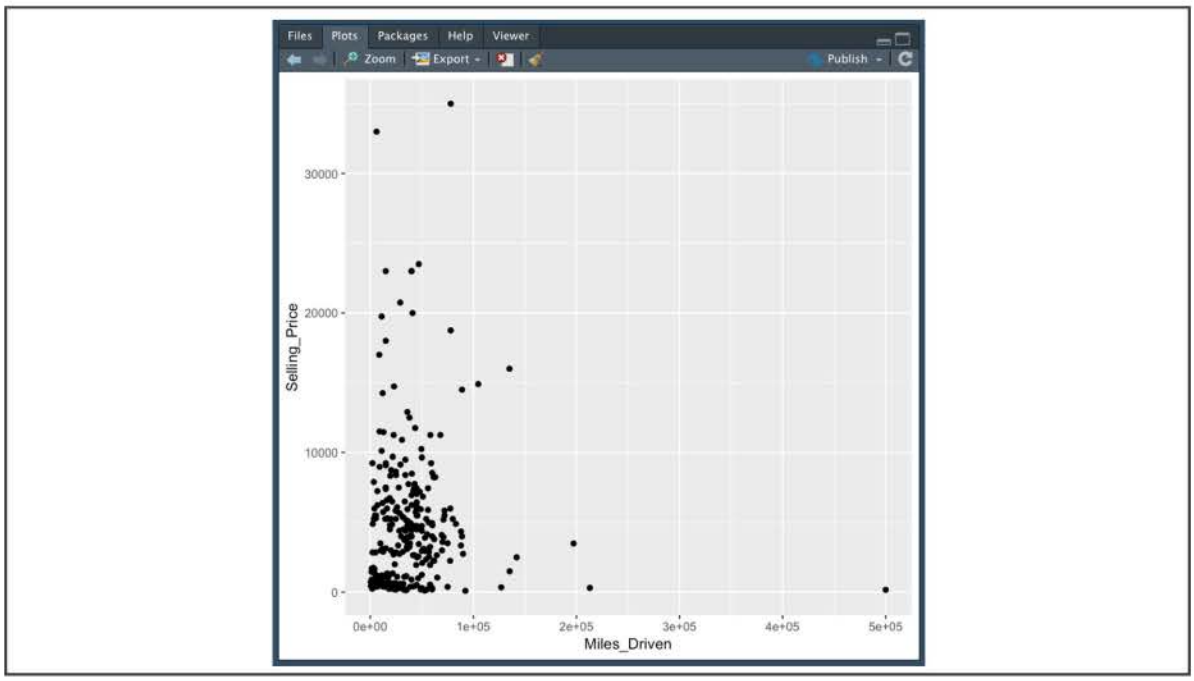
For another example, let's reuse our `used_cars` dataset:

```
> used_cars <- read.csv('used_car_data.csv',stringsAsFactors = F) #read i
> head(used_cars)
```



For this example, we'll test whether or not vehicle miles driven and selling price are correlated. Once again, we'll plot our two variables using the `geom_point()` function:

```
> plt <- ggplot(used_cars,aes(x=Miles_Driven,y=Selling_Price)) #import da
> plt + geom_point() #create a scatter plot
```



Compared to our previous example, our scatter plot did not help us determine whether or not our two variables are correlated. However, let's see what happens if we calculate the Pearson correlation coefficient using the `cor()` function:

```
> cor(used_cars$Miles_Driven,used_cars$Selling_Price) #calculate correlat
```

Our calculated r-value is 0.02, which means that there is a negligible correlation between miles driven and selling price in this dataset.

In most cases, we'll use correlation analysis as a means of exploring data and looking for trends. Although we can calculate the correlation of each pair of numerical variables in a dataset, this process can be highly time-consuming.

Instead of computing each pairwise correlation, we can use the `cor()` function to produce a correlation matrix. A **correlation matrix** is a lookup table where the variable names of a data frame are stored as rows and columns, and the intersection of each variable is the corresponding Pearson correlation coefficient. We can use the `cor()` function to produce a correlation matrix by providing a matrix of numeric vectors.

For example, if we want to produce a correlation matrix for our used_cars dataset, we would first need to select our numeric columns from our data frame and convert to a matrix. Then we can provide our numeric matrix to the `cor()` function as follows:

```
> used_matrix <- as.matrix(used_cars[,c("Selling_Price","Present_Price",'
> cor(used_matrix)
```

If we look at the correlation matrix using either rows or columns, we can identify pairs of variables with strong correlation (such as selling price versus present price), or no correlation (like our previous example of miles driven versus selling price).

The correlation matrix is a very powerful data exploration tool that allows an analyst to scan large numerical datasets for variables of interest. Once the variables of interest have been identified, the analyst can move on to more rigorous data analysis and hypothesis testing.

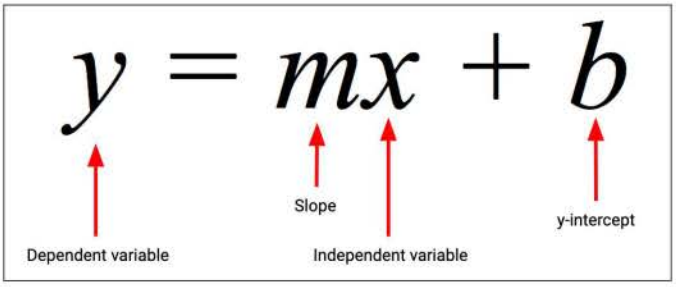
15.7.2: Return to Linear Regression

Jeremy is really starting to hit his stride. So, when his team wants to figure out how to predict one variable given another, he's pretty excited that he knows the answer: use linear regression!

Of course, before he actually uses the technique, he wants to dig into just exactly how it works. After all, he's leading a team into new territory, and it's up to him to make sure they are headed in the right direction.

Earlier we learned about linear regression and how it can be used to determine our dependent *y* variable from an independent *x* variable. In a more formal definition, **linear regression** is a statistical model that is used to predict a continuous dependent variable based on one or more independent variables fitted to the equation of a line.

In school, most students learned that the equation of a line is written as $y = mx + b$:



The job of a linear regression analysis is to calculate the slope and *y* intercept values (also known as coefficients) that minimize the overall distance between each data point from the linear model. There are two basic types of linear regression:

- **Simple linear regression** builds a linear regression model with one independent variable.
- **Multiple linear regression** builds a linear regression model with two or more independent variables.

Linear regression is popular in data science because it has multiple applications. First and foremost, linear regression can be used as a predictive modeling tool where future observations and measurements can be predicted and extrapolated from a linear model. Linear regression can also be used as an exploratory tool to quantify and measure the variability of two correlated variables.

A good linear regression model should approximate most data points accurately if two variables are strongly correlated. In other words, linear regression can be used as an extension of correlation analysis. Compared to correlation analysis, which asks if there is any relationship between variable A and variable B, linear regression asks if we can predict values from variable A using a linear model and values from variable B.

