



1

PREPARING DATA FOR ANALYSIS AND VISUALIZATION IN R

The R-Team and
the Pot Policy Problem

1.1 Choosing and learning R

Leslie walked past her adviser's office and stopped. She backed up to read a flyer hanging on the wall. The flyer announced a new local chapter of *R-Ladies* (see Box 1.1). Yes! she thought. She'd been wanting to learn R the entire year.

1.1 R-Ladies

R-Ladies is a global group with chapters in cities around the world. The mission of R-Ladies is to increase gender diversity in the R community. To learn more, visit the R-Ladies Global website at <https://rladies.org/> and the R-Ladies Global Twitter feed, @RLadiesGlobal.

Leslie arrived at the R-Ladies event early. "Hi, I'm Leslie," she told the first woman she met.

"Hey! Great to meet you. I'm Nancy," answered the woman as they shook hands.

"And I'm Kiara, one of Nancy's friends," said another woman, half-hugging Nancy as she reached out to shake Leslie's hand. "Can we guess you're here to learn more about R?"

Leslie nodded.

"You've come to the right place," said Nancy. "But let's introduce ourselves first. I'm an experienced data scientist working for a biotech startup, and I love to code."

"You might call me a data management guru," Kiara said. "I just gave notice at my job with a large online retailer because I'm starting a job with the Federal Reserve next month."

Leslie asked Kiara and Nancy about their experience with R. "What do you like about R compared to other traditional statistics software options I've been learning in my degree program?"


Nancy thought for a minute and answered, "Three main reasons: cost, contributors, and community."

First, Nancy explained, "The cost of R can't be beat. R is free, while licenses for other statistics software can cost hundreds of dollars for individuals and many thousands for businesses. While large, successful businesses and universities can often afford these licenses, the cost can be an insurmountable burden for small businesses, nonprofits, students, teachers, and researchers."

Kiara added, "The cost of the tools used in data science can be a social justice issue [Krishnaswamy & Marinova, 2012; Sullivan, 2011]. With R, students, researchers, and professionals in settings with limited resources have just as much access as an executive in a fancy high-rise in the middle of downtown San Francisco."

The second thing that Nancy loved about R was the contributors. She explained, "R is not only free but it is also *open source*. Anyone can contribute to it!"

Leslie looked confused. Nancy explained that anyone can write a *package* in the R language and contribute the package to a repository that is accessible online. Packages are small pieces of software that are often developed to do one specific thing or a set of related things.

 Visit edge.sagepub.com/harris1e to watch an R tutorial

Kiara offered, “A package I use a lot is called **tidyverse**. The **tidyverse** package includes functions that are useful for common data management tasks like recoding variables. By using code that someone else has written and made available, I don’t have to write long programs from scratch to do the typical things I do on a regular basis.”

Leslie asked, “OK, but how is it possible for everyone to have access to software written by anyone in the world?”

Kiara explained that people write packages like **tidyverse** and submit them to the Comprehensive R Archive Network, also known as CRAN. After volunteers and resources from the nonprofit R Foundation (<https://www.r-project.org/foundation/>) review them, they decide whether to reject or accept new packages. New packages are added to the CRAN (CRAN Repository Policy, n.d.) for everyone to access.

Leslie nodded. Then Kiara pulled up the **contributed packages** website and showed Leslie the more than 14,000 packages available on the **CRAN** (<https://cran.r-project.org/submit.html>). Leslie was still a little confused about the idea of a package.

Kiara said, “Think of a package as a computer program that you open when you want to do a specific thing. For example, if you wanted to create a slide show, you might open the Microsoft PowerPoint program. But if you wanted to do a data management or analysis task, you would use packages in R. Unlike PowerPoint, however, anyone in the world can write a package and contribute it to the CRAN for anyone in the world to use.”

Nancy had saved the best for last: the R community. She explained, “The R community is inclusive and active online, and R community groups like R-Ladies Global [Daish et al., 2019] specifically support voices that are underrepresented in the R community and in data science. Plus, R users love to share their new projects and help one another.”

Kiara agreed enthusiastically. “I look at (and post to) the #rstats hashtag often on Twitter and keep learning great new features of R and R packages.”

Kiara shared two more benefits of using R. The first was great graphics. She explained that R is extraordinary for its ability to create high-quality visualizations of data. “The code is extremely flexible, allowing users to customize graphics in nearly any way they can imagine,” she said. The second benefit was that R is a great tool for conducting analyses that are **reproducible** by someone else. She noted that the R community is actively building and supporting new packages and other tools that support reproducible workflows.

Nancy mentioned that reproducibility has become an important part of science as the scientific community addressed the problem of poor and unethical scientific practices exposed in published research (Steen et al., 2013) (see Box 1.2).

“This all sounds great,” said Leslie. “What’s the catch?”

Nancy and Kiara looked at each other for a minute and smiled.

“OK,” Kiara said. “I admit there are a few things that are challenging about R, but they are related to the reasons that R is great.” The first challenge, she explained, is that the contributors to R can be anyone from anywhere. With such a broad range of people creating packages for R, the packages end up following different formats or rules. This means that learning R can be tricky sometimes when a function or package does not work the way other, similar packages do.

Nancy agreed and said, “Also, since R is open source, there is no company behind the product that will provide technical support. But there is a very active community of R users, which means that solutions to problems can often be solved relatively quickly with a tweet, email, or question posted to a message board.”



1.2 Kiara's reproducibility resource: Reproducible research

The scientific standard for building evidence is replication, which is repeating scientific studies from the beginning and comparing results to see if you get the same thing. While replication is ideal, it can be very time-consuming and expensive. One alternative to replication is reproducibility. Reproducing a study is reanalyzing existing data to determine if you get the same results. Reproducibility requires, at a minimum, accessible data and clear instructions for data management and analysis (Harris et al., 2019).

Science is currently facing a reproducibility crisis. Recent research has found that

- 20% to 80% of papers published in a sample of journals included an unclear or unknown sample size (Gosselin, n.d.),
- up to 40% of papers per journal in a sample of journals included unclear or unknown statistical tests (Gosselin, n.d.),
- approximately 6% of p -values were reported incorrectly in a sample of psychology papers (Nuijten, Hartgerink, Assen, Epskamp, & Wicherts, 2015),
- 11% of p -values were incorrect in a sample of medical papers (García-Berthou & Alcaraz, 2004),
- just 21% of 67 drug studies and 40% to 60% of 100 psychological studies were successfully replicated (Anderson et al., 2016; Open Science Collaboration, 2015; Prinz, Schlange, & Asadullah, 2011), and
- 61% of economics papers were replicated (Camerer et al., 2016).

As you make your way through this text, you will find tips on how to format the code you write to manage and analyze your data. Writing, formatting, and annotating your code clearly can increase reproducibility.

Leslie was intrigued. She wanted to learn more about R. Kiara and Nancy had enjoyed the conversation, too.

“Why don’t we form our own small group to teach Leslie about R and learn collaboratively?” Kiara said.

“Let’s do it!” said Nancy. She had been watching a lot of 1980s TV lately, so she suggested that they call themselves the “R-Team.” “We’ll be like the 80s show, the *A-Team*.” Then she sang some of the theme song, “If you have a problem, if no one else can help . . .”

“I think we get the idea,” said Kiara, laughing.

“I don’t know that show,” Leslie said, smiling. “But I’m all for joining the R-Team!”

1.2 Learning R with publicly available data

Before the evening ended, Kiara recommended that they use publicly available data to learn strategies and tools that would be useful in the real world. Who wants to use fake data? Real data may be messier, but they are more fun and applicable. Nancy agreed. Since she was especially interested in data about current issues in the news, she suggested that they begin by working with data on marijuana policy. She pointed out that several states had legalized medicinal and recreational marijuana in the past few years and that she'd recently come across legalization questions in a large national publicly available data set called the *General Social Survey* (National Opinion Research Center, 2019) that is available online from the National Opinion Research Center (NORC) at the University of Chicago.

Nancy suggested that the first day of R should be focused on getting used to R and preparing data for analysis. The R-Team agreed that they would work on preparing data for analysis and would use marijuana legalization data to practice the skills. Kiara put together a list of things to achieve in their first meeting.

1.3 Achievements to unlock

- Achievement 1: Observations and variables
- Achievement 2: Using reproducible research practices
- Achievement 3: Understanding and changing data types
- Achievement 4: Entering or loading data into R
- Achievement 5: Identifying and treating missing values
- Achievement 6: Building a basic bar chart

1.4 The tricky weed problem

1.4.1 MARIJUANA LEGALIZATION

When Leslie showed up for the first meeting of the R-Team, Kiara offered her coffee and bagels. Leslie was enjoying learning R already! To start the meeting, Nancy shared some marijuana legalization research. She told them she had learned that California had become the first state to legalize medical marijuana in 1996 ("Timeline of Cannabis Laws," n.d.). She had also learned that marijuana use remained illegal under federal law, but that 29 states and the District of Columbia had legalized marijuana at the state level for medical or recreational use or both by 2017. With new ballot measures at the state level being introduced and passed by voters on a regular basis, as of 2017 there appeared to be momentum for a nationwide shift toward legalization.

Nancy refilled her coffee and continued. She said that in 2017, Jeff Sessions was appointed as attorney general of the United States. Sessions did not support legalization and regularly expressed interest in prosecuting medical marijuana providers. Despite this difficult climate, in 2018, Michigan voters approved a ballot measure to legalize recreational cannabis, and voters in Missouri, Oklahoma, and Utah passed ballot measures legalizing medical marijuana. In 2018, Vermont became the first to legalize recreational marijuana via the state legislature (Wilson, 2018).

In 2019, Sessions was replaced as attorney general by William Barr, who testified that he would not pursue marijuana companies that complied with state laws and stated that he was supportive of expanding marijuana manufacturing for scientific research (Angell, 2019). However, Barr did not indicate his support, or lack of support, for additional legalization. With the existing federal policy and the unstable legal environment, it is unclear what will happen next for marijuana policy.

Nancy explained that, with the exception of Vermont, policy changes had primarily happened through successful ballot initiatives. She suggested that learning more about support among voters for legalization could help understand what is likely to happen next. Leslie and Kiara agreed.

Kiara looked up the General Social Survey, or GSS. She explained to Leslie that the GSS is a large survey of U.S. residents conducted each year since 1972, with all of the data available for public use. The GSS Data Explorer (<https://gssdataexplorer.norc.org>) allows people to create a free account and browse the data that have been collected in the surveys, which have changed over time. In several years, including 2018, the GSS survey included a question asking the survey participants whether they support marijuana legalization. Kiara used the Data Explorer to select the marijuana legalization question and a question about age.

Leslie asked her why she selected age. Kiara explained that, since marijuana legalization had been primarily up to voters so far, the success of ballot initiatives in the future will depend on the support of people of voting age. If younger people are more supportive, this suggests that over time, the electorate will become more supportive as the old electorate decreases. Leslie found that to be logical, although a little morbid.

Kiara was also interested in how the marijuana legalization and age questions were worded in the GSS and what the response options were. She saw that the GSS question was worded as follows:

Do you think the use of marijuana should be legal or not?

Below the question, the different response options were listed: legal, not legal, don't know, no answer, not applicable.

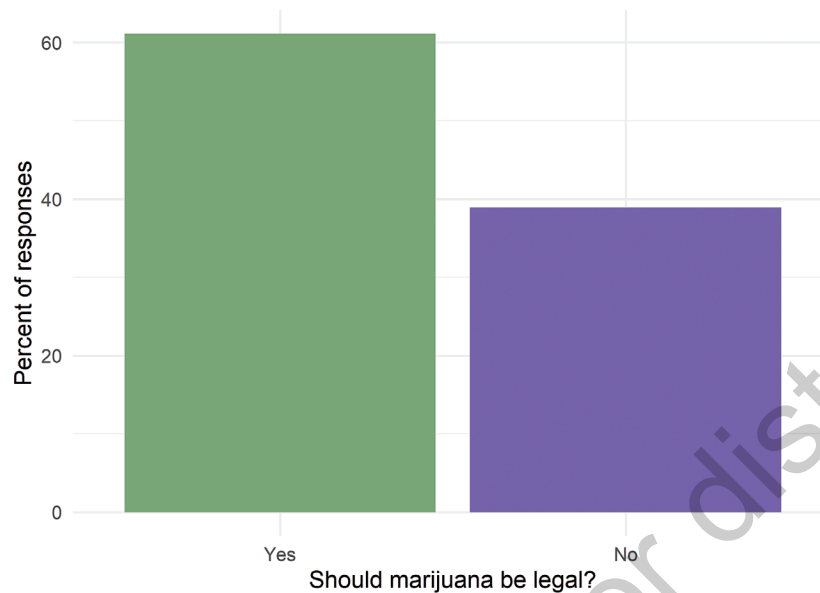
Kiara found the `age` variable and noticed that the actual question was not shown and just listed as "RESPONDENT'S AGE." The variable is recorded as whole numbers ranging from 18 to 88. At the bottom of the web page about the `age` variable, the GSS Data Explorer showed that age was recorded as "89 OR OLDER" for anyone 89 years old or older.

Nancy was eager to put her love of coding to work. She imported the GSS data and created a graph about marijuana legalization to get them started (Figure 1.1).

Leslie examined the graph. She saw that the x -axis across the bottom was labeled with the marijuana question and the two response categories. She noticed that the y -axis was the percentage who responded. The bar with the Yes label went up to just past 60 on the y -axis, indicating that just over 60% of people support legalization. The bar labeled No stopped just under 40, so just under 40% do not think marijuana should be legal.