To answer this question, linear regression tests the following hypotheses:

H_0 : The slope of the linear model is zero, or $m = 0$

H_a : The slope of the linear model is not zero, or $m \neq 0$

If there is no significant linear relationship, each dependent value would be determined by random chance and error. Therefore, our linear model would be a flat line with a slope of 0.

To quantify how well our linear model can be used to predict future observations, our linear regression functions will calculate an r-squared value. The **r-squared (r^2) value** is also known as the coefficient of determination and represents how well the regression model approximates real-world data points. In most cases, the

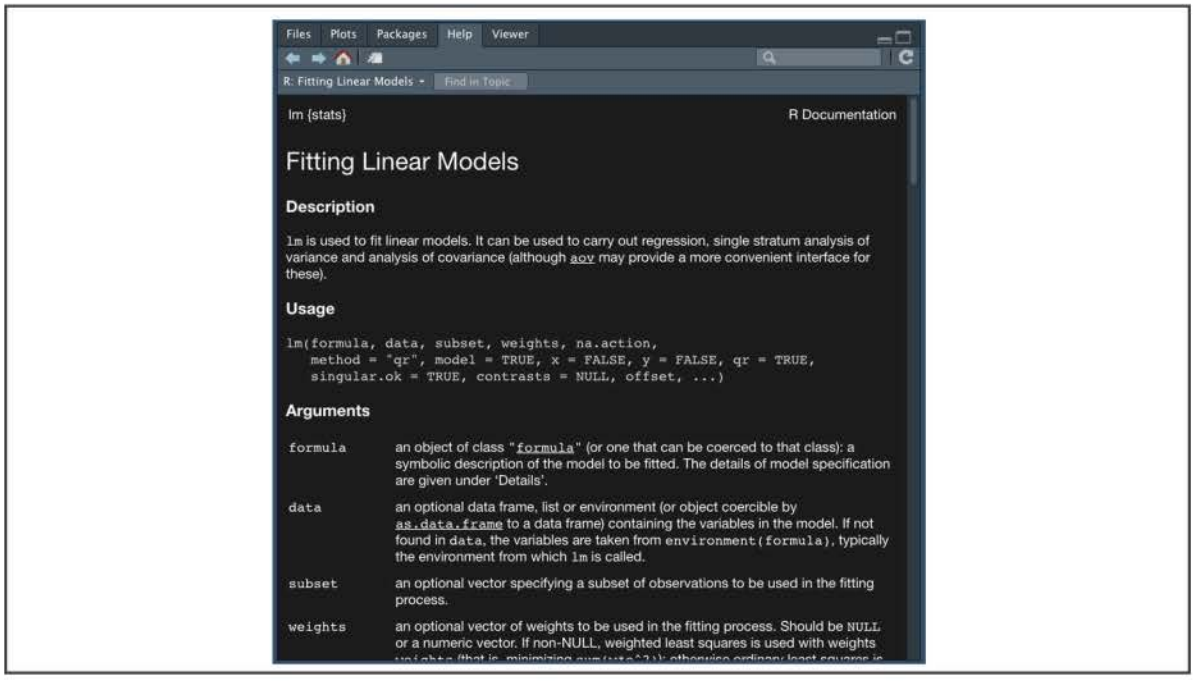
r-squared value will range between 0 and 1 and can be used as a probability metric to determine the likelihood that future data points will fit the linear model.

When using a simple linear regression model, the r-squared metric can be approximated by calculating the square of the Pearson correlation coefficient between the two variables of interest.

By combining the p-value of our hypothesis test with the r-squared value, the linear regression model becomes a powerful statistics tool that both quantifies a relationship between variables and provides a meaningful model to be used in any decision-making process.

Although the interpretation of a simple linear regression is different from a multiple linear regression, their model implementation is the same. In R, we'll build our linear models using the built-in `lm()` function. Type the following code into the R console to look at the `lm()` documentation in the Help pane:

```
>?lm()
```



Even though there are many optional arguments for the `lm()` function, the `lm()` function only requires us to provide two arguments:

- **formula**

- **data**

Looking at the R documentation for the `lm()` function, fill in the name of the argument that corresponds to each of the following descriptions:

1. [ ] is the same R statement that we use for the `aov()` function.

The formula statement tells R how to interpret the different variables and factors. With simple linear regression, we'll use the formula `Y ~ A` where `Y` is the column name of the dependent variable, and `A` is the column name of the independent variable.

2. [ ] is the name of our input data frame. The data frame should contain columns for each variable.

Check Answer

Similar to our t-test analysis, there are a few assumptions about our input data that must be met before we perform our statistical analysis:

1. The input data is numerical and continuous.
2. The input data should follow a linear pattern.
3. There is variability in the independent *x* variable. This means that there must be more than one observation in the x-axis and they must be different values.
4. The residual error (the distance from each data point to the line) should be normally distributed.

> **Important**
> Validating the fourth assumption is outside the scope of this course as it involves more robust statistical methods. However, in most real-world cases, we can expect our data to meet the fourth assumption.

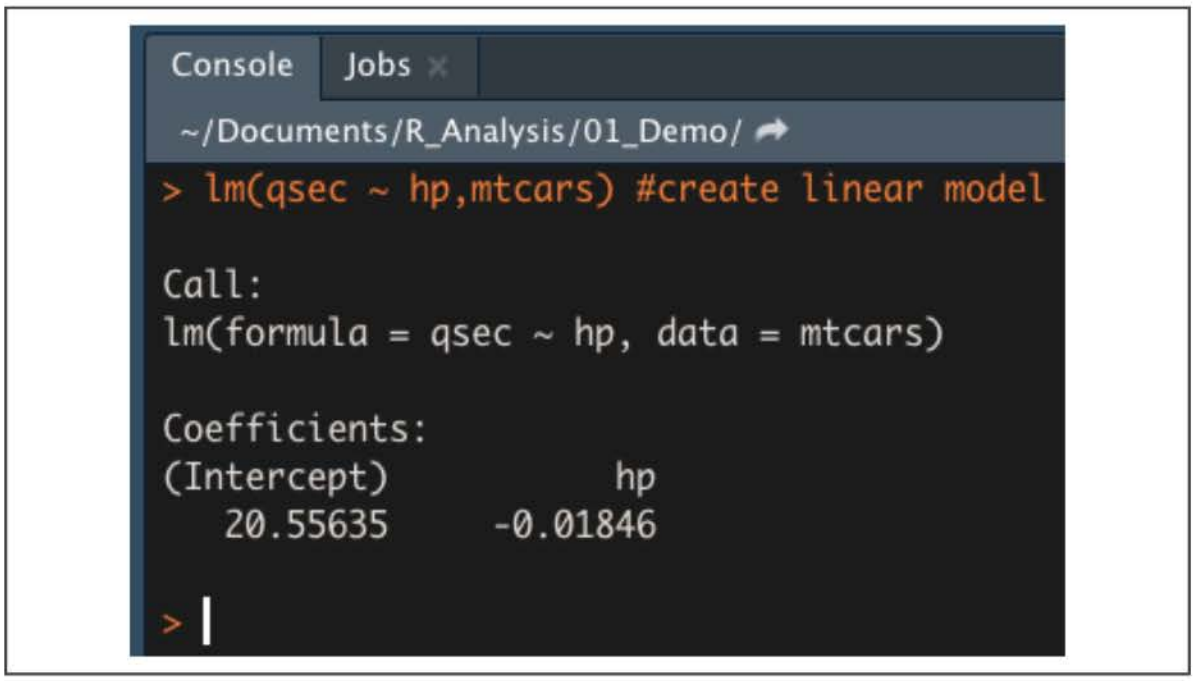
Once we have our data in a single data frame that meets the assumptions of our linear regression analysis, we're ready to implement the `lm()` function.