Although marijuana legalization appeared to have a lot of support from this first graph, Leslie thought it might not be that simple. Policy change depended on *who* supports marijuana legalization. Are the supporters voters? Do they live in states that have not yet passed legalization policy or in the states that have already legalized it? In addition to answering these important questions, one thing that might provide a little more information about the future is to examine support by the age of the supporter. If supporters tend to be older voters, then enthusiasm for legalization may weaken as the population ages. If supporters are younger voters, then enthusiasm for legalization may strengthen as the population ages. Nancy

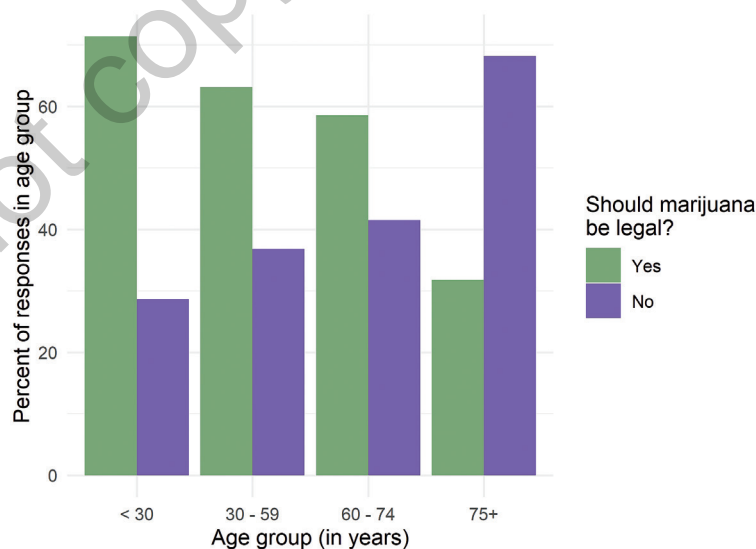
FIGURE 1.1 Support for marijuana legalization among participants in the 2016 General Social Survey



was delighted at a second chance (already) to jump right in and write some additional code so that the graph included age groups as well (Figure 1.2).

The R-Team could see a pretty clear pattern in the graph. The x -axis now showed the age groups, while the y -axis showed the percentage of people. The bar colors represent Yes and No. Leslie saw that the percentage of people supporting legalization looked three times larger than the percentage who did not

FIGURE 1.2 Support for marijuana legalization by age group among participants in the 2016 General Social Survey





1.3 Kiara's reproducibility resource: Installing R and RStudio

To follow along with this text, install R and RStudio. R can be downloaded from The Comprehensive R Archive Network (<https://cran.r-project.org>). Once R has been installed, then install RStudio (<https://www.rstudio.com>). RStudio is an interactive development environment, which in this case makes

R much easier to use.

Open RStudio (not R) and make sure that everything has installed correctly. There should be a window open on the left-hand side of the RStudio screen that says "Console" in small bold print in the top left corner. RStudio automatically finds R and runs it for you within this console window.

Check to see if R is working properly by typing in the following code (shown in shading) at the R > prompt in the console on the left. Press Enter after typing each line of code to get the results shown on the lines that begin with ##:

```
2+2
## [1] 4
(4+6)/2
## [1] 5
10^2
## [1] 100
a <- 3
a
## [1] 3
```

support in the youngest age group. In the oldest age group, those who do not support legalization are a much larger group. Leslie noted that, as those in the younger categories age, it appeared that support would continue to rise and marijuana legalization ballot initiatives would continue to be supported in future elections.

Kiara explained to Leslie that visual representations of data like Figures 1.1 and 1.2 could be powerful tools for understanding and communicating with data. Although the two graphs may look simple, there is a lot going on behind the scenes. For the remainder of the R-Team meeting, they would focus on how to prepare data for analysis and visualization in R. Preparing data for analysis is *data management*, so Kiara explained she would be the primary guide (with help from Nancy, who loves to code).

Kiara told Leslie that she would use R and RStudio for examples and that she highly recommended Leslie follow along with R and RStudio on her own computer. To make sure that Leslie could follow along, Kiara wrote instructions for installing R and RStudio on a computer (see Box 1.3).

1.5 Achievement 1: Observations and variables

1.5.1 DEFINING OBSERVATIONS AND VARIABLES

Before Kiara began data management tasks, she thought a few vocabulary terms would be useful to discuss. Kiara explained that data scientists were usually interested in the characteristics and behaviors of humans and organizations. To understand these things, scientists often measured and recorded information about people or organizations. For example, a data scientist working on political science might be interested in understanding whether income is related to voting. To get this information, she could ask a group of people whether they voted in the most recent election and what they earned in income in the most recent year. In this case, each person is an *observation*, and there are two variables, `income` and `voting` behavior.

Leslie thought she understood, so she summarized that people being measured are observations and the various pieces of information about each person are *variables*. Kiara nodded in agreement and emphasized that observations and variables were key concepts in data science, so it is worth taking a few more minutes to think about. Kiara thought a visual representation of observations and variables might be useful. She explained that, in a typical data set, observations are the rows and variables are the columns. For the example of `voting` and `income`, a data set might look like this:

```
income voted
1 34000 yes
2 123000 no
3 21500 no
```

In this very small data set, there are three observations (the rows) and two variables (the columns). The first observation is a person with an income of \$34,000 who answered “yes” for voted. The second observation is a person with an income of \$123,000 who answered “no” for voted. The third observation is a person with an income of \$21,500 who responded “no” for voted. The two variables are `income` and `voted`.

1.5.2 ENTERING AND STORING VARIABLES IN R

Now that she had introduced the basic idea for observations and variables, Kiara transitioned into talking about R. She explained that R stores information as *objects*, and then data analysis and data management are performed on these stored objects. Before an object can be used in data management or analysis in R, it has to be stored in the R environment.

Information is stored as objects in R by *assigning* the information a name, which can be a single letter or some combination of letters and numbers that will serve as the name of the object. Assigning an object to a name is done by using an arrow like this: `<-`. The arrow separates the name of the object on the left from the object itself on the right, like this: `name <- object`. An object can be as simple as one letter or number, or as complex as several data sets combined.

Since this was their first task in R, Kiara had Leslie try storing the value of 29—for the number of states with legal medical marijuana—in an object called `states` by typing the following at the R prompt (`>`) in the Console pane of the RStudio window. Leslie had read about the panes in an email from Kiara earlier (see Box 1.4) and typed the following at the `>` prompt:

```
states <- 29
```



1.4 Kiara's reproducibility resource: RStudio is a pane

When opening RStudio for the first time, you will notice it is divided into sections. Each of these sections is a **pane** in the RStudio window. The default panes are the Console (left), Environment & History (top right), and Files & Plots & Connections & Packages & Help & Viewer (bottom right). The panes you will use the most throughout this text are the Source pane, which opens when a new R script file is opened through the File menu, Console, Environment & History, Help, and Plots.

- **Source:** Allows you to write R code
- **Console:** Shows the results of running code
- **Environment:** Shows what objects you currently have open and available in R
- **History:** Keeps a running list of the code you have used so far
- **Help:** Provides information on functions and packages you are using
- **Plots:** Shows graphs you create

If you would like to choose and organize your visible panes, click on the View menu and choose Panes to see all of the available options.

When Leslie tried typing the code and pressing Enter, she noticed that running the `states <- 29` code did not seem to result in anything actually happening in the Console window. Kiara explained that when she pressed Enter, the object was stored and there is no result to display. Specifically, R is storing the number 29 as an object called `states` for Leslie to use in her work. While nothing happened in the Console, something did happen. Kiara explained that the `states` object is now stored under the Environment tab in the top right pane of the RStudio window. Looking at the tab, Leslie noticed that the window shows `states` and its value of 29 under a heading titled Values (Figure 1.3).

Leslie saw that there was a **History** tab next to the **Environment** tab in the upper right pane. She clicked on the History tab and was surprised to see `states <- 29`. Kiara explained that this tab holds all of the code run since the History pane was last cleared. Nancy admitted this was one of her favorite parts of R; you can double-click on any of the code shown in the History pane and R will send the code to the Console, ready to run again. There was no need to type anything twice!

FIGURE 1.3 Environment window in RStudio showing the newly created `states` variable and its value



Leslie had clearly missed some things in Kiara’s email and wanted to review it again to learn more about the panes, but for now she was excited to try double-clicking on the `states <- 29` code in the History pane. As soon as she did, the code was sent to the Console. Leslie was delighted! This seemed like a great feature. To see the value of the `states` object created in the Console pane, Kiara had Leslie type the name of the object, `states`, at the R > prompt and press Enter. In shaded sections throughout this text, the rows starting “##” show the output that will appear after running the R code just above it.

```
states
## [1] 29
```

Now the value of `states` appears!

To demonstrate using the `states` object in a mathematical expression, Kiara told Leslie to type the expression `2 + states` at the R > prompt and press Enter.

```
2 + states
## [1] 31
```

Leslie noted that 31 is printed in the Console. This is the value of $2 + 29$. Before they continued, Kiara wanted to explain a few vocabulary terms to Leslie. When Leslie entered code and then hit Enter, the result displayed on the screen is the *output*. For example, the 31 printed after running the code above is output.

1.5.3 ACHIEVEMENT 1: CHECK YOUR UNDERSTANDING

Assign your age in years to an object with your name. Add 5 to the object and press Enter to see how old you will be in 5 years.

1.6 Achievement 2: Using reproducible research practices

Before getting too far into coding, Kiara wanted Leslie to be thinking about how to choose things like object names so that they are useful not only right now but in the future for anyone (including the original author) who relies on the R code. Kiara had learned this lesson well in her many years of coding.

1.6.1 USING COMMENTS TO ORGANIZE AND EXPLAIN CODE

For example, the meaning of the code above—`states <- 29`, `states`, and `2 + states`—may seem obvious right now while they are new, but in a few weeks, it might be less clear why `states` has a value of 29 and what this object means. One way to keep track of the purpose of code is to write short explanations in the code while coding. For this to work, the code needs to be written in the *Source* pane, which is opened by creating a new *Script file*. To create a new Script file, Kiara told Leslie to go to the File menu in the upper left corner of RStudio and choose “New File” from the choices. Then, from the New File menu, choose “R Script.” Nancy suggested using the shortcut command of Control-Shift-n. Leslie tried the shortcut, and a fourth pane opened in the upper left side of the RStudio window with a new blank file that said “Untitled1” in a tab at the top. This is a Script file in the Source pane that can be used for writing code.

Kiara paused here to give Leslie more information about the difference between writing R code in the Console versus writing R code in a Script file. She explained that the difference is mostly about being able

to edit and save code; code written in the Console at the `>` prompt is executed immediately and cannot be edited or saved (e.g., like an instant message). Leslie suggested that the History pane saves the code. Kiara confirmed that this is true and that the history can even be saved as a file using the save icon in the history tab. However, she explained that code saved from the history tab is not usually well formatted and cannot be edited or formatted before saving, so this code will usually be pretty messy and not too useful.

A Script file, however, is a text file similar to something written in the Notepad text editor on a Windows computer or the TextEdit text editor on a Mac computer. A Script file can be edited, saved, and shared just like any text file. When Script files of R code are saved, they have the `.R` file extension.

Kiara opened a new Script file and showed Leslie how to include information about the code by typing comments that explain what the code is for, like this:

```
# create an object with the number of states with
# legal medical marijuana in 2017
states <- 29

# print the value of the states object
states

# determine how many states there would be if 2
# more passed this policy
2 + states
```

Each line of code is preceded by a short statement of its purpose. These statements are called comments, and the practice of adding comments to code is called **commenting** or **annotation**. The practice of commenting or annotating is one of the most important habits to develop in R or in any programming language.

In R, comments are denoted by a hashtag `#`, which notifies R that the text following the `#` on the same line is a comment and not something to be computed or stored. Comments are not necessary for code to run, but are important for describing and remembering what the code does. Annotation is a *best practice* of coding. When writing and annotating code, keep two goals in mind.

- Write clear code that does not need a lot of comments.
- Include useful comments where needed so that anyone (including yourself in the future) can run and understand your code.

Kiara explained that clear R code with useful annotation will help Leslie's work be *reproducible*, which is one of the most important characteristics of good data science. Kiara had collected some information about reproducible research for Leslie (see Box 1.2).

Before moving on, Leslie tried writing the code and comments above in the Script file she had open in the Source pane (see Box 1.4). She finished writing the code but then was not sure how to run the code to check her work. Kiara explained that the code can be run in several ways. One way is to highlight all the code at once and click on Run at the top right corner of the Source pane. To run one line of code at a time, highlighting the line of code or putting the cursor anywhere in the line of code and clicking Run also works. The keyboard shortcut for Run is Control-Enter (or Command-Enter on a Mac), so putting the cursor on a line of code and pressing Control-Enter will run the code on that line (Figure 1.4).

Leslie highlighted all of the code and clicked on Run at the top of the Source pane. In the Console window, she saw the code and the output from the code shown in Figure 1.5.

FIGURE 1.4 Source pane in RStudio showing R code and comments

```
1 # create an object with the number of states with
2 # legal medical marijuana in 2017
3 states <- 29
4
5 # print the value of the states object
6 states
7
8 # determine how many states there would be if 2
9 # more passed this policy
10 2 + states
```

FIGURE 1.5 Console pane in RStudio showing R code, comments, and results

```
> # create an object with the number of states with
> # legal medical marijuana in 2017
> states <- 29
>
> # print the value of the states object
> states
[1] 29
>
> # determine how many states there would be if 2
> # more passed this policy
> 2 + states
[1] 31
>
```

1.6.2 INCLUDING A PROLOG TO INTRODUCE A SCRIPT FILE

Before moving on to more statistics and R code, Kiara wanted Leslie to add one more thing to her code, a *prolog*. She explained that a prolog is a set of comments at the top of a code file that provides information about what is in the file. Including a prolog is another best practice for coding. A prolog can have many features, including the following:

- Project name
- Project purpose
- Name(s) of data set(s) used in the project
- Location(s) of data set(s) used in the project
- Code author name (you!)
- Date code created
- Date last time code was edited

Kiara gave Leslie two examples, one formal and one informal. The formal prolog might be set apart from the code by a barrier of hashtags, like this:

```
# PROLOG #####
# PROJECT: NAME OF PROJECT HERE #
# PURPOSE: MAJOR POINT(S) OF WHAT I AM DOING WITH THE DATA HERE #
# DIR: list directory(-ies) for files here #
# DATA: list data set file names/availability here, e.g., #
# filename.correctextension #
# somewebaddress.com #
# AUTHOR: AUTHOR NAME(S) #
```

```

# CREATED: MONTH dd, YEAR #
# LATEST: MONTH dd, YEAR #####
# NOTES: indent all additional lines under each heading, #
#         & use the hashmark bookends that appear #
#         KEEP PURPOSE, AUTHOR, CREATED & LATEST ENTRIES IN UPPER CASE, #
#         with appropriate case for DIR & DATA, lower case for notes #
#         If multiple lines become too much, #
#         simplify and write code book and readme. #
#         HINT #1: Decide what a long prolog is. #
#         HINT #2: copy & paste this into new script & replace text. #

# PROLOG #####

```

An informal prolog might just include the following elements:

```

#####
# Project name
# Project purpose
# Code author name
# Date last edited
# Location of data used
#####

```

Kiara had Leslie write a prolog at the top of her code file in the Source pane. Leslie's code in the Source pane now looked like this:

```

#####
# Project: R-Team meeting one
# Purpose: Code examples for meeting one
# Author: Leslie
# Edit date: April 19, 2019
# No external data files used
#####
# create an object with the number of states with
# legal medical marijuana in 2017
states <- 29

# print the value of the states object
states

# determine how many states there would be if 2
# more passed this policy
2 + states

```

Before continuing, Kiara suggested Leslie save her Script using the Save icon at the top left side of the Source pane or through the File menu.

Leslie saved it as `analysis.R` on her desktop and looked up just in time to see Kiara cringe. When saving a file, Kiara explained, include information in the file name that is a reminder of what is contained in the file. For example, a file name with `date_project_author` will make identifying the most recent file created for a project easier. In this case, Leslie might save the file as `171130_chap1_leslie.R` for the date of November 30, 2017. Kiara mentioned that putting the year first and then the month and the day is a good idea to avoid problems with reading the date since not all countries use the same order in common practice.

Leslie resaved her file with a better name.

1.6.3 NAMING OBJECTS

In addition to annotating code, using a prolog, and including useful information in a file name, Kiara suggested to Leslie that she name objects so they are easy to understand. It is much easier to guess what might be in an object called `states` than what might be in an object called `var123`. Kiara remembered when she used to use generic names for her variables and was asked to revise a table for an important report that she had finished a few months earlier. It took her *hours* to figure out what she meant in the code when all the variables were named things like `x1` and `x2`.

Kiara mentioned that, in addition to choosing meaningful names, some letters and words are already used by R and will cause some confusion if used as object names. For example, the uppercase letters `T` and `F` are used in the code as shorthand for `TRUE` and `FALSE`, so they are not useful as object names. When possible, use words and abbreviations that are not common mathematical terms.

1.6.3.1 NAMING CONSTANTS

Kiara explained that there are recommended methods for naming objects in R that depend on the type of object (“Google’s R Style Guide,” n.d.). There are several types of objects in R. The `states` object is a *constant* because it is a single numeric value. The recommended format for constants is starting with a “k” and then using *camel case*. Camel case is capitalizing the first letter of each word in the object name, with the exception of the first word (the capital letters kind of look like camel humps 🐫). Leslie thought she understood and wanted to correct the naming of the `states` object. Kiara said she could make an entirely new object from scratch, like this:

```
# make a new object with well-formatted name
kStates <- 29
```

Or, she could assign the existing `states` object to a new name, like this:

```
# assign the existing states object a new name
kStates <- states
```

Leslie noticed that this uses the same format with the `<-` as assigning the value of 29 to `states`. Kiara explained that this arrow assigns whatever is on the right side of the arrow to the object name on the left. In this case, the `states` object is assigned to the `kStates` object name. Leslie noticed that `states` and `kStates` are now both listed in the environment. This was unnecessary since they both hold the same information. Kiara showed her how to remove an object using the `rm()` function.

```
# remove the states object
rm(states)
```

This looked different from what they had been doing so far since it included a function, `rm()`, and the name of an object (`states`). Kiara said that this format is common in R, having some instruction or function for R and a set of parentheses like `function()`. Then, inside the parentheses is typically the name of one or more objects to apply the function to, so `function(object)` is a common thing to see when using R. The information inside the parentheses is called an **argument**, so the `states` object is the argument entered into the `rm()` function. Sometimes, R functions will need one argument to work, and sometimes they will require multiple arguments. Arguments do not all have to be objects; some are just additional instructions for R about how the functions should work. Leslie was a little confused by this, but Kiara said it would become more clear as they learned more functions. Kiara wanted to mention one last thing before the next topic. She explained that it is common for R users to call `rm()` and other functions “commands” instead of “functions,” and these two words tend to be used interchangeably by many R users.

1.6.3.2 NAMING VARIABLES

Another type of object is a variable. Variables are measures of some characteristic for each observation in a data set. For example, `income` and `voted` are both variables. Variable objects are named using **dot case** or camel case. Dot case puts a dot between words in a variable name while camel case capitalizes each word in the variable name (except the first word 🐾). For example, if Leslie measured the number of medical marijuana prescriptions filled by each cancer patient in a data set during a year, she could use dot case and call the variable `filled.script.month` or use camel case and call it `filledScriptMonth`. Kiara mentioned that dot case and camel case are frequently used, and there are other variable naming conventions used by some R users (see Box 1.5).

1.6.3.3 NAMING FUNCTIONS

Functions are objects that perform a series of R commands to do something in particular. They are usually written when someone has to do the same thing multiple times and wants to make the process more



1.5 Kiara's reproducibility resource: Naming variables

Using useful names for variables in code will improve clarity. For example, a variable named `bloodPressure` probably contains blood pressure information, while a variable named `var123` could be anything. A couple of widely used practices for naming variables are as follows:

- Use nouns for variable names like `age`, `income`, or `religion`.
- Use dot case or camel case to separate words in multiple-word variable names.
 - `blood.pressure` uses dot case with a period separating words
 - `bloodPressure` is camel case with capital letters starting each word, except for the first word

efficient. Kiara explained that writing functions is a more advanced skill that they would cover later. For now, she just wanted to give the naming format for a function, which is camel case with the first letter capitalized (this is also called “upper camel case” or “*Pascal case*”). For example, a function that multiplies everything in a data set by 2 might be called `MultiplyByTwo` or something similar.

Kiara explained to Leslie that when she first starts to code, she should develop a coding style and begin using a consistent way of annotating code and one of the recommended ways of naming things. Kiara preferred dot case for variable names and using underscores for file names. Leslie agreed. Nancy thought one more thing was worth mentioning while they were talking about writing clear code. Lines of code can get far too long to read without annoying side scrolling, especially on small laptop screens. The recommended limit for the length of a line of code is 80 characters, but shorter is even better. Leslie wrote this down to make sure she remembered this new detail later when she was writing more complicated code.

1.6.4 ACHIEVEMENT 2: CHECK YOUR UNDERSTANDING

Open a new Script file (or modify the existing file if you have been following along) and create a prolog. Make a constant named `kIllegalNum` and assign it the value of 21. Subtract 2 from the `kIllegalNum` object and check the output to find the value.

1.7 Achievement 3: Understanding and changing data types

Kiara explained to Leslie that objects like `kStates` are interpreted by R as one of several *data types*. To see what data type `kStates` is, Kiara demonstrated the `class()` function, like this:

```
# identify data type for states object
class(x = kStates)
## [1] "numeric"
```

1.7.1 NUMERIC DATA TYPE

In the case of the `kStates` object, R prints the data type *numeric* from the `class()` function. The numeric data type is the default that R assigns to constants and variables that contain only numbers. The numeric data type can hold whole numbers and numbers with decimal places, so it is the most appropriate data type for variables measured along a continuum, or *continuous* variables. For example, both height and temperature can be measured along a continuum and would usually be a numeric data type in R.

To practice, Leslie created a constant that contains the ounces of medical marijuana legally available to purchase per person in Rhode Island, then used `class()` to identify the data type. As she wrote, she annotated the code.

```
# assign Rhode Island limit for medical marijuana
# in ounces per person
kOuncesRhode <- 2.5

# identify the data type for kOuncesRhode
class(x = kOuncesRhode)
## [1] "numeric"
```

1.7.2 INTEGER DATA TYPE

The *integer* data type is similar to numeric but contains only whole numbers. There are true integers that can only be measured in whole numbers, like the number of cars parked in a lot. There are also things that could be numeric but are measured as integers, like measuring age as *age in years*. When a whole number is assigned to a variable name in R, the default type is numeric. To change the variable type to integer, use the R function `as.integer()`. The `as.integer()` function can also be used to truncate numbers with decimal places. Note that truncation is not the same as rounding! Truncation cuts off everything after the decimal place. For example, truncating the value 8.9 would leave 8. Rounding goes up or down to the nearest whole number, so 8.9 would round to 9.

Kiara explained to Leslie that the default integer type is not always the best type for the data and had her explore the integer data type.

```
# assign the value of 4 to a constant called kTestInteger
# make sure it is an integer
kTestInteger <- as.integer(x = 4)

# use class() to determine the data type of kTestInteger
class(x = kTestInteger)
## [1] "integer"

# use as.integer() to truncate the constant kOuncesRhode
as.integer(x = kOuncesRhode)
## [1] 2

# multiply the kTestInteger and kOuncesRhode constants
kTestInteger * kOuncesRhode
## [1] 10

# multiply kTestInteger and integer kOuncesRhode constants
kTestInteger * as.integer(x = kOuncesRhode)
## [1] 8

# type the object name to see what is currently saved
# in the object
kOuncesRhode
## [1] 2.5
```

1.7.3 LOGICAL DATA TYPE

The *logical* data type contains the values of TRUE and FALSE. The values of TRUE and FALSE can be assigned to a logical constant, like this:

```
# create the constant
kTestLogical <- TRUE
```

```

# print the value of the constant
kTestLogical
## [1] TRUE

# check the constant data type
class(x = kTestLogical)
## [1] "logical"

```

Logical constants can also be created as the result of some expression, such as the following:

```

# store the result of 6 > 8 in a constant called kSixEight
kSixEight <- 6 > 8

# print kSixEight
kSixEight
## [1] FALSE

# determine the data type of kSixEight
class(x = kSixEight)
## [1] "logical"

```

Because 6 is not greater than 8, the expression `6 > 8` is `FALSE`, which is assigned to the `kSixEight` constant.

1.7.4 CHARACTER DATA TYPE

The *character* data type contains letters, words, or numbers that cannot logically be included in calculations (e.g., a zip code). They are always wrapped in either single or double quotation marks (e.g., `'hello'` or `"world"`). Kiara had Leslie try creating a few character constants.

```

# make constants
kFirstName <- "Corina"
kLastName <- "Hughes"

# check the data type
class(x = kFirstName)
## [1] "character"

# create a zip code constant
# check the data type
kZipCode <- "97405"
class(x = kZipCode)
## [1] "character"

```

Leslie was confused as to why the zip code class was character when it is clearly an integer. Kiara reminded her that putting things in quote marks signifies to R that it is a character data type.

1.7.5 FACTOR DATA TYPE

In addition to the data types above, the *factor* data type is used for constants and variables that are made up of data elements that fall into categories. Variables measured in categories are *categorical*.

Examples of categorical variables can include variables like religion, marital status, age group, and so on. There are two types of categorical variables: *ordinal* and *nominal*. Ordinal variables contain categories that have some logical order. For example, categories of age can logically be put in order from younger to older: 18–25, 26–39, 40–59, 60+. Nominal variables have categories that have no logical order. Religious affiliation and marital status are examples of nominal variable types because there is no logical order to these characteristics (e.g., Methodist is not inherently greater or less than Catholic).

1.7.6 ACHIEVEMENT 3: CHECK YOUR UNDERSTANDING

Check the data type for the `kIllegalNum` constant created in the Check Your Understanding exercise for Achievement 1.

1.8 Achievement 4: Entering or loading data into R

Usually, when social scientists collect information to answer a question, they collect more than one number or word since collecting only one would be extremely inefficient. As a result, there are groups of data elements to be stored together. There are many ways to enter and store information like this. One commonly used object type is a *vector*. A vector is a set of data elements saved as the same type (numeric, logical, etc.). Each entry in a vector is called a *member* or *component* of the vector. Vectors are commonly used to store variables.

1.8.1 CREATING VECTORS FOR DIFFERENT DATA TYPES

The format for creating a vector uses the `c()` function for concatenate. The parentheses are filled with the member of the vector separated by commas. If the members of the vector are meant to be saved as character-type variables, use single or double quotes around each member. Kiara demonstrated creating and printing character, numeric, and logical vectors:

```
# creates character vector char.vector
char.vector <- c('Oregon', 'Vermont', 'Maine')
# prints vector char.vector
char.vector
## [1] "Oregon" "Vermont" "Maine"

# creates numeric vector nums.1.to.4
nums.1.to.4 <- c(1, 2, 3, 4)
# prints vector nums.1.to.4
nums.1.to.4
## [1] 1 2 3 4

# creates logical vector logic.vector
logic.vector <- c(TRUE, FALSE, FALSE, TRUE)
```



```
# prints vector logic.vector
logic.vector
## [1] TRUE FALSE FALSE TRUE
```

Kiara mentioned that she had added a space after each comma when creating the vectors, and that this was one of the good coding practices Leslie should use. While the space is not necessary for the code to work, it does make it easier to read. Leslie wondered why some of the code is a different color. Specifically, she saw the code after a hashtag is a different color from other code. Nancy explained that these are the comments. They can appear alone on a line, or they can be at the end of a line of regular code.

Nancy chimed in with her favorite thing, a coding trick. This one is for creating new objects and printing them at the same time by adding parentheses around the code that creates the object.

```
# create and print vectors
( char.vector <- c('Oregon', 'Vermont', 'Maine') )
## [1] "Oregon" "Vermont" "Maine"
( nums.1.to.4 <- c(1, 2, 3, 4) )
## [1] 1 2 3 4
( logic.vector <- c(TRUE, FALSE, FALSE, TRUE) )
## [1] TRUE FALSE FALSE TRUE
```

The next thing Kiara covered is how vectors can be combined, added to, subtracted from, subsetted, and other operations. She used the `nums.1.to.4` vector to show examples of each of these with comments that explain what is happening with each line of code.

```
# add 3 to each element in the nums.1.to.4 vector
nums.1.to.4 + 3
## [1] 4 5 6 7

# add 1 to the 1st element of nums.1.to.4, 2 to the 2nd element, etc
nums.1.to.4 + c(1, 2, 3, 4)
## [1] 2 4 6 8

# multiply each element of nums.1.to.4 by 5
nums.1.to.4 * 5
## [1] 5 10 15 20

# subtract 1 from each element and then divide by 5
(nums.1.to.4 - 1) / 5
## [1] 0.0 0.2 0.4 0.6

# make a subset of the vector including numbers > 2
nums.1.to.4[nums.1.to.4 > 2]
## [1] 3 4
```

As she read the code, Leslie kept reminding herself that the dots in the middle of the variable names are not decimal points, but are there to separate parts of the variable names, which use dot case.