For practice, let's revisit our correlation example using the mtcars dataset. Using our simple linear regression model, we'll test whether or not quarter-mile race time (`qsec`) can be predicted using a linear model and horsepower (`hp`).

Remember from our correlation example that our Pearson correlation coefficient's r-value was -0.71, which means there is a strong negative correlation between our variables. Therefore, we anticipate that the linear model will perform well.

To create a linear regression model, our R statement would be as follows:

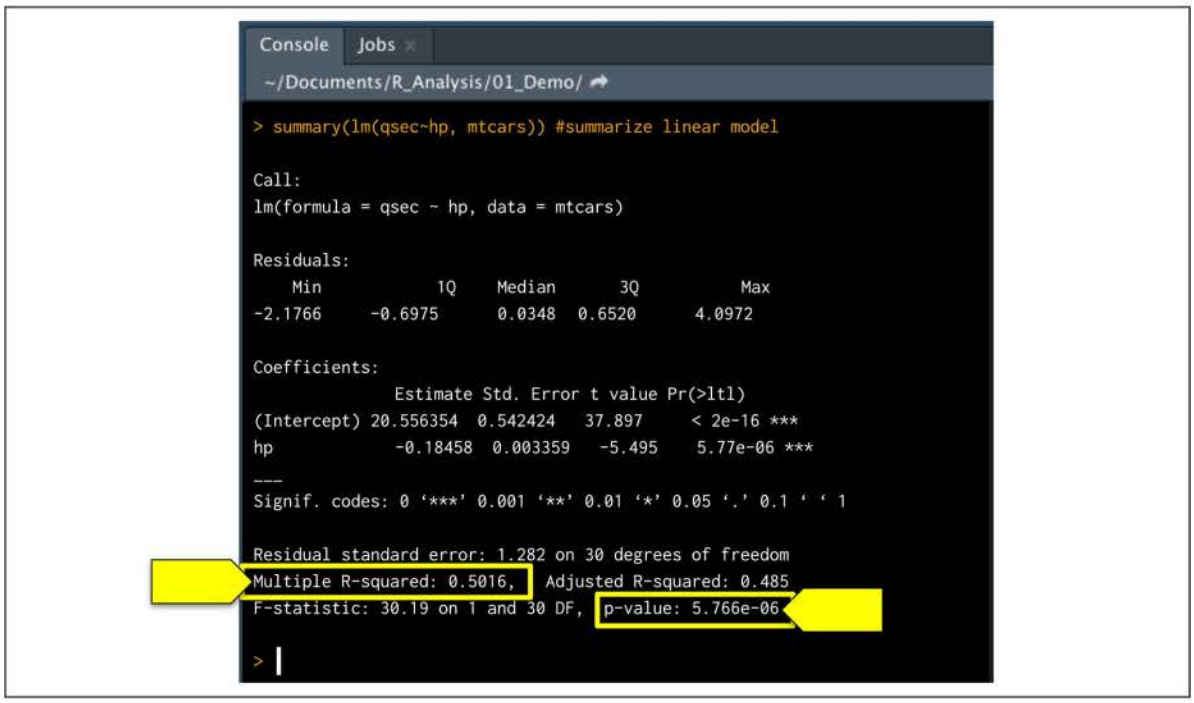
```
> lm(qsec ~ hp,mtcars) #create linear model
```



The output of the `lm()` function will be the metrics from our model. Specifically, the `lm()` function returns our y intercept (`Intercept`) and slope (`hp`) coefficients. Therefore, the linear regression model for our dataset would be `qsec = -0.02hp + 20.56`.

To determine our p-value and our r-squared value for a simple linear regression model, we'll use the `summary()` function:

```
> summary(lm(qsec~hp,mtcars)) #summarize linear model
```



Although there are a number of quantitative metrics produced by the `summary(lm())` function, we are only concerned with the r-squared and p-value metrics at the bottom of the output.

From our linear regression model, the r-squared value is 0.50, which means that roughly 50% of all quarter mile time predictions will be correct when using this linear model. Compared to the Pearson correlation coefficient between quarter-mile race time and horsepower of -0.71, we can confirm our r-squared value is approximately the square of our r-value.

In addition, the p-value of our linear regression analysis is 5.77×10^{-6}, which is much smaller than our assumed significance level of 0.05%. Therefore, we can state that there is sufficient evidence to reject our null hypothesis, which means that the slope of our linear model is not zero.

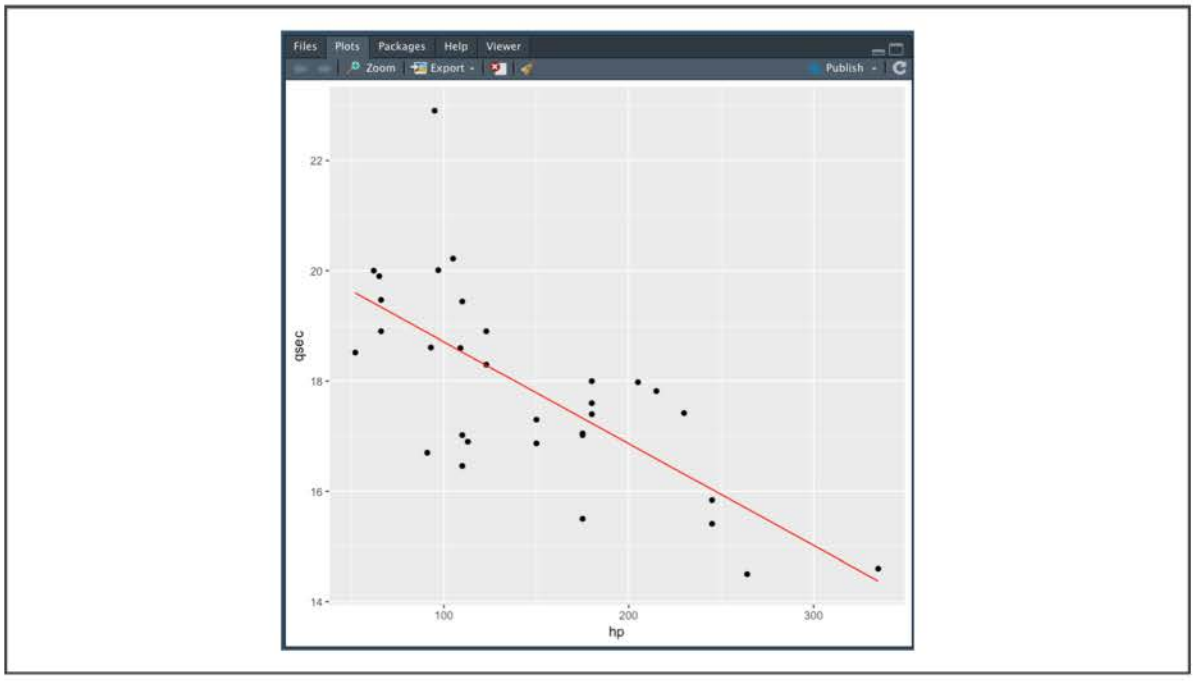
Once we have calculated our linear regression model, we can visualize the fitted line against our dataset using ggplot2.