So far, the results of these operations are just printed in the Console; they are nowhere to be found in the Environment pane. Leslie asked how to keep the results. Kiara explained that the results could be assigned to a new vector object using the assignment arrow, like this:

```
# add 3 to number vector and save
# as new vector
( nums.1.to.4.plus.3 <- nums.1.to.4 + 3 )
## [1] 4 5 6 7

# divide vector by 10 and save
# as new vector
( nums.1.to.4.div.10 <- nums.1.to.4 / 10 )
## [1] 0.1 0.2 0.3 0.4
```

The results show the original vector with 3 added to each value and the result of that addition divided by 10. The results print in the Console and also are saved and can be found in the Environment pane.

Kiara explained that it is possible to do multiple computations on a single vector.

```
# add 3 and divide by 10 for each vector member
( nums.1.to.4.new <- (nums.1.to.4 + 3) / 10 )
## [1] 0.4 0.5 0.6 0.7
```

1.8.2 CREATING A MATRIX TO STORE DATA IN ROWS AND COLUMNS

In addition to the vector format, Kiara explained that R also uses the *matrix* format to store information. A matrix is information, or data elements, stored in a rectangular format with rows and columns. Coders can perform operations on matrices, or more than one matrix, as with vectors.

The R function for producing a matrix is, surprisingly, `matrix()`. This function takes arguments to enter the data, `data =`, and to specify the number of rows, `nrow =`, and columns, `ncol =`. Kiara explained that the most confusing part of `matrix()` is the `byrow =` argument, which tells R whether to fill the data into the matrix by filling across first (fill row 1, then fill row 2, etc.) or by filling down first (fill column 1 first, then fill column 2, etc.). In this case, Kiara chose `byrow = TRUE` so the data fill across first. For the columns to fill first, she would have to use `byrow = FALSE` instead.

```
# create and print a matrix
( policies <- matrix(data = c(1, 2, 3, 4, 5, 6),      #data in the matrix
                    nrow = 2,                      # number of rows
                    ncol = 3,                      # number of columns
                    byrow = TRUE) )                # fill the matrix by rows

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

Say the matrix includes the number of states with policies legalizing medical, recreational, and both types of marijuana that were in effect in 2013 and 2014. Leslie asked about naming the rows and columns so she can remember what they are. Kiara started to explain when Nancy jumped in to demonstrate by writing the code, which uses `dimnames()` to assign names to rows and columns. As Nancy typed the code, Kiara explained that the names are entered in vectors inside a list, with the first vector being the row names and the second vector being the column names. In this case, the row names were `c("2013", "2014")` for the two years of data and the column names were `c("medical", "recreational", "both")` for the three types of policy.

```
# add names to the rows and columns of the matrix
dimnames(x = policies) <- list(
  c("2013", "2014"),           # row names
  c("medical", "recreational", "both") # column names
)

# print the policies matrix
policies
##      medical recreational both
## 2013         1             2    3
## 2014         4             5    6
```

Now Leslie could find specific data elements in her matrix, such as the number of states with legal medical marijuana policies in 2014.

Leslie was still trying to remember all the data types and asked what would happen if she had a vector of the types of policies that had passed instead of the number of policies per year. Would this be a factor data type?

```
# vector of policy types
policy.2013.and.2014 <- c('medical', 'medical', 'both', 'recreational',
  'medical', 'both', 'both')

# data type
class(x = policy.2013.and.2014)
## [1] "character"
```

Leslie thought this would be a factor data type since the policy type is a categorical variable, but R assigned the character type to her vector. Kiara explained that she could use the `as.factor()` function to change the variable type to factor instead.

```
# change the data type to factor
policy.2013.and.2014 <- as.factor(x = policy.2013.and.2014)

# check the data type
class(x = policy.2013.and.2014)
## [1] "factor"
```

1.8.3 CREATING A DATA FRAME

Similar to a matrix format, the *data frame* format has rows and columns of data. In the data frame format, rows are observations and columns are variables. Data frames are often entered outside of R into a spreadsheet or other type of file and then imported into R for analysis. However, R users can also make their own data frame using vectors or matrices. For example, if Kiara looked up five states, the year they made medical marijuana legal, and the limit per person in ounces for possession of medical marijuana, she could enter these data into three vectors and combine them into a data frame using the `data.frame()` function, like this:

```
# state, year enacted, personal oz limit medical marijuana
# create vectors
state <- c('Alaska', 'Arizona', 'Arkansas', 'California', 'Colorado')
year.legal <- c('1998', '2010', '2016', '1996', '2000')
ounce.lim <- c(1, 2.5, 3, 8, 2)

# combine vectors into a data frame
# name the data frame pot.legal
pot.legal <- data.frame(state, year.legal, ounce.lim)
```

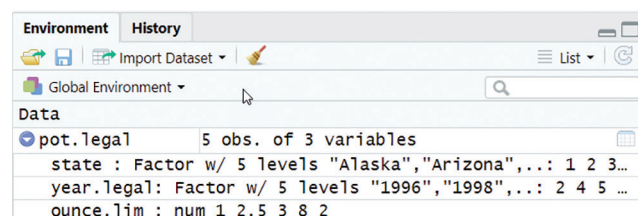
Just like in the `matrix()` function, the `data.frame()` function reads in multiple arguments. The `data.frame()` function has three arguments: `state`, `year.legal`, and `ounce.lim`. This time, all of the arguments are objects, but that will not always be the case. In fact, arguments can even be functions with their own arguments!

After entering and running these code lines, Kiara suggested that Leslie check the Environment pane, where she should now see a new entry called `pot.legal`. To the right of the label `pot.legal`, Leslie saw “5 obs. of 3 variables” indicating she had entered five observations and three variables. The blue and white circle with a triangle in it to the left of `pot.legal` allowed Leslie to expand this entry to see more information about what is contained in the `pot.legal` object, like Figure 1.6.

Leslie noticed in the Environment window that the `state` variable in the `pot.legal` data frame was assigned the variable type of factor, which is incorrect. Names of states are unique and not categories in this data set. Leslie wanted to change the name variable to a character variable using the `as.character()` function.

Because the `state` variable is now part of a data frame object, Kiara explained to Leslie that she would have to identify both the data frame and the variable in order to change it. To demonstrate, Kiara

FIGURE 1.6 Environment window in RStudio showing the newly created data frame



entered the name of the data frame first, a \$ to separate the data frame name, and the variable name, like this:

```
# change state variable from pot.legal data frame
# to a character variable
pot.legal$state <- as.character(x = pot.legal$state)

# check the variable type
class(x = pot.legal$state)
## [1] "character"
```

Now that Leslie had a data frame, there were many options open for data management and analysis. For example, she could examine basic information about the variables in her data by using the `summary()` function. The `summary()` function requires at least one argument that identifies the object that should be summarized, like this:

```
# summarize the data frame
summary(object = pot.legal)
##      state          year.legal    ounce.lim
## Length:5          1996:1         Min.    :1.0
## Class :character  1998:1         1st Qu.:2.0
## Mode  :character  2000:1        Median :2.5
##                               2010:1        Mean   :3.3
##                               2016:1        3rd Qu.:3.0
##                               Max.    :8.0
```

This output looked a little confusing to Leslie, so Kiara explained what she was seeing. The top row contains the names of the three variables in the `pot.legal` data frame. Below each variable is some information about that variable. What is shown there depends on the data type of the variable. The `state` variable is a character variable, so the information below `state` shows how many observations there were for this variable in the `Length:5` row. The next row shows the class of the `state` variable with `Class :character` and the mode or most common value of the variable. Leslie was curious about this use of mode since she had learned it before as a measure of *central tendency*; Kiara explained that mode is one of the descriptive statistics they would talk about next time they met.

The next column of information is for the `year.legal` variable, which is a factor variable. This entry shows each of the categories of the factor and how many observations are in that category. For example, 1996 is one of the categories of the factor, and there is one observation for 1996. Likewise, 2016 is one of the categories of the factor, and there is one observation in this category. Kiara mentioned that this output shows up to six rows of information for each variable in a data frame, and many variables will have more than six categories. When this is the case, the six categories with the most observations in them will be shown in the output of `summary()`.

Finally, the `ounce.lim` column, based on the `ounce.lim` numeric variable, shows `Min. :1.0`, which indicates that the minimum value of this variable is 1. This column also shows `Max. :8.0` for the maximum value of 8 and a few other *descriptive statistics* that Kiara assured Leslie the R-Team would discuss more the next time they meet.

1.8.4 IMPORTING DATA FRAMES FROM OUTSIDE SOURCES

Kiara mentioned that, while typing data directly into R is possible and sometimes necessary, most of the time analysts like Leslie will open data from an outside source. R is unique among statistical software packages because it has the capability of importing and opening data files saved in most formats. Some formats open directly in the base version of R. Other data formats require the use of an *R package*, which Kiara reminded Leslie is a special program written to do something specific in R.

To know what format a data file is saved in, examine the file extension. Common file extensions for data files are as follows:

- **.csv**: comma separated values
- **.txt**: text file
- **.xls** or **.xlsx**: Excel file
- **.sav**: *SPSS* file
- **.sasb7dat**: SAS file
- **.xpt**: SAS transfer file
- **.dta**: Stata file

1.8.5 IMPORTING A COMMA SEPARATED VALUES (CSV) FILE

Kiara added that, in addition to knowing which kind of file a data file is, Leslie would need to know the location of the file. R can open files saved locally on a computer, in an accessible shared location, or directly from the Internet. The file Nancy analyzed at the beginning of the day was saved in csv format online. There are several possible ways to read in this type of file; the most straightforward way is with the `read.csv()` function; however, Kiara warned that this function may sometimes result in misreading of variable names or row names and to look out for that.

While the GSS data can be read into R directly from the GSS website, Kiara had experienced this and knew that it could be frustrating. Since this was Leslie's first time importing data into R from an external source, Kiara decided they should try a more straightforward example. She saved two of the variables from the GSS data Nancy imported for Figure 1.1 and made the data file available at edge.sagepub.com/harris1e with the file name `legal_weed_age_GSS2016_ch1.csv`.

Kiara explained to Leslie that it might work best to make a data folder inside the folder where she is keeping her code and save the downloaded data there. Once Leslie created the folder and saved the data, Kiara explained that the `read.csv()` function could be used to import the data from that folder location, like this:

```
# read the GSS 2016 data
gss.2016 <- read.csv(file = "[data folder location]/data/legal_weed_age_
GSS2016_ch1.csv")

# examine the contents of the file
summary(object = gss.2016)
##           i..grass      age
## DK           : 110    57      : 70
## IAP          : 911    58      : 67
```



```
## LEGAL      :1126  52      : 65
## NOT LEGAL:  717  53      : 60
## NA's      :    3  27      : 58
##
##              (Other):2537
##
##              NA's   :  10
```

The `summary()` function output shows two column headings, `i.grass` and `age`. Kiara pointed out the strange `i.grass` heading should probably just be `grass`, and this is an example of what can happen with `read.csv()`. These two column headings are the two variables in the data set. Under each of these headings are two more columns separated by colons. The column before the colon lists the values that are present in the variable. The column after the colon lists the number of times the value is present in the variable. The information under the `i.grass` column heading shows that `DK` is one of the values of the `i.grass` variable, and this value occurs 110 times.

Kiara advised Leslie that the `fread()` function in the `data.table` package or the `read_csv()` function in the `tidyverse` package might be more useful for opening `csv` files saved from online sources. To install a package, go to the Tools menu in RStudio and select `Install Packages...` Type “`data.table`” or the name of whichever package should be installed in the dialog box that opens. For other ways to install packages, see Box 1.6.



1.6 Nancy's fancy code: Working with R packages

The basic R functions included with R can do a lot, but not everything. Additional functions are included in packages developed by researchers and others around the world and contributed to the R open-source platform. We will use many of these packages throughout this text. One that was used to create the plots above was `ggplot2`, which is available as a standalone package or as part of the `tidyverse` package. To use a package, it first has to be installed. There are at least two ways to install an R package. One is to use the Tools menu and select `Install Packages...` and choose or type the exact name of the package you want to install.

The second way is to use the R `install.packages()` function, like this:

```
install.packages(pkgs = "tidyverse")
```

Using `Install Packages...` from the Tools menu may work best because installing a package is a *one-time* task, so writing code is not very efficient.

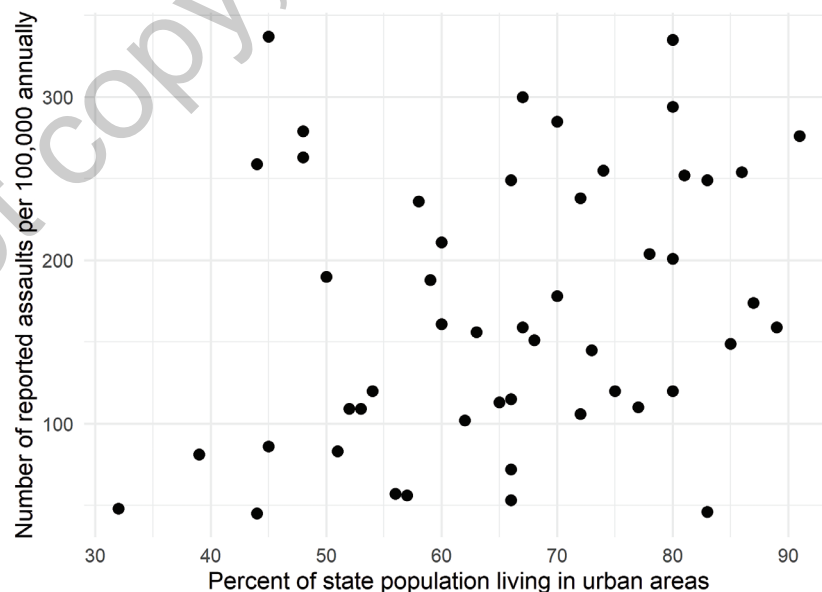
To use `ggplot2` after `tidyverse` is installed, it has to be opened. Unlike installing, every time a package is used, it must be opened first. This is similar to other software programs. For example, Microsoft Word is installed once but opened every time it is used. Use the `library()` function to open an R package.

```
library(package = "tidyverse")
```

Once a package is open, all of its functions are available. Run the code below to make Figure 1.7 from one of the data sets built into R. The USArrests data set includes information on assaults, murder, rape, and percentage of the population living in urban areas for all 50 states. Note that the code may seem confusing now since the `ggplot2` functions are complicated, but code to make great graphs will be explored and repeated throughout the upcoming meetings.

```
# pipe the data set into ggplot (Figure 1.7)
# in the aesthetics provide the variable names for the x and y axes
# choose a geom for graph type
# add axis labels with labs
# choose a theme for the overall graph look
USArrests %>%
  ggplot(aes(x = UrbanPop, y = Assault)) +
  geom_point() +
  labs(x = "Percent of state population living in urban areas",
       y = "Number of reported assaults per 100,000 annually") +
  theme_minimal()
```

FIGURE 1.7 Urban population and assaults at the state level from USArrests built-in R data source



Once a package is installed, Kiara explained, there are two ways Leslie could use it. If she wanted to use more than one or two functions from the package, she could open it with the `library()` function. In this case, `library(package = "data.table")` opens the `data.table` package for use. Once the package is open, Leslie could use the functions defined in the package. Also useful, explained Kiara, is the documentation for each package that shows all the functions available (“Available CRAN Packages,” n.d.; Dowle & Srinivasan, 2019).

Using `library(package =)` is the common practice for opening and using packages most of the time in R. Once the package is opened using `library(package =)`, it stays open until R is closed. When using a function from a package one time, it is not necessary to open the package and leave it open. Instead, there is another way to open a package temporarily just to use a particular function. To temporarily open a package in order to use a function from the package, add the package name before the function name and separate with two colons, like this: `package.name::function()`.

At Kiara’s suggestion, Leslie installed `data.table` and used the temporary way with the `::` to open the `data.table` package and used the fast and friendly file finagler `fread()` function from the package to open the GSS data file. She then used the summary function to see what was in the file.

```
# bring in GSS 2016 data
gss.2016 <- data.table::fread(input = "[data folder location]/data/legal_
weed_age_GSS2016_ch1.csv")

# examine the contents of the file
summary(object = gss.2016)
##      grass          age
## Length:2867      Length:2867
## Class :character Class :character
## Mode  :character Mode  :character
```

Leslie noticed that the variable names now look better, but both the variables now seem to be character variables, so they might have to use `as.factor()` and `as.numeric()` to fix the data types before using these variables.

Before they continued, Kiara wanted to mention another important benefit of the `::` way of opening a package for use. Occasionally, two different packages can have a function with the same name. If two packages containing function names that are the same are opened at the same time in an R file, there will be a *namespace* conflict where R cannot decide which function to use. One example is the function `summarize()`, which is included as part of the `dplyr` package and the `Hmisc` package. When both packages are open, using the `summarize()` function results in an error. Kiara explained that the `dplyr` package is loaded with the `tidyverse`. To demonstrate the error, she installed and opened `tidyverse` and `Hmisc`.

```
# load Hmisc and tidyverse
library(package = "tidyverse")
library(package = "Hmisc")
```

Kiara typed the `summarize()` function to try to get the length of the `age` variable from the `gss.2016` data frame. Kiara noted that this code is more complicated than what they had looked at so far, and they will go through the formatting soon, but for now, this is just to demonstrate how the use of the `summarize()` function results in a conflict when both `dplyr` and `Hmisc` are open.

```
# use the summarize function
gss.2016 %>%
  summarize(length.age = length(x = age))
```

The result of running her code is the error message in Figure 1.8.

FIGURE 1.8 Namespace error with `summarize()` function

```
Error in summarize(., length.age = length(x = age)) :
argument "by" is missing, with no default
```

Kiara mentioned that this was relatively rare, but it was a good thing to keep in mind when a function does not run. Leslie asked what could be done at this point. Kiara said there are a couple of ways to check to see if a namespace conflict is occurring. The first is to use the `conflicts()` function.

```
# check for conflicts
conflicts()
## [1] "%>%"          "%>%"          "src"          "summarize"
## [5] "%>%"          "discard"     "col_factor"  "%>%"
## [9] "add_row"     "as_data_frame" "as_tibble"   "data_frame"
## [13] "data_frame_" "frame_data"  "glimpse"     "lst"
## [17] "lst_"        "tbl_sum"     "tibble"     "tribble"
## [21] "trunc_mat"   "type_sum"    "alpha"      "enexpr"
## [25] "enexprs"     "enquo"       "enquos"     "ensym"
## [29] "ensyms"     "expr"        "quo"        "quo_name"
## [33] "quos"       "sym"         "syms"       "vars"
## [37] "between"    "first"       "last"       "transpose"
## [41] "filter"     "lag"         "body<-"     "format.pval"
## [45] "intersect"  "kronecker"   "Position"   "setdiff"
## [49] "setequal"   "union"       "units"
```

Among the conflicts, Leslie saw the `summarize()` function. Kiara said the easiest thing to do to address the conflict is to use the `::` and specify which package to get the `summarize()` function from. To use the `summarize()` function from `dplyr`, the code would look like this:

```
# use summarize from dplyr
gss.2016 %>%
  dplyr::summarize(length.age = length(x = age))
##   length.age
## 1       2867
```

The function now works to find the length of the `age` variable. Another way to check and see if a function is in conflict after an error message is to use the `environment()` function and check which package is the source for the `summarize()` function. Kiara wrote the code to do this:

```
# check source package for summarize
environment(fun = summarize)
## <environment: namespace:Hmisc>
```

The output shows that the namespace for `summarize()` is the **Hmisc** package instead of the **dplyr** package. Use of the `::` works to fix this, but another strategy would be to detach the **Hmisc** package before running `summarize()`, like this:

```
# detach Hmisc
detach(name = package:Hmisc)

# try summarize
gss.2016 %>%
  summarize(length.age = length(x = age))
##   length.age
## 1         2867
```

This works too! If the package is not needed again, this method of addressing the namespace conflict will avoid additional conflicts in the document.

1.8.6 CLEANING DATA TYPES IN AN IMPORTED FILE

After that long detour, Kiara went back to the task at hand. She noted for Leslie that, while the variable names look good after loading with `fread()`, both of the variables were the character data type. Leslie knew that most data sets have a codebook that lists all the variables and how they were measured. This information would help her to identify what data types are appropriate for variables and other information about the data. She checked the codebook for the GSS (National Opinion Research Center, 2019) that was saved as [gss_codebook.pdf](https://edge.sagepub.com/harris1e) at edge.sagepub.com/harris1e to determine what data types these variables are. On page 304 of the codebook, it shows the measurement of the variable `grass`, which has five possible responses:

- Do you think the use of marijuana should be made legal or not?
 - Should
 - Should not
 - Don't know
 - No answer
 - Not applicable

Leslie remembered that variables with categories are categorical and should be factor type variables in R.

Leslie tried to find the `age` variable in the codebook, but the codebook is difficult to use because it is too long. Kiara suggested Leslie look on the GSS Data Explorer website for more information about the `age` variable and how it is measured. Leslie found and reviewed the `age` variable in the Data Explorer, and it appears to be measured in years up to age 88, and then “89 OR OLDER” represents people who are 89 years old or older.

Kiara suggested Leslie use the `head()` function to get a sense of the data. This function shows the first six observations.

```
# first six observations in the gss.2016 data frame
head(x = gss.2016)
##      grass age
## 1:    IAP 47
## 2:   LEGAL 61
## 3: NOT LEGAL 72
## 4:    IAP 43
## 5:   LEGAL 55
## 6:   LEGAL 53
```

After she viewed the six observations, Leslie determined `grass` should be a factor and `age` should be numeric. Kiara agreed, but before Leslie wrote the code, Nancy added that since “89 OR OLDER” is not saved as just a number, trying to force the `age` variable with “89 OR OLDER” in it into a numeric variable would result in an error. She suggested that before converting `age` into a numeric variable, they should first *recode* anyone who has a value of “89 OR OLDER” to instead have a value of “89.” Nancy explained that this will ensure that `age` can be treated as a numeric variable. Kiara warned that they will need to be careful in how they use and report this recoded `age` variable since it would be inaccurate to say that every person with the original “89 OR OLDER” label was actually 89 years old. However, Nancy reminded Kiara that they were going to look at age categories like she did for Figure 1.2 and that changing the “89 OR OLDER” people to have an age of 89 would be OK for making a categorical age variable. Their plan was to change the `grass` variable to a factor and recode the `age` variable before changing it to either numeric or integer. When it is unclear whether to choose between numeric and integer data types, numeric is more flexible.

Leslie started with converting `grass` into a factor. Because she was *not* changing the contents of the variables, she kept the same variable names. To do this, she used the arrow to keep the same name for the variable with the new assigned type. Kiara pointed out the data frame name and variable name on the left of the assignment arrow `<-` are exactly the same as on the right. When new information is assigned to an existing variable, it overwrites whatever was saved in that variable.

```
# change grass variable to a factor
gss.2016$grass <- as.factor(x = gss.2016$grass)
```

For the trickier bit of recoding `age`, Nancy took over.

```
# recode the 89 OR OLDER category to 89
gss.2016$age[gss.2016$age == "89 OR OLDER"] <- "89"
```

Nancy explained that this line of code can be read as follows: “In the `age` variable of the `gss.2016` data frame, find any observation that is equal to ‘89 OR OLDER’ and assign those particular observations to be the character ‘89.’” Kiara reassured Leslie that even though this particular line of code is tricky, it would be covered in more detail later and Leslie would surely get the hang of it.

Leslie went back and tried to integrate what she had learned.

```
# bring in GSS 2016 data
gss.2016 <- data.table::fread(input = "[data folder location]/data/legal_
weed_age_GSS2016_ch1.csv")

# change the variable type for the grass variable
gss.2016$grass <- as.factor(x = gss.2016$grass)

# recode "89 OR OLDER" into just "89"
gss.2016$age[gss.2016$age == "89 OR OLDER"] <- "89"

# change the variable type for the age variable
gss.2016$age <- as.numeric(x = gss.2016$age)

# examine the variable types and summary to
# check the work
class(x = gss.2016$grass)
## [1] "factor"
class(x = gss.2016$age)
## [1] "numeric"
summary(object = gss.2016)
##           grass           age
## DK           : 110   Min.      :18.00
## IAP           : 911   1st Qu.:34.00
## LEGAL         :1126   Median  :49.00
## NOT LEGAL     : 717   Mean     :49.16
## NA's          :    3   3rd Qu.:62.00
##                Max.     :89.00
##                NA's     :10
```

Leslie used `class()` and `summary()` to check and confirm that the variables were now the correct type.

1.8.7 ACHIEVEMENT 4: CHECK YOUR UNDERSTANDING

Use `fread()` to open the GSS 2016 data set. Look in the Environment pane to find the number of observations and the number and types of variables in the data frame.

1.9 Achievement 5: Identifying and treating missing values

In addition to making sure the variables used are an appropriate type, Kiara explained that it was also important to make sure that missing values were treated appropriately by R. In R, missing values are recorded as `NA`, which stands for *not available*. Researchers code missing values in many different

ways when collecting and storing data. Some of the more common ways to denote missing values are the following:

- blank
- 777, -777, 888, -888, 999, -999, or something similar
- a single period
- -1
- NULL

Other responses, such as “Don’t know” or “Inapplicable,” may sometimes be treated as missing or as response categories depending on what is most appropriate given the characteristics of the data and the analysis goals.

1.9.1 RECODING MISSING VALUES TO NA

In the summary of the GSS data, the `grass` variable has five possible values: DK (don’t know), IAP (inapplicable), LEGAL, NOT LEGAL, and NA (not available). The DK, IAP, and NA could all be considered missing values. However, R treats only NA as missing. Before conducting any analyses, the DK and IAP values could be converted to NA to be treated as missing in any analyses. That is, the `grass` variable could be recoded so that these values are all NA. Note that NA is a reserved “word” in R. In order to use NA, both letters must be uppercase (Na or na does not work), and there can be no quotation marks (R will treat “NA” as a character rather than a true missing value).

There are many ways to recode variables in R. Leslie already saw one way, using Nancy’s bit of code for the `age` variable. Kiara’s favorite way uses the data management package **tidyverse** (<https://www.rdocumentation.org/packages/tidyverse/>). Kiara closed her laptop to show Leslie one of her laptop stickers. It shows the **tidyverse** logo in a hexagon. Kiara explained that R users advertise the packages they use and like with hexagonal laptop stickers. It is not unusual, she said, to see a laptop covered in stickers like the one in Figure 1.9.

FIGURE 1.9 tidyverse hex laptop sticker



Source: RStudio.

Since Leslie had installed and opened **tidyverse**, they could start on data management. Kiara mentioned that if a package is not installed before using `library()` to open it, the `library` function will show an error.

Kiara showed Leslie the pipe feature, `%>%`, that is available in the **tidyverse** package and useful for data management and other tasks. The `%>%` works to send or *pipe* information through a function or set of functions. In this case, Kiara said, they would pipe the `gss.2016` data set into a `mutate()` function that can be used to recode values. The `mutate()` function takes the name of the variable to recode and then information on how to recode it. Kiara thought it might just be best to show Leslie the code and walk through it.