First, we need to calculate the data points to use for our line plot using our `lm(hp ~ qsec,mtcars)` coefficients as follows:

```
> model <- lm(hp ~ qsec,mtcars) #create linear model
> yvals <- model$coefficients['hp']*mtcars$hp + model$coefficients['(Inte
```

Once we have calculated our line plot data points, we can plot the linear model over our scatter plot:

```
> plt <- ggplot(mtcars,aes(x=hp,y=qsec)) #import dataset into ggplot2
> plt + geom_point() + geom_line(aes(y=yvals), color = "red") #plot scatt
```



Using our visualization in combination with our calculated p-value and r-squared value, we have determined that there is a significant relationship between horsepower and quarter-mile time.

Although the relationship between both variables is statistically significant, this linear model is not ideal. According to the calculated r-squared value, using only quarter-mile time to predict horsepower is roughly as accurate as guessing using a coin toss. In other words, the variability we observed within our horsepower data must come from multiple sources of variance. To accurately predict future horsepower observations, we need to use a more robust model.

15.7.3: Perform Multiple Linear Regression

After reminding himself of how helpful linear regression is, Jeremy decides to keep going—he's really on a roll, and multiple linear regression is ready for him!

Multiple linear regression is a statistical model that extends the scope and flexibility of a simple linear regression model. Instead of using a single independent variable to account for all variability observed in the dependent variable, a multiple linear regression uses multiple independent variables to account for parts of the total variance observed in the dependent variable.

As a result, the linear regression equation is no longer $y = mx + b$. Instead, the multiple linear regression equation becomes $y = m_1x_1 + m_2x_2 + \ldots + m_nx_n + b$, for all independent x variables and their m coefficients.

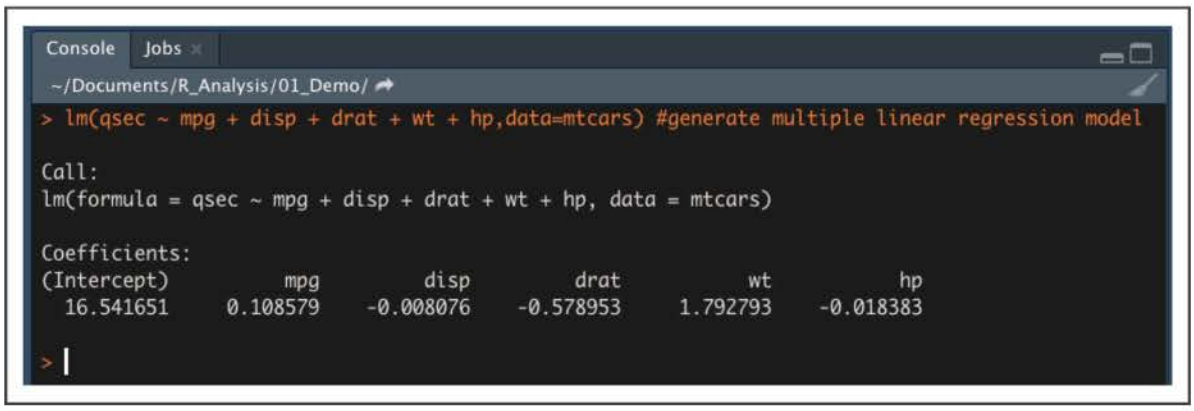
In actuality, a multiple linear regression is a simple linear regression in disguise—all of the assumptions, hypotheses, and outputs are the same. The only difference between multiple linear regression and simple linear regression is how we will evaluate the outputs.

When it comes to multiple linear regression, we'll look at each independent variable to determine if there is a significant relationship with the dependent variable. Once we have evaluated each independent variable, we'll evaluate the r-squared value of the model to determine if the model sufficiently predicts our dependent variable.

To practice multiple linear regression, let's revisit our mtcars dataset. From our last example, we determined that quarter-mile time was not adequately predicted from just horsepower. To better predict the quarter-mile time (`qsec`) dependent variable, we can add other variables of interest such as fuel efficiency (`mpg`), engine size (`disp`), rear axle ratio (`drat`), vehicle weight (`wt`), and horsepower (`hp`) as independent variables to our multiple linear regression model.

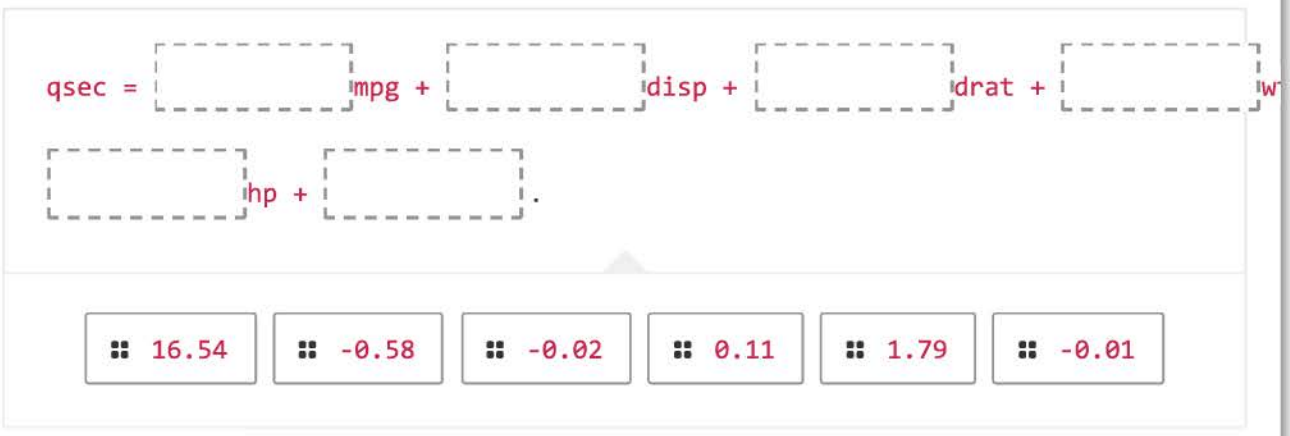
In R, our multiple linear regression statement is as follows:

```
> lm(qsec ~ mpg + disp + drat + wt + hp,data=mtcars) #generate multiple l
```



Similar to the simple linear regression, the output of multiple linear regression using the `lm()` function produces the coefficients for each variable in the linear equation.

Look at the coefficients from the previous multiple linear regression statement. Then fill in the missing coefficient values for the multiple linear regression model. (Round to the nearest hundredth.)



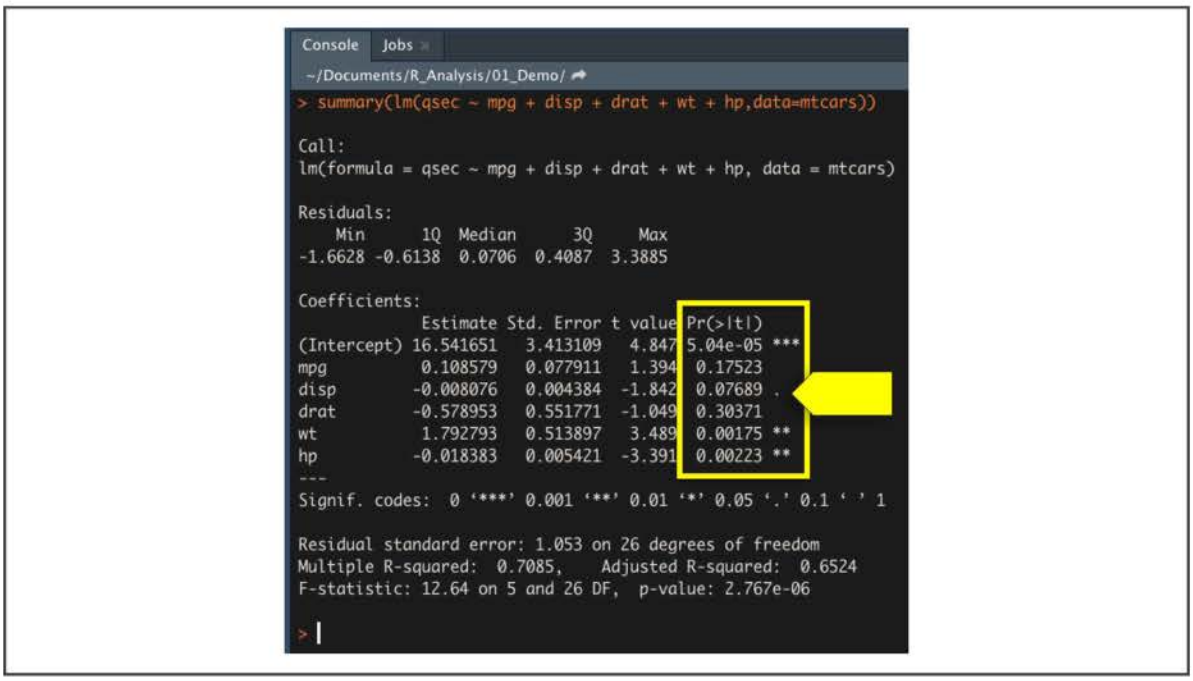
Check Answer

Finish ▸

> **Note**
> Because multiple linear regression models use multiple variables and dimensions, they are almost impossible to plot and visualize.

Now that we have our multiple linear regression model, we need to obtain our statistical metrics using the `summary()` function. In your R console, use the following statement:

```
>summary(lm(qsec ~ mpg + disp + drat + wt + hp,data=mtcars)) #generate su
```



In addition to overall model fit and the statistical test for slope, most data scientists would be curious about the contribution of each variable to the multiple linear regression model. To determine which variables provide a significant contribution to the linear model, we must look at the individual variable p-values.

In the summary output, each `Pr(>|t|)` value represents the probability that each coefficient contributes a random amount of variance to the linear model. According to our results, vehicle weight and horsepower (as well as intercept) are statistically unlikely to provide random amounts of variance to the linear model. In other words the vehicle weight and horsepower have a significant impact on quarter-mile race time. When an intercept is statistically significant, it means there are other variables and factors that contribute to the variation in quarter-mile time that have not been included in our model. These variables may or may not be within our dataset and may still need to be collected or observed.

Despite the number of significant variables, the multiple linear regression model outperformed the simple linear regression. According to the summary output, the r-squared value has increased from 0.50 in the simple linear regression model to 0.71 in our multiple linear regression model while the p-value remained significant.

Caution

> Although the multiple linear regression model is far better at predicting our current dataset, the lack of significant variables is evidence of overfitting. Overfitting means that the performance of a model performs well with a current dataset, but fails to generalize and predict future data correctly. Later in this course we'll learn more about overfitting and ways to avoid it.

Depending on the dataset, the questions being asked, and the audience, a simple linear regression model may be more appropriate than a multiple linear regression model. However, the amount of information that can be obtained and analyzed will be far greater using a multiple linear regression.

As with any data model, it takes practice to learn how to identify variables of interest, select an appropriate model, and refine a model to increase performance. Before moving to the next section, take some time to perform correlation analysis on our previous datasets. Then use the correlation analysis to identify potential variables of interest. Once you have variables of interest, practice generating simple and multiple linear regression models to try and create accurate predictive models.

15.8.1: Category Complexities

Now that Jeremy has learned about statistical concepts such as t-test and linear regression, he feels comfortable analyzing numerical datasets. However, Jeremy recognizes that not all data will be numerical, and eventually he will need to analyze data that is completely categorical. Once again, Jeremy must go back and learn another statistical test that will enable him to compare and contrast the frequency of categorical data.

As we learned previously, categorical data is generally any data that is not measured, or qualitative data. Even though categorical data may not require an instrument to measure, it can be just as informative as numerical data.

One common form of categorical data is **frequency data**, where we record how often something was observed within a single variable. For example, in the mpg dataset, if we were to count up the number of vehicles for each vehicle class, the output would be a form of frequency data.

In data science, we'll often compare frequency data across another dichotomous factor such as gender, A/B groups, member/non-member, and so on. In these cases, we may ask ourselves, "Is there a difference in frequency between our first and second groups?" To test this question, we can perform a chi-squared test.

The **chi-squared test** is used to compare the distribution of frequencies across two groups and tests the following hypotheses:

H_0 : There **is no difference** in frequency distribution between both groups.