```
# start over by bringing in the data again
gss.2016 <- data.table::fread(input = "[data folder location]/data/legal_
weed_age_GSS2016_ch1.csv")

# use tidyverse pipe to change DK to NA
gss.2016.cleaned <- gss.2016 %>%
  mutate(grass = as.factor(x = grass)) %>%
  mutate(grass = na_if(x = grass, y = "DK"))

# check the summary, there should be 110 + 3 in the NA category
summary(object = gss.2016.cleaned)
##      grass      age
## DK      :    0  Length:2867
## IAP     :  911  Class :character
## LEGAL   :1126  Mode  :character
## NOT LEGAL:  717
## NA's    :  113
```

Kiara walked Leslie through the data management code. First is the `gss.2016.cleaned <-`, which indicates that whatever happens on the right-hand side of the `<-` will be assigned to the `gss.2016.cleaned` object name. The first thing after the arrow is `gss.2016 %>%`, which indicates that the `gss.2016` data are being piped into whatever comes on the next line; in this case, it is being piped into the `mutate()` function. The `mutate()` function on the next line uses the `na_if()` function to make the `grass` variable equal to `NA` if the `grass` variable is currently coded as `DK`.

Leslie was a little confused but tried to summarize what the code did. First she asked Kiara why they now have a new name for the data with `gss.2016.cleaned`. Kiara explained that it is good practice to keep the original data unchanged in case you need to go back to it later. Then Leslie said she believed the function was changing the `DK` values in the `grass` variable to `NA`, which is R shorthand for missing. Kiara said that was correct and it was completely fine to be confused. She admitted it just took a while to get used to the way R works and the way different structures like the `%>%` work. Kiara thought maybe adding the `IAP` recoding to the code might be useful for reinforcing the ideas. She added to her code to replace `IAP` with `NA`.

```

# use tidyverse pipe to change DK and IAP to NA
gss.2016.cleaned <- gss.2016 %>%
  mutate(grass = as.factor(x = grass)) %>%
  mutate(grass = na_if(x = grass, y = "DK")) %>%
  mutate(grass = na_if(x = grass, y = "IAP"))

# check the summary, there should now be 110 + 911 + 3 in the NA category
summary(object = gss.2016.cleaned)
##      grass      age
## DK       :    0  Length:2867
## IAP      :    0  Class :character
## LEGAL    :1126  Mode  :character
## NOT LEGAL: 717
## NA's     :1024

```

That worked!

Leslie found the summary information accurate, with zero observations coded as DK or IAP. However, the DK and IAP category labels were still listed even though there are no observations with these coded values. Kiara explained that R will keep all the different levels of a factor during a recode, so Leslie would need to remove unused categories with a `droplevels()` function if she no longer needed them. Leslie wanted to try this herself and added a line of code to Kiara's code.

```

# use tidyverse pipe to change DK and IAP to NA
gss.2016.cleaned <- gss.2016 %>%
  mutate(grass = as.factor(x = grass)) %>%
  mutate(grass = na_if(x = grass, y = "DK")) %>%
  mutate(grass = na_if(x = grass, y = "IAP")) %>%
  mutate(grass = droplevels(x = grass))

# check the summary
summary(object = gss.2016.cleaned)
##      grass      age
## LEGAL    :1126  Length:2867
## NOT LEGAL: 717  Class :character
## NA's     :1024  Mode  :character

```

Leslie was pretty excited that she had figured it out! She asked Kiara if she could do the change of data type for the `grass` and `age` variables in the same set of functions as the recoding of NA values. Kiara

thought this was a great idea and suggested that they do one more recoding task and combine the recoding of `age` and `grass` functions so they would have everything all together. Nancy explained that this means a slightly different way of recoding the “89 OR OLDER” observations using `mutate()`. Leslie was excited to see this! It worked well to have all the data management together in one place.

In addition to adding the `age` and `grass` recoding, the final function to add was to create the age categories shown in Figure 1.2. The `age` variable currently holds the age in years rather than age categories. The graph Kiara made at the beginning showed age in four categories:

- 18–29
- 30–59
- 60–74
- 75+

Kiara suggested naming the categorical `age` variable `age.cat`. She clarified that this is not referring to the age of actual cats, but instead is for the categories of ages. Nancy rolled her eyes at this joke attempt. Kiara was unfazed and showed Leslie the function `cut()`, which can be used to divide a continuous variable into categories by cutting it into pieces and adding a label to each piece. Leslie added `as.numeric()` and `as.factor()` to the `mutate()` functions in the set of data management tasks and then asked Kiara for help with the `cut()` function. Kiara explained that `cut` takes a variable like `age` as the first argument, so it would look like `cut(x = age, .`

The second thing to add after the variable name is a vector made up of the *breaks*. Breaks specify the lower and upper limit of each category of values. The first entry is the lowest value of the first category, the second entry is the highest value of the first category, the third entry is the highest value of the second category, and so on. The function now looks like `cut(x = age, breaks = c(-Inf, 29, 59, 74, Inf), .` Leslie noticed that the first and last values in the vector are `-Inf` and `Inf`. She guessed that these are negative infinity and positive infinity. Kiara confirmed that this was correct and let Leslie know that this was for convenience rather than looking up the smallest and largest values. It also makes the code more flexible in case there is a new data point with a smaller or larger value.

The final thing to add is a vector made up of the *labels* for the categories, with each label inside quote marks, like this: `labels = c("< 30", "30 - 59", "60 - 74", "75+")`. The final `cut()` function would include these three things. Leslie gave it a try.

```
# use tidyverse to change data types and recode
gss.2016.cleaned <- gss.2016 %>%
  mutate(age = recode(.x = age, "89 OR OLDER" = "89")) %>%
  mutate(age = as.numeric(x = age)) %>%
  mutate(grass = as.factor(x = grass)) %>%
  mutate(grass = na_if(x = grass, y = "DK")) %>%
  mutate(grass = na_if(x = grass, y = "IAP")) %>%
  mutate(grass = droplevels(x = grass)) %>%
  mutate(age.cat = cut(x = age,
                      breaks = c(-Inf, 29, 59, 74, Inf),
                      labels = c("< 30", "30 - 59", "60 - 74", "75+" )))
summary(object = gss.2016.cleaned)
```

```
##      grass      age      age.cat
## LEGAL      :1126  Min.    :18.00  < 30   : 481
## NOT LEGAL:  717  1st Qu.:34.00  30 - 59:1517
## NA's      :1024  Median :49.00  60 - 74: 598
##                               Mean    :49.16  75+    : 261
##                               3rd Qu.:62.00  NA's   :   10
##                               Max.    :89.00
##                               NA's    :10
```

Kiara thought they were ready to try the last task for the day, making a bar chart. First, though, she asked Leslie to practice putting all her code together and adding a prolog. She suggested Leslie make one change, which is to rename the data `gss.2106.cleaned` after all the data management and cleaning. This way she would have both the original `gss.2016` and the cleaned version of the data if she needed both.

```
#####
# Project: First R-team meeting
# Purpose: Clean GSS 2016 data
# Author: Leslie
# Edit date: April 20, 2019
# Data: GSS 2016 subset of age and marijuana use variables
#####

# bring in GSS 2016 data from the web and examine it
library(package = "data.table")
gss.2016 <- fread(file = "[data folder location]/data/legal_weed_age_
GSS2016_ch1.csv")

# use tidyverse to clean the data
library(package = "tidyverse")
gss.2016.cleaned <- gss.2016 %>%
  mutate(age = recode(.x = age, "89 OR OLDER" = "89")) %>%
  mutate(age = as.numeric(x = age)) %>%
  mutate(grass = as.factor(x = grass)) %>%
  mutate(grass = na_if(x = grass, y = "DK")) %>%
  mutate(grass = na_if(x = grass, y = "IAP")) %>%
  mutate(grass = droplevels(x = grass)) %>%
  mutate(age.cat = cut(x = age,
                       breaks = c(-Inf, 29, 59, 74, Inf),
                       labels = c("< 30", "30 - 59", "60 - 74", "75+" )))
```



```
# check the summary
summary(object = gss.2016.cleaned)
##      grass      age      age.cat
## LEGAL      :1126 Min.   :18.00 < 30   : 481
## NOT LEGAL: 717 1st Qu.:34.00 30 - 59:1517
## NA's      :1024 Median :49.00 60 - 74: 598
##              Mean   :49.16 75+   : 261
##              3rd Qu.:62.00 NA's   : 10
##              Max.   :89.00
##              NA's   :10
```

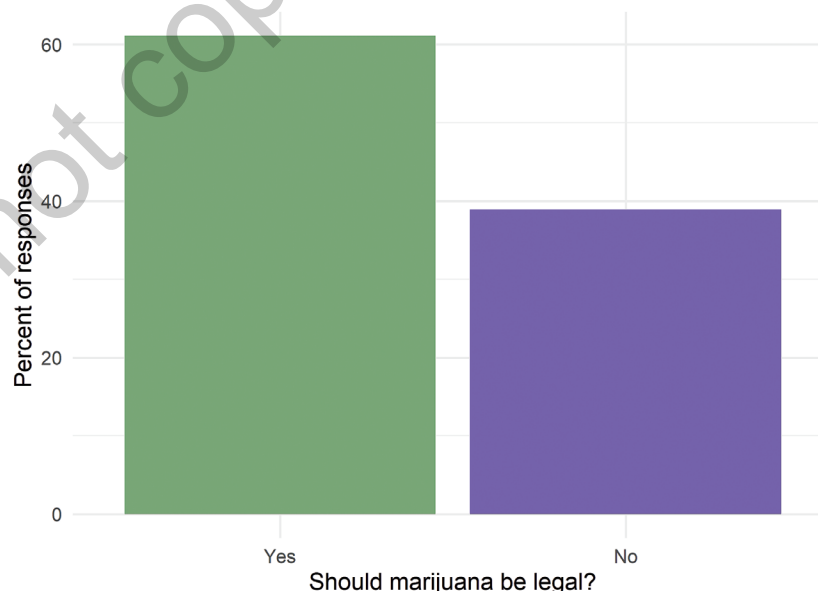
1.9.2 ACHIEVEMENT 5: CHECK YOUR UNDERSTANDING

Describe what `mutate()`, `na_if()`, and `%>%` did in the final code Leslie wrote.

1.10 Achievement 6: Building a basic bar chart

Leslie was now ready to finish up a *very* long first day of R by creating the graph from the beginning of their meeting. Kiara introduced her to an R package called `ggplot2` to create this graph. The “gg” in `ggplot2` stands for the “grammar of graphics.” The `ggplot2` package (<https://www.rdocumentation.org/packages/ggplot2/>) is part of the `tidyverse`, so it did not need to be installed or opened separately, and creating the graph would use some of the `tidyverse` skills from the data management. Before Kiara showed Leslie how to make the first graph, she examined it one more time (Figure 1.10).

FIGURE 1.10 Support for marijuana legalization among participants in the 2016 General Social Survey



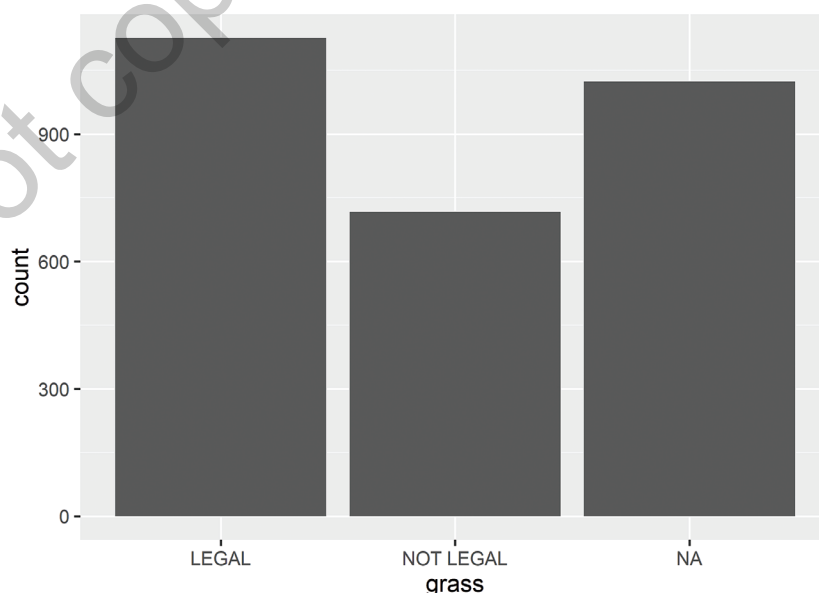
Kiara pointed out some of the important features of the graph:

- Both axes have titles.
- The y -axis is a percentage.
- The x -axis is labeled as Yes and No.
- It includes an overall title.
- The background for the graph is white with a light gray grid.
- The Yes bar is green, and the No bar is purple.

With Kiara's help, Leslie started with a basic plot using the `ggplot()` function. Kiara advised her to store the graph in a new object with a new name. Leslie chose `legalize.bar`. Kiara said to follow a similar structure to the data management from earlier. She started by piping the `gss.2016.cleaned` data frame. Kiara said the data would be piped into the `ggplot()` function this time. The `ggplot()` function needs to know which variable(s) from `gss.2016.cleaned` will be placed on the axes. In the *grammar of graphics*, this information is considered *aesthetics* and is included in the `aes()` function within the `ggplot()` function. There is only one variable for this graph, the `grass` variable, which is on the x -axis. Kiara helped Leslie write the code. After the basics of the graph were included in the `ggplot()` function, the graph type was added in a new *layer*.

Kiara explained that graphs built with `ggplot()` are built in layers. The first layer starts with `ggplot()` and contains the basic information about the data that are being graphed and which variables are included. The next layer typically gives the graph type, or *geometry* in the grammar of graphics language, and starts with `geom_` followed by one of the available types. In this case, Leslie was looking for a bar chart, so `geom_bar()` is the geometry for this graph. Leslie started to write this by adding a `%>%` after the line with `ggplot()` on it, but Kiara stopped her. The `geom_bar()` is not a separate new function, but is a layer of the plot and so is added with a `+` instead of a `%>%`. Leslie typed the code to create Figure 1.11.

FIGURE 1.11 Support for marijuana legalization among participants in the 2016 General Social Survey



```
# make a bar chart for grass variable (Figure 1.11)
legalize.bar <- gss.2016.cleaned %>%
  ggplot(aes(x = grass)) +
  geom_bar()

# show the chart
legalize.bar
```

Leslie was happy it worked, even though it looked wrong. Kiara was happy, too—this was a great result for a first use of `ggplot()`. One of the first things Leslie noticed was that there were three bars instead of two. The missing values are shown as a bar in the graph. In some cases, Leslie might be interested in including the missing values as a bar, but for this graph she was interested in comparing the Yes and No values and wanted to drop the NA bar from the graphic.

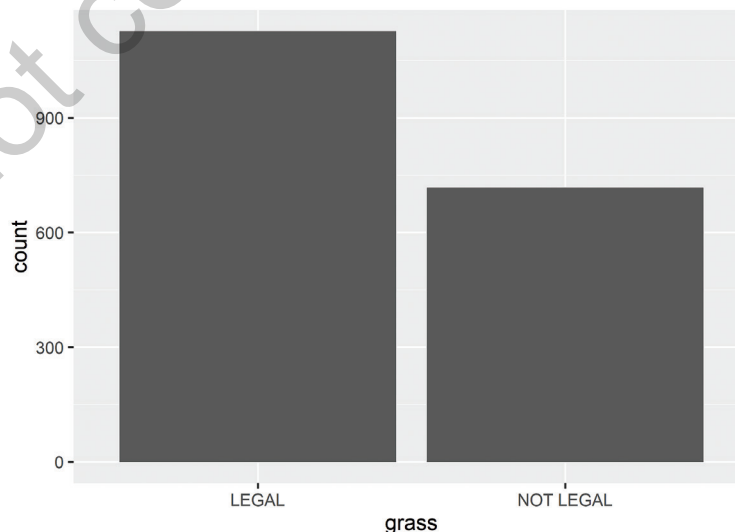
1.10.1 OMITTING NA FROM A GRAPH

Kiara explained that there are many ways to remove the NA bar from the graph, but one of the easiest is adding `drop_na()` to the code. In this case, the NA should be dropped from the `grass` variable. To drop the NA values before the graph, Kiara suggested Leslie add `drop_na()` above `ggplot()` in the code. Leslie gave it a try, creating Figure 1.12.