H_a : There **is a difference** in frequency distribution between both groups

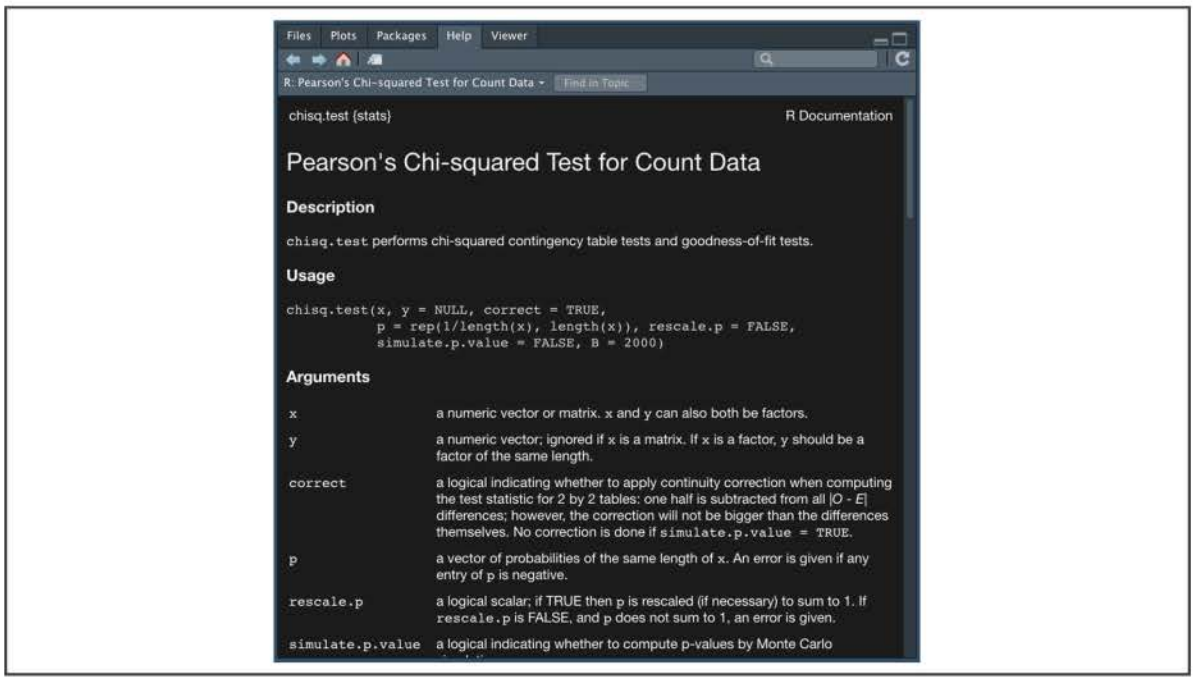
Before we can perform our chi-squared analysis, we must ensure that our dataset meets the assumptions of the statistical test:

1. Each subject within a group contributes to only one frequency. In other words, the sum of all frequencies equals the total number of subjects in a dataset.
2. Each unique value has an equal probability of being observed.
3. There is a minimum of five observed instances for every unique value for a 2x2 chi-squared table.
4. For a larger chi-squared table, there is at least one observation for every unique value and at least 80% of all unique values have five or more observations.

Once we have confirmed our categorical dataset meets all of the assumptions of the chi-square analysis, we can perform our chi-squared test.

In R, we'll compute our chi-squared test using the `chisq.test()` function. Type the following code into the R console to look at the `chisq.test()` documentation in the Help pane.

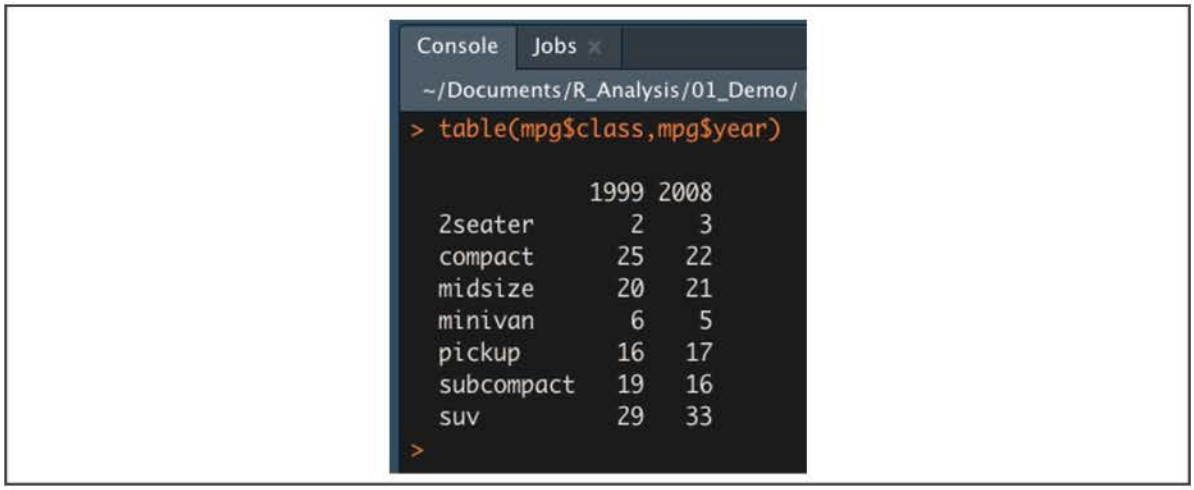
```
>?chisq.test()
```



Depending on the structure of your dataset, you can implement the `chisq.test()` function in multiple ways using the optional arguments. The most straightforward implementation of `chisq.test()` function is passing the function to a contingency table. A **contingency table** is another name for a frequency table produced using R's `table()` function. R's `table()` function does all the heavy lifting for us by calculating frequencies across factors.

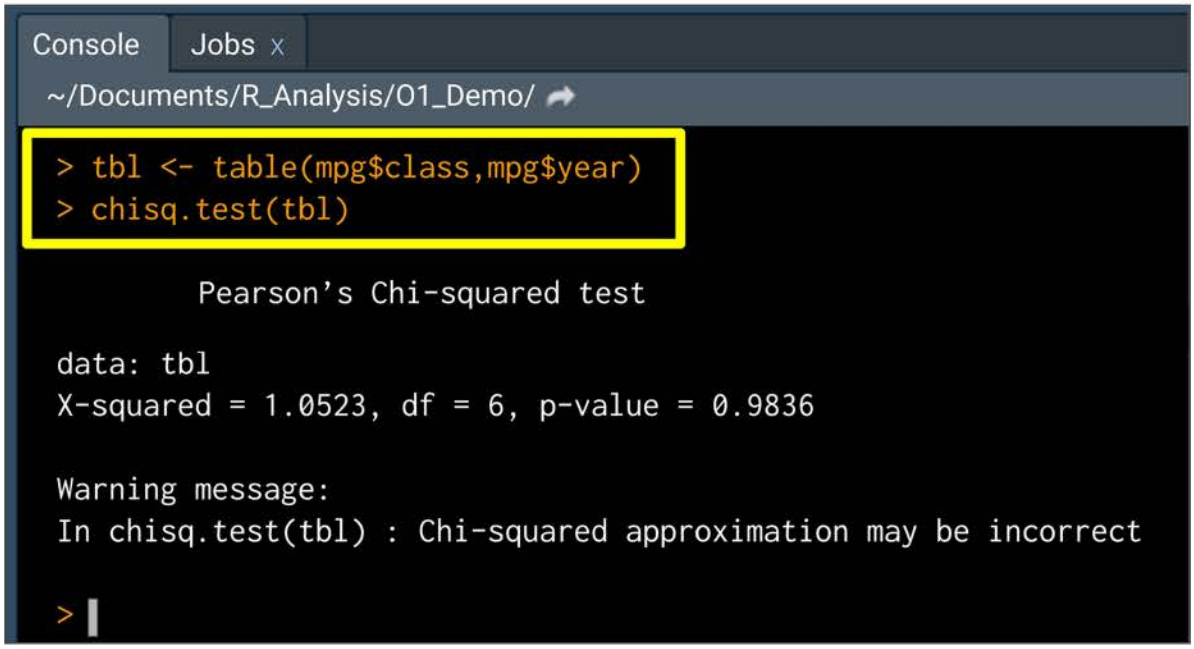
For example, if we want to test whether there is a statistical difference in the distributions of vehicle class across 1999 and 2008 from our mpg dataset, we would first need to build our contingency table as follows:

```
> table(mpg$class,mpg$year) #generate contingency table
```



Then, pass the contingency table to the `chisq.test()` function:

```
> tbl <- table(mpg$class,mpg$year) #generate contingency table
> chisq.test(tbl) #compare categorical distributions
```



Assuming a significance level of 5%, is there sufficient evidence to reject the null hypothesis of the paired t-test?

- Reject the null hypothesis
- Fail to reject the null hypothesis

Check Answer

Finish ▸

IMPORTANT

The chi-squared warning message is due to the small sample size. Because the p-value is so large, we are not too concerned that our interpretation may be incorrect.

Despite having no quantitative input, the chi-squared test enables data scientists to quantify the distribution of categorical variables. Although this test can be applied to more groups and larger datasets, it does have a limit. As the number of groups increases, the likelihood that any change in frequency will be considered significantly different. Therefore, it's important to keep the number of unique values and groups relatively low. A good rule of thumb is to keep the number of unique values and groups lower than 20, which means the degrees of freedom (`df` in the output) is less than or equal to 19.

Take some additional time to practice implementing contingency tables and chi-squared tests using categorical data from our previous datasets. Feel free to tweak the frequency values in the contingency tables tp see what happens to the chi-squared and p-value metrics.

15.9.1: Practice A/B Testing

Jeremy has finally rounded out his basic statistical tests and R practice and is ready to start applying these concepts to his job on the data team. As he begins to take a look at some of the analyses and reports from his colleagues, he notices that these reports keep referring to this concept of A/B Testing. Curious, Jeremy begins to search for more information about what A/B testing is and how he can use it.

Often when performing analysis and testing on a well-established product, website, or software, making changes can be difficult. Well-established products typically have a large consumer base and reliable sales and usage metrics, and are highly valued by their company. As a result, it's too risky to implement changes directly to the product without proper evaluation of the consequences.

To properly evaluate potential product changes, companies can use a technique called A/B testing. **A/B testing** is a randomized controlled experiment that uses a control (unchanged) and experimental (changed) group to test potential changes using a success metric. A/B testing is used to test whether or not the distribution of the success metric increases in the experiment group instead of the control group; we would not want to make changes to the product that would cause a decrease in the success metric.

Although A/B testing has been around for almost a century, giant software and tech companies such as Google and Amazon have popularized the practice by providing in-depth analytic metrics for their Google AdSense and AWS platforms.

Regardless of the industry or product, the process of A/B testing is the same. First, we must decide what changes will be made between the control and experiment groups. Typically, the number of changes will be very limited to ensure comparisons are equal; however, more substantial changes can also be tested using an A/B framework.

Once a consensus has been made on the changes between the two groups, a success metric should be determined. The success metric can vary widely, depending on what is being tested. For example, a website might use consumer engagement as a success metric (e.g., number of visitors, clicked links, or time spent on a page). Alternatively, an automotive design team might want to know how performance changes after a slight design change to a vehicle's form factor, so the team's success metric might be mpg fuel efficiency.

Once we have decided on our experimental changes and the success metric, we must determine which statistical test is most appropriate. In this course, we'll only concern ourselves with normally distributed data and categorical data, which limits the number of statistical tests we'll need. However, if the A/B test groups are disproportionately uneven, or if the success metric distribution is non-normal, more elaborate statistical analysis may be required.

For our purposes, we can apply the following logic to determine the most appropriate statistical test:

- If the success metric is **numerical** and the **sample size is small**, a **z-score summary statistic** can be sufficient to compare the mean and variability of both groups.
- If the success metric is **numerical** and the **sample size is large**, a **two-sample t-test** should be used to compare the distribution of both groups.
- If the success metric is **categorical**, you may use a **chi-squared test** to compare the distribution of categorical values between both groups.

After determining the testing conditions and statistical test, the next biggest consideration in A/B testing is sample size. It's important to collect a sufficient number of data points for each group to ensure that the A/B test results are meaningful.

There are multiple ways to determine optimal sample size, such as quantitative power analyses, but often a qualitative estimate is sufficient. If the changes between the control and experiment groups are expected to have a strong effect on the success metric (often referred to in data science as an **effect size**), fewer data points are necessary for the test. In contrast, if the effect size is small, a larger sample size is required to ensure that statistical findings would be meaningful.

For example, if we were testing purchase rates on an experiment group that receives a pop-up notification when visiting the AutosRUs website, we may use historical purchase rates as an indicator of effect size. If in general people who visit the site are likely to purchase a vehicle, our A/B test sample size can be small. However, if most people who visit the site are not likely to purchase a vehicle, we would need a large number of data points to confirm if the pop-up notifications make a statistical difference.

> **Note**
> Using a quantitative power analysis can be helpful to determine sample size when effect size is unknown or resources are limited. Although performing a power analysis is outside the scope of this course, there is robust documentation online regarding **implementation (http://www.statsoft.com/Textbook/Power-Analysis)** and **interpretation (https://www.statisticssolutions.com/statistical-power-analysis/)** .

Due to its simple design and flexible application, the A/B testing framework is quickly becoming a go-to standard in the data science industry and one of the most highly desired data skills for Fortune 500 companies. Regardless, if you have experience in product design or optimization, you can use A/B testing to make informed design changes and confident development decisions.

15.10.1: Whose Analysis Is It Anyway?

Jeremy has put in a huge amount of work, covered a huge amount of topics, and has really stepped into his leadership position.

Of course, he was hired for the leadership position, even without a strong R background, because of his experience at AutosRUs. That's because there is something even more important than knowing how the math works or knowing how to code in R when it comes to driving value for your company: *knowing the right questions to ask.*

When using data to make informed decisions in a professional environment, implementing a statistics function is not the biggest challenge. Rather, it's determining what questions to ask.

In data science, researchers use **retrospective analysis** to analyze and interpret a previously generated dataset where the outcome is already known. Retrospective analyses are helpful because there are no upfront costs to generate data and statistical results can be compared to the known outcomes. Depending on the dataset and input variables, there is a (potentially) limitless number of statistical questions that can be asked from the data:

- Are two groups statistically different? Use a t-test with one dichotomous independent variable and one continuous dependent variable.
- Can one continuous dependent variable be predicted using another independent variable? What about multiple independent variables and one

dependent variable? Use regression analysis.

- Are there multiple categorical variables tightly linked in a dataset? Are the distributions of the different categorical variables equal? We can test with chi-squared.

In contrast, researchers can **design their own study** to answer their own specific questions. In this case, the data types and size of the dataset will be directly reflective of how complicated their hypotheses are, and what statistical analyses are required to answer the question.