```
# make a bar chart for grass variable (Figure 1.12)
legalize.bar <- gss.2016.cleaned %>%
  drop_na(grass) %>%
  ggplot(aes(x = grass)) +
  geom_bar()

# show the chart
legalize.bar
```

FIGURE 1.12 Support for marijuana legalization among participants in the 2016 General Social Survey



1.10.2 WORKING WITH COLOR IN A BAR CHART

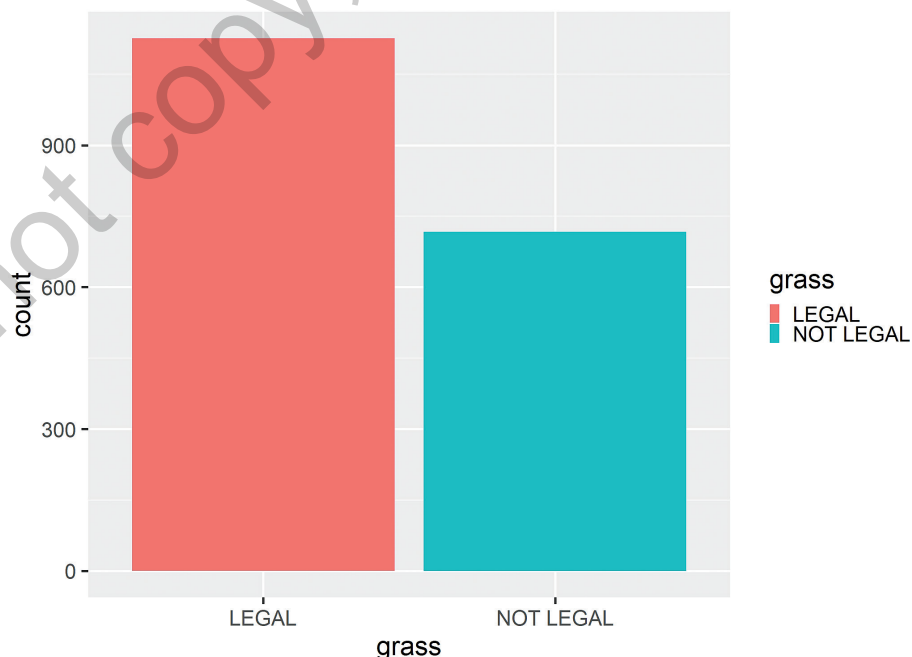
Well, that was easy! Leslie thought. She wanted to work on the look of the graph next. She noticed that there was a light gray background instead of white and that the bars were dark gray instead of green and purple. Kiara introduced Leslie to the concept of `fill =`. To fill the bars with color based on a category of the `grass` variable, the aesthetic needs to have `fill =` specified. Leslie looked at Kiara with a blank stare. Kiara reached over to Leslie's keyboard and added `fill = grass` to the aesthetic so that the bars would each be filled with a different color for each category of the `grass` variable (Figure 1.13).

```
# make a bar chart for grass variable (Figure 1.13)
legalize.bar <- gss.2016.cleaned %>%
  drop_na(grass) %>%
  ggplot(aes(x = grass, fill = grass)) +
  geom_bar()

# show the chart
legalize.bar
```

Leslie wasn't sure that was any closer to correct since the colors weren't right and there was now a legend to the right of the graph that was redundant with the x-axis. Kiara told her not to worry, they could fix both of these things with a single added layer. The `scale_fill_manual()` layer allows the selection of colors for whatever argument is included in `fill =`, and it also has a `guide =` option to specify whether

FIGURE 1.13 Support for marijuana legalization among participants in the 2016 General Social Survey.



or not the legend appears. Kiara added the `scale_fill_manual()` function with the options for Leslie (Figure 1.14).

```
# make a bar chart for grass variable (Figure 1.14)
legalize.bar <- gss.2016.cleaned %>%
  drop_na(grass) %>%
  ggplot(aes(x = grass, fill = grass)) +
  geom_bar() +
  scale_fill_manual(values = c("#78A678", "#7463AC"),
                    guide = FALSE)

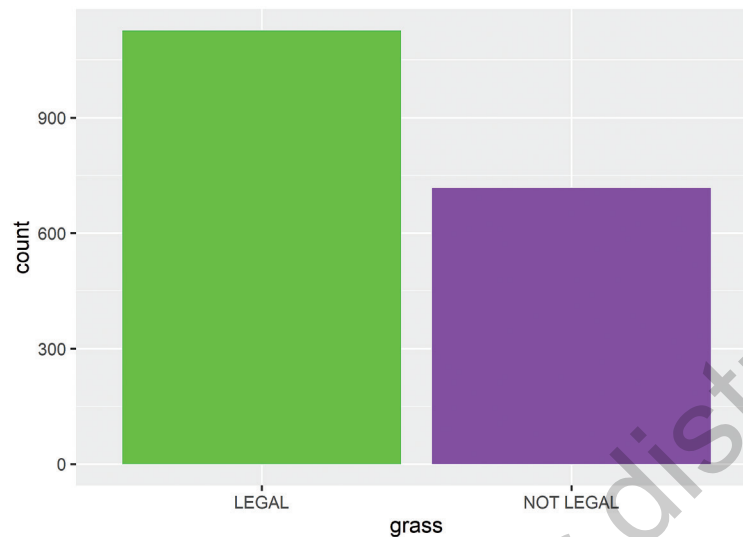
# show the chart
legalize.bar
```

It was starting to look good, thought Leslie. Still, she wondered about the meaning of the values 78A678 and 7463AC. Kiara said those are **RGB** (red-green-blue) codes that specify colors. She told Leslie that the Color Brewer 2.0 website (<http://colorbrewer2.org>) is a great place to find RGB codes for colors that work well for different sorts of graphs, are color-blind safe, and work with printing or copying. The names of colors can also be used; for example, after replacing the codes with the words “green” and “purple,” the graph will look like Figure 1.15.

FIGURE 1.14 Support for marijuana legalization among participants in the 2016 General Social Survey



FIGURE 1.15 Support for marijuana legalization among participants in the 2016 General Social Survey



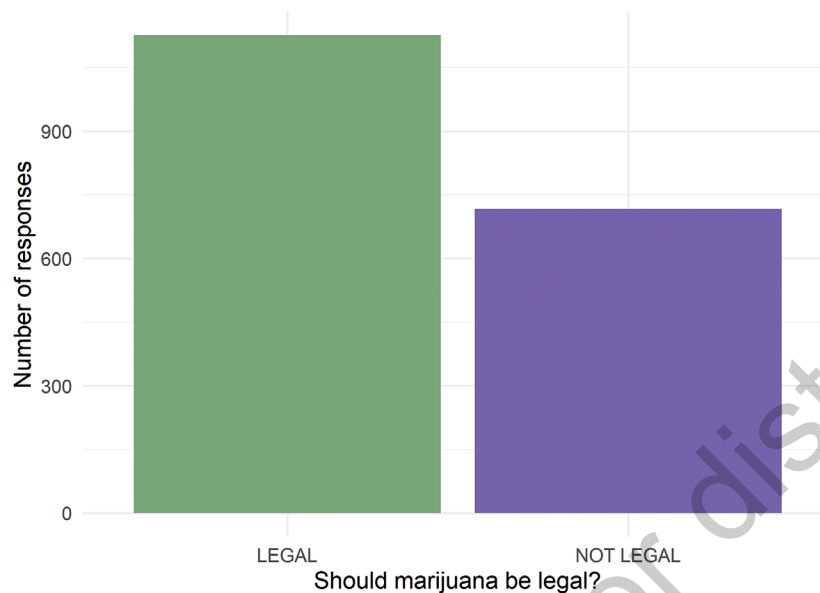
```
# make a bar chart for grass variable (Figure 1.15)
legalize.bar <- gss.2016.cleaned %>%
  drop_na(grass) %>%
  ggplot(aes(x = grass, fill = grass)) +
  geom_bar() +
  scale_fill_manual(values = c("green", "purple"),
                    guide = FALSE)

# show the chart
legalize.bar
```

Yikes! Leslie thought the RGB codes were much better and changed the code back to show the original green and purple. The rest of the R-Team agreed, and Nancy mentioned that besides RGB codes, there are also R color palettes that mimic the color schemes in scientific journals and from shows like *Star Trek* and *Game of Thrones* (Xiao & Li, 2019).

Leslie decided that next she would like to remove the gray background and add the labels to the *x*-axis and *y*-axis. Kiara let her know that the background is part of a theme and that there are many themes to choose from (Wickham, n.d.). The theme that Nancy used in the original graphs was the *minimal* theme, which uses minimal color so that printing the graph requires less ink. This sounded great to Leslie. Kiara said this theme can be applied by adding another layer using `theme_minimal()`. She said another layer for the labels can be added using the `labs()` function with text entered for *x* = and *y* =. Nancy was starting to get bored and really wanted to help with the coding, so she asked Leslie if she could take over and let Leslie direct her. Leslie agreed and directed her to add the theme and the labels. Nancy added the theme layer and the labels layer and ran the code for Figure 1.16.

FIGURE 1.16 Support for marijuana legalization among participants in the 2016 General Social Survey



```
# make a bar chart for grass variable (Figure 1.16)
legalize.bar <- gss.2016.cleaned %>%
  drop_na(grass) %>%
  ggplot(aes(x = grass, fill = grass)) +
  geom_bar() +
  scale_fill_manual(values = c("#78A678", "#7463AC"),
                    guide = FALSE) +
  theme_minimal() +
  labs(x = "Should marijuana be legal?",
       y = "Number of responses")

# show the chart
legalize.bar
```

This looked great but Leslie was starting to get tired of this graph and the code was really complicated. Nancy was so fast at the coding, which looked confusing to Leslie. Leslie started to think that she was in over her head.

Kiara reassured her that everyone felt this way when they started to learn `ggplot()`. Like many things, `ggplot()` and R will make more sense with practice and time. Kiara encouraged her to make the last two changes that were needed. First, the *y*-axis should be a percentage. Second, the labels on the *x*-axis should be Yes and No.

1.10.3 USING SPECIAL VARIABLES IN GRAPHS

To get the *y*-axis to show percentage rather than count, the *y*-axis uses *special variables* with double periods around them. Special variables are statistics computed from a data set; the *count* special variable

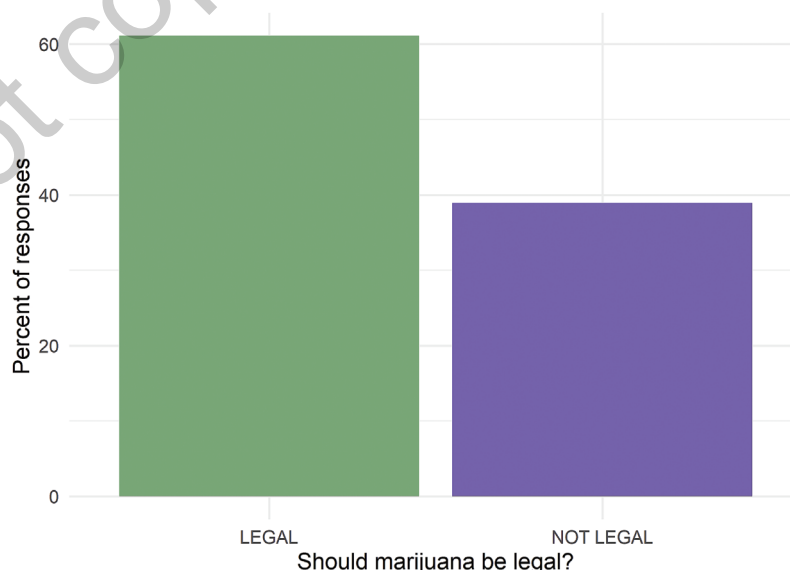
counts the number of observations in the data set. Nancy saw that Leslie was tired and thought that now was her opportunity to do some more coding. She slid the laptop away from Leslie and added the special variables to the aesthetics using `..count..` to represent the *frequency* of a category, or how often it occurred, and `sum(..count..)` to represent the sum of all the frequencies. She multiplied by 100 to get a percentage in Figure 1.17.

```
# make a bar chart for grass variable (Figure 1.17)
legalize.bar <- gss.2016.cleaned %>%
  drop_na(grass) %>%
  ggplot(aes(x = grass,
             y = 100 * (..count..) / sum(..count..),
             fill = grass)) +
  geom_bar() +
  theme_minimal() +
  scale_fill_manual(values = c("#78A678", "#7463AC"),
                   guide = FALSE) +
  labs(x = "Should marijuana be legal?",
       y = "Percent of responses")

# show the chart
legalize.bar
```

The last thing to do was to recode the levels of the `grass` variable to be Yes and No. Kiara nodded at Nancy and Nancy added the final code needed with `mutate()` and `recode_factor()` to create Figure 1.18. Leslie thought the `mutate()` with `recode_factor()` looked complicated, and Nancy assured her they would practice it many times in their next few meetings.

FIGURE 1.17 Support for marijuana legalization among participants in the 2016 General Social Survey



```

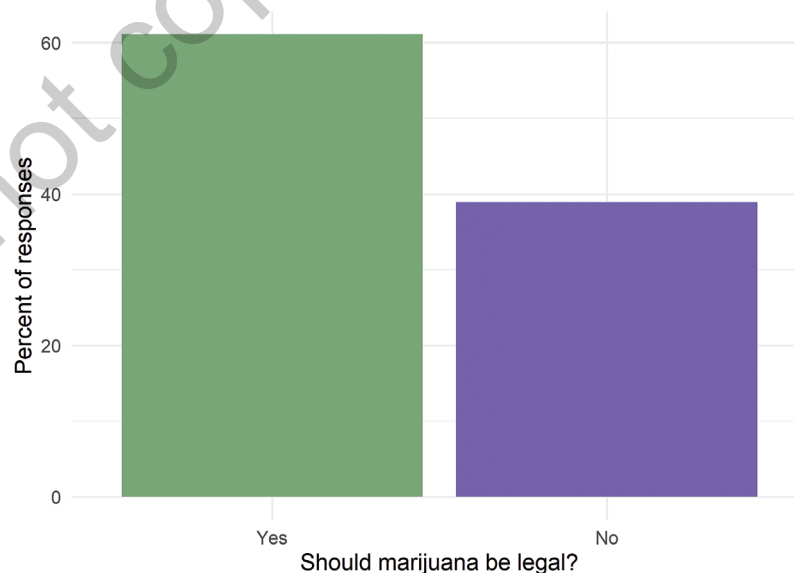
# make a bar chart for grass variable (Figure 1.18)
legalize.bar <- gss.2016.cleaned %>%
  drop_na(grass) %>%
  mutate(grass = recode_factor(.x = grass,
                              `LEGAL` = "Yes",
                              `NOT LEGAL` = "No")) %>%
  ggplot(aes(x = grass,
             y = 100 * (..count..) / sum(..count..),
             fill = grass)) +
  geom_bar() +
  theme_minimal() +
  scale_fill_manual(values = c("#78A678", "#7463AC"),
                   guide = FALSE) +
  labs(x = "Should marijuana be legal?",
       y = "Percent of responses")

# show the chart
legalize.bar

```

Kiara wanted to show Leslie one more trick to add the `age.cat` variable into the graphic, but she realized Leslie had had about enough `ggplot()`. Kiara told Nancy what she wanted, and Nancy wrote the code with Leslie looking over her shoulder. She changed the `x`-axis variable in the aesthetics to be `x = age.cat`, removed the `guide = FALSE` from the `scale_fill_manual()` layer, changed the `x`-axis label, and added `position = 'dodge'` in the `geom_bar()` layer. The code `position = 'dodge'` makes the bars for Yes and No in each age category show side by side (Figure 1.19).

FIGURE 1.18 Support for marijuana legalization among participants in the 2016 General Social Survey



```

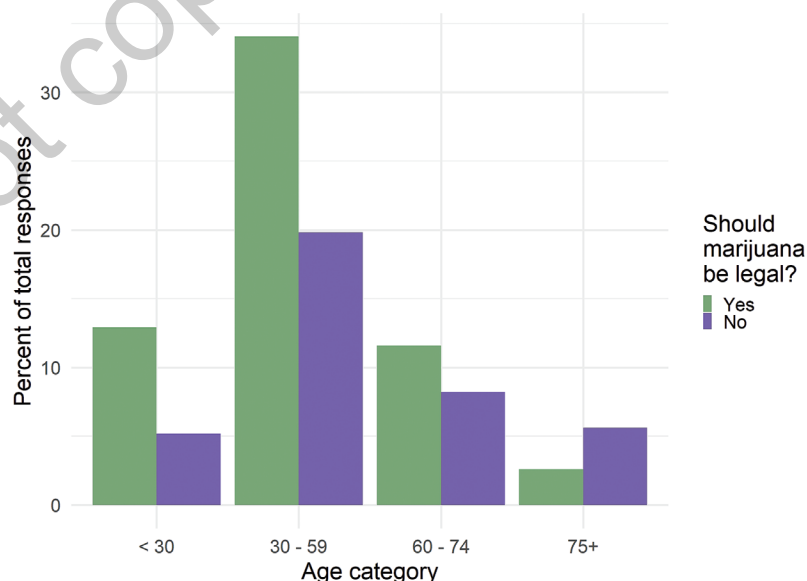
# adding dodge to show bars side by side (Figure 1.19)
legalize.bar <- gss.2016.cleaned %>%
  drop_na(grass) %>%
  drop_na(age) %>%
  mutate(grass = recode_factor(.x = grass,
                              `LEGAL` = "Yes",
                              `NOT LEGAL` = "No")) %>%

  ggplot(aes(x = age.cat,
             y = 100*(..count..)/sum(..count..),
             fill = grass)) +
  geom_bar(position = 'dodge') +
  theme_minimal() +
  scale_fill_manual(values = c("#78A678", "#7463AC"),
                    name = "Should marijuana\nbe legal?") +
  labs(x = "Age category",
       y = "Percent of total responses")
legalize.bar

```

Finally, the full graph appeared. Leslie was overwhelmed by what seemed to be hundreds of layers in `ggplot()` to create a single graph. Kiara and Nancy both reassured her that this is complicated coding and she will start to understand it more as she practices. They planned to do a whole day about graphs soon, but they wanted her to see the power of R early on. Leslie noticed that while the general pattern was the same, Figure 1.19 showed different percentages than Figure 1.2. Nancy explained that the bars in Figure 1.19 added up to 100% total, whereas the bars in Figure 1.2 added up to 100% in each age

FIGURE 1.19 Support for marijuana legalization among participants in the 2016 General Social Survey



group. She said that this required the use of some additional R coding tricks that they would get to in their next two meetings. Leslie was curious about the code, so Nancy showed her as a preview, but suggested they wait to discuss all the new functions until next time. Leslie agreed that this was a good idea, given all the R ideas swimming around in her head already.

```
# code to create Figure 1.2
legalize.bar <- gss.2016.cleaned %>%
  drop_na(grass) %>%
  drop_na(age) %>%
  mutate(grass = recode_factor(.x = grass,
                              `LEGAL` = "Yes",
                              `NOT LEGAL` = "No")) %>%
  group_by(grass, age.cat) %>%
  count() %>%
  group_by(age.cat) %>%
  mutate(perc.grass = 100*n/sum(n)) %>%
  ggplot(aes(x = age.cat, fill = grass,
            y = perc.grass)) +
  geom_col(position = 'dodge') +
  theme_minimal() +
  scale_fill_manual(values = c("#78A678", "#7463AC"),
                    name = "Should marijuana\nbe legal?") +
  labs(x = "Age group (in years)",
       y = "Percent of responses in age group")
legalize.bar
```

The R-Team learned a little more about marijuana policy during their introduction to R day. They found that more people support legalizing marijuana than do not support it. They also found that support for legalization is higher in younger age categories, so support is likely to increase as those who are currently in the younger categories get older over time. Leslie mentioned that this seems important for state-level policymakers to consider. Even if legalization does not have enough support currently, this may change over time, and state government officials might start paying attention to the challenges and benefits realized by those states that have already adopted legal marijuana policies.

1.10.4 ACHIEVEMENT 6: CHECK YOUR UNDERSTANDING

Think about the number of missing values for `grass` (after `DK` and `IAP` were converted to `NA`s) and `age`. Run the `summary` function to confirm. If 1,836 of the 2,867 observations had values for both `grass` and `age`, 10 observations were missing `age`, and 1,024 observations were missing `grass`, then how many observations were missing values for both?

Now try some visual changes to the graph. Change the No bar to the official R-Ladies purple color 88398a and change the Yes bar to the color gray40. Change the theme to another theme of your choice by selecting from the `ggplot2` themes available online (see Wickham, n.d.).

/// 1.11 CHAPTER SUMMARY

1.11.1 Achievements

unlocked in this chapter: Recap

Congratulations! Like Leslie, you've learned and practiced a lot in this chapter.

1.11.1.1 Achievement 1 recap: Observations and variables

R is a coding language in which information is stored as objects that can then be used in calculations and other procedures. Information is assigned to an object using the `<-`, which puts the information on the right of the arrow into an object name included to the left of the arrow. To print the object in R, type its name and use one of the methods for running code.

Data are stored in many formats, including vectors for single variables and matrix and data frame formats for rows and columns. In data frames, rows typically hold observations (e.g., people, organizations), while columns typically hold variables (e.g., age, revenue).

1.11.1.2 Achievement 2 recap: Using reproducible research practices

It is important to think about how to write and organize R code so it is useful not only right now but in the future for anyone (including the original author) to use. Some of the practices to start with are using comments to explain code, limiting lines of code to 80 characters, naming variables with logical names and consistent formatting, naming files with useful human and machine readable information, and including a prolog in each code file.

1.11.1.3 Achievement 3 recap: Understanding and changing data types

Most statistics are appropriate for only certain types of data. R has several data types, with the more commonly used ones being numeric, integer, factor, character, and logical. The `class` function can be used to check a data type, and the appropriate `as` function (e.g., `as.factor()`) can be used to change data types.

1.11.1.4 Achievement 4 recap: Entering or loading data into R

R is unique in its ability to load data from most file formats. Depending on what file type the data are saved as,

a different R function can be used. For example, `read.csv()` will read in a data file saved in the comma separated values (csv) format.

1.11.1.5 Achievement 5 recap: Identifying and treating missing values

In addition to making sure the variables used are an appropriate type, it is also important to make sure that missing values are treated appropriately by R. In R, missing values are recorded as `NA`, which stands for *not available*. Researchers code missing values in many different ways when collecting and storing data. Some of the more common ways to denote missing values are blank, `777`, `-777`, `888`, `-888`, `999`, `-999`, a single period, `-1`, and `NULL`. Other responses, such as “Don't know” or “Inapplicable,” may sometimes be treated as missing or treated as response categories depending on what is most appropriate given the characteristics of the data and the analysis goals. There are many ways to recode these values to be `NA` instead.

1.11.1.6 Achievement 6 recap: Building a basic bar chart

One of the biggest advantages to using R is the ability to make custom graphics. The `ggplot2`, or grammar of graphics, package is useful for making many types of graphs, including bar charts. The `ggplot()` function takes the name of the data frame object and the name of the variable within the data frame that you would like to graph. Layers following the initial function are used to change the look of the graph and refine its elements.

1.11.2 Chapter exercises

The coder and hacker exercises are an opportunity to apply the skills from this chapter to a new scenario or a new data set. The coder edition evaluates the application of the concepts and code learned in this R-Team meeting to scenarios similar to those in the meeting. The hacker edition evaluates the use of the concepts and code from this R-Team meeting in new scenarios, often going a step beyond what was explicitly explained.

The coder edition might be best for those who found some or all of the Check Your Understanding activities to be challenging, or if you needed review before picking the correct responses to the multiple-choice questions. The hacker edition might be best if the Check Your Understanding

activities were not too challenging and the multiple-choice questions were a breeze.

The multiple-choice questions and materials for the exercises are online at edge.sagepub.com/harris1e.

Q1: Which R data type is most appropriate for a categorical variable?

- Numeric
- Factor
- Integer
- Character

Q2: Which of the following opens `ggplot2`?

- `install.packages("ggplot2")`
- `library(package = "ggplot2")`
- `summary(object = ggplot2)`
- `open(x = ggplot2)`

Q3: The block of text at the top of a code file that introduces the project is called

- library.
- summary.
- prolog.
- pane.

Q4: In a data frame containing information on the age and height of 100 people, the people are the _____ and age and height are the _____.

- observations, variables
- variables, observations
- data, factors
- factors, data

Q5: The results of running R code show in which pane?

- Source
- Environment
- History
- Console

1.11.2.1 Chapter exercises: Coder edition

Use the National Health and Nutrition Examination Survey (NHANES) data to examine marijuana use in the United States. Spend a few minutes looking through the NHANES website (<https://www.cdc.gov/nchs/nhanes/index.htm>) before you begin, including finding the online codebook for the

2013–2014 data. Complete the following tasks to explore whether age is related to marijuana use in the United States.

- Open the 2013–2014 NHANES data file saved as `nhanes_2013_ch1.csv` with the book materials at edge.sagepub.com/harris1e (Achievement 4).
- Examine the data types for `DUQ200`, `RIDAGEYR`, and `RIAGENDR`, and fix data types if needed based on the NHANES codebook (Achievement 3).
- Based on the online NHANES codebook, code missing values appropriately for `DUQ200`, `RIDAGEYR`, and `RIAGENDR` (Achievement 5).
- Create a bar chart showing the percentage of NHANES participants answering yes and no to marijuana use (Achievement 6).
- Recode `age` into a new variable called `age.cat` with 4 categories: 18–29, 30–39, 40–49, 50–59 (Achievement 5).
- Create a bar chart of marijuana use by age group (Achievement 6).
- Add a prolog and comments to your code (Achievement 2).
- Following the R code in your code file, use comments to describe what you found. Given what you found and the information in the chapter, what do you predict will happen with marijuana legalization in the next 10 years? Discuss how the omission of older people from the marijuana use question for NHANES influenced your prediction. Write your prediction and discussion in comments at the end of your code file (Achievement 2).

1.11.2.2 Chapter exercises: Hacker edition

Read the coder instructions and complete #1–#5 from the coder edition. Then do the following:

- Create a bar chart of marijuana use by age group and sex with side-by-side bars (Achievement 6).
- Add a prolog and comments to your code (Achievement 2).
- Following the R code in your code file, use comments to describe what you found in no more than a few sentences. Given what you found and the information in the chapter, what do you predict will happen with marijuana legalization in the next 10 years? Discuss how

the omission of older people from the marijuana use question for NHANES influenced your prediction. Write your prediction and discussion in comments at the end of your code file (Achievement 2).

1.11.2.3 Instructor note

Solutions to exercises can be found on the website for this book, along with ideas for gamification for those who want to take it further.



Visit edge.sagepub.com/harris1e to download the data sets, complete the chapter exercises, and watch R tutorial videos.

Do not copy, post, or distribute