For example, if we want to verify that a car battery ages at an appropriate rate, we would need to test our question with a regression model. If we were to use a multiple linear regression model, we would need to collect numerical variables, such as number of uses, time, battery capacity, tire tread, and engine horsepower. Once we select the variables to collect, we would estimate sample size based on how low of a significance level is necessary and how sensitive the measurements are.

Regardless, if we collect and measure the data ourselves, or if the data has been curated from a previous dataset, statistical tests can help us provide quantitative interpretation to the results.

Module 15 Challenge

Submit Assignment

Due Sunday by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> A few weeks after starting his new role, Jeremy is approached by upper management to help with a special project. AutosRUs' newest prototype, the MechaCar, is suffering from production troubles. There are a number of issues surrounding the vehicle's specifications and manufacturing process that are blocking the manufacturing team from proceeding. AutosRUs' upper management has called on Jeremy and the data analytics team to help analyze the production data in order to justify some last-minute design decisions. With Jeremy's help, the launch of the MechaCar should be one of the most successful product launches in the company's history.

In this challenge, you'll perform a series of statistical tests and create a technical report that provides your interpretation of the findings.

Background

From the upper management team, Jeremy received two datasets:

- The results of an mpg testing dataset of 50 potential prototype MechaCars. The MechaCar prototypes were produced using multiple design specifications to identify ideal vehicle performance. Multiple metrics such as vehicle length, vehicle weight, spoiler angle, drivetrain, and ground clearance were collected for each vehicle.
- MechaCar suspension coil test results from multiple production lots. In this dataset, the weight capacity of multiple suspension coils were tested to determine if the manufacturing process is consistent across lots.

By combining his understanding of R and statistics with the manufacturing datasets provided by the upper management, Jeremy should have all the materials he needs to generate a robust technical report. Technical reports such as the one Jeremy will design are common in product development and are used to justify design choices using quantitative and qualitative reasoning.

Objectives

The goals of this challenge are for you to complete the following:

- Design and interpret a multiple linear regression analysis to identify variables of interest.
- Calculate summary statistics for quantitative variables.
- Perform a t-test in R and provide interpretation of results.
- Design your own statistical study to compare vehicle performance of two vehicles.

Instructions

MPG Regression

- Create a new RScript in your R source pane and save it to your active directory. Name this new RScript file `MechaCarChallenge.RScript`. (**Hint:** Create a new RScript by going to the File menu. Select "New File" followed by "RScript." Or you can click the icon in the top-left corner of the RStudio window. Note that the icon looks like a white square with a plus sign in the top left corner.)
- Download the **MechaCar mpg dataset** and place it in your active directory for your R session.
- Using multiple linear regression, design a linear model that predicts the mpg of MechaCar prototypes using a number of variables within the MechaCar mpg dataset. Create a separate text file called `MechaCarWriteUp.txt`. In the text file,

provide a small writeup of your interpretation of the multiple linear regression results. Be sure to include the following details:

  - Which variables/coefficients provided a non-random amount of variance to the mpg values in the dataset?
  - Is the slope of the linear model considered to be zero? Why or why not?
  - Does this linear model predict mpg of MechaCar prototypes effectively? Why or why not?

Suspension Coil Summary

- Download the **suspension coil test** result dataset and place it in your active directory for your R session.
- In the same `MechaCarChallenge.RScript` file, create a summary statistics table for the suspension coil's pounds-per-inch continuous variable.
  - Be sure to include the following metrics:
    - Mean
    - Median
    - Variance
    - Standard deviation
  - Using the same `MechaCarWriteUp.txt` text file, provide a short write-up of your interpretation and findings for the suspension coil summary statistics. Be sure to include the following details:
    - The design specifications for the MechaCar suspension coils dictate that the variance of the suspension coils must not exceed 100 pounds per inch. Does the current manufacturing data meet this design specification? Why or why not?

Suspension Coil T-Test

- Using the same suspension coil data and the `MechaCarChallenge.RScript` file, determine if the suspension coil's pound-per-inch results are statistically different from the mean population results of 1,500 pounds per inch. (**Hint:** Refer to the t-test section of this module to determine which statistical test to use.)
- In the `MechaCarWriteUp.txt` text file, provide a small writeup of your interpretation and findings for the t-test results.

Design Your Own Study

Upper management is looking for your expertise and wants you to design a study that compares the performance of the MechaCar prototype vehicle to other comparable vehicles on the market. In the `MechaCarWriteUp.txt` text file, write a short description of a statistical study that can quantify how the MechaCar outperforms the competition. In your study design, be sure to write about the following considerations:

- Think critically about what metrics you would think would be of interest to a consumer (cost, fuel efficiency, color options, etc.).
- Determine what question we would ask, what the null and alternative hypothesis would be to answer that question, and what statistical test could be used to test this hypothesis.
- Knowing what test should be used, what data should be collected? **Hint:** Look at the **cheat sheet** for required variables.

Submission

Make sure your repo is up to date and includes the following:

- A `MechaCarChallenge.RScript` file containing all of your R code for the technical report
- A `MechaCarWriteUp.txt` text file, containing your interpretation and findings for each statistical analysis.

Submit a link to your repository through Canvas.

Rubric

Please **download the detailed rubric** to access the assessment criteria.

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Module 15 Rubric

Criteria	Ratings					Pts
Multiple Linear Regression Written Analysis Please see detailed rubric linked in Challenge description.	**15.0 pts Mastery**	**12.0 pts Approaching Mastery**	**9.0 pts Progressing**	**6.0 pts Emerging**	**0.0 pts Incomplete**	15.0 pts
Multiple Linear Regression Model Please see detailed rubric linked in Challenge description.	**15.0 pts Mastery**	**12.0 pts Approaching Mastery**	**9.0 pts Progressing**	**6.0 pts Emerging**	**0.0 pts Incomplete**	15.0 pts
Summary Statistics Written Analysis Please see detailed rubric linked in Challenge description.	**10.0 pts Mastery**	**8.0 pts Approaching Mastery**	**6.0 pts Progressing**	**4.0 pts Emerging**	**0.0 pts Incomplete**	10.0 pts
Summary Statistics Model Please see detailed rubric linked in Challenge description.	**10.0 pts Mastery**	**8.0 pts Approaching Mastery**	**6.0 pts Progressing**	**4.0 pts Emerging**	**0.0 pts Incomplete**	10.0 pts
Statistical Difference Written Analysis Please see detailed rubric linked in Challenge description.	**10.0 pts Mastery**	**8.0 pts Approaching Mastery**	**6.0 pts Progressing**	**4.0 pts Emerging**	**0.0 pts Incomplete**	10.0 pts
Statistical Difference Model Please see detailed rubric linked in Challenge description.	**10.0 pts Mastery**	**8.0 pts Approaching Mastery**	**6.0 pts Progressing**	**4.0 pts Emerging**	**0.0 pts Incomplete**	10.0 pts
Statistical Study Design Please see detailed rubric linked in Challenge description.	**30.0 pts Mastery**	**24.0 pts Approaching Mastery**	**18.0 pts Progressing**	**12.0 pts Emerging**	**0.0 pts Incomplete**	30.0 pts
						Total Points: 100.0