Chapter 1 Getting Started with Data in R

Before we can start exploring data in R, there are some key concepts to understand first:

1. What are R and RStudio?
2. How do I code in R?
3. What are R packages?

We'll introduce these concepts in the upcoming Sections 1.1-1.3. If you are already somewhat familiar with these concepts, feel free to skip to Section 1.4 where we'll introduce our first dataset: all domestic flights departing one of the three main New York City (NYC) airports in 2013. This is a dataset we will explore in depth for much of the rest of this book.

1.1 What are R and RStudio?

Throughout this book, we will assume that you are using R via RStudio. First time users often confuse the two. At its simplest, R is like a car's engine while RStudio is like a car's dashboard as illustrated in Figure 1.1.

R: Engine	RStudio: Dashboard
	

FIGURE 1.1: Analogy of difference between R and RStudio.

More precisely, R is a programming language that runs computations, while RStudio is an *integrated development environment (IDE)* that provides an interface by adding many convenient features and tools. So just as the way of having access to a speedometer, rearview mirrors, and a navigation system makes driving much easier, using RStudio's interface makes using R much easier as well.

1.1.1 Installing R and RStudio

> **Note about RStudio Server or RStudio Cloud**: If your instructor has provided you with a link and access to RStudio Server or RStudio Cloud, then you can skip this section. We do recommend after a few months of working on RStudio Server/Cloud that you return to these instructions to install this software on your own computer though.

You will first need to download and install both R and RStudio (Desktop version) on your computer. It is important that you install R first and then install RStudio.

1. **You must do this first:** Download and install R by going to https://cloud.r-project.org/.
    - If you are a Windows user: Click on "Download R for Windows", then click on "base", then click on the Download link.
    - If you are macOS user: Click on "Download R for (Mac) OS X", then under "Latest release:" click on R-X.X.X.pkg, where R-X.X.X is the version number. For example, the latest version of R as of November 25, 2019 was R-3.6.1.
    - If you are a Linux user: Click on "Download R for Linux" and choose your distribution for more information on installing R for your setup.

2. **You must do this second:** Download and install RStudio at https://www.rstudio.com/products/rstudio/download/.
    - Scroll down to “Installers for Supported Platforms” near the bottom of the page.
    - Click on the download link corresponding to your computer’s operating system.

1.1.2 Using R via RStudio

Recall our car analogy from earlier. Much as we don’t drive a car by interacting directly with the engine but rather by interacting with elements on the car’s dashboard, we won’t be using R directly but rather we will use RStudio’s interface. After you install R and RStudio on your computer, you’ll have two new *programs* (also called *applications*) you can open. We’ll always work in RStudio and not in the R application. Figure 1.2 shows what icon you should be clicking on your computer.

R: Do not open this	RStudio: Open this
	

FIGURE 1.2: Icons of R versus RStudio on your computer.

After you open RStudio, you should see something similar to Figure 1.3. (Note that slight differences might exist if the RStudio interface is updated after 2019 to not be this by default.)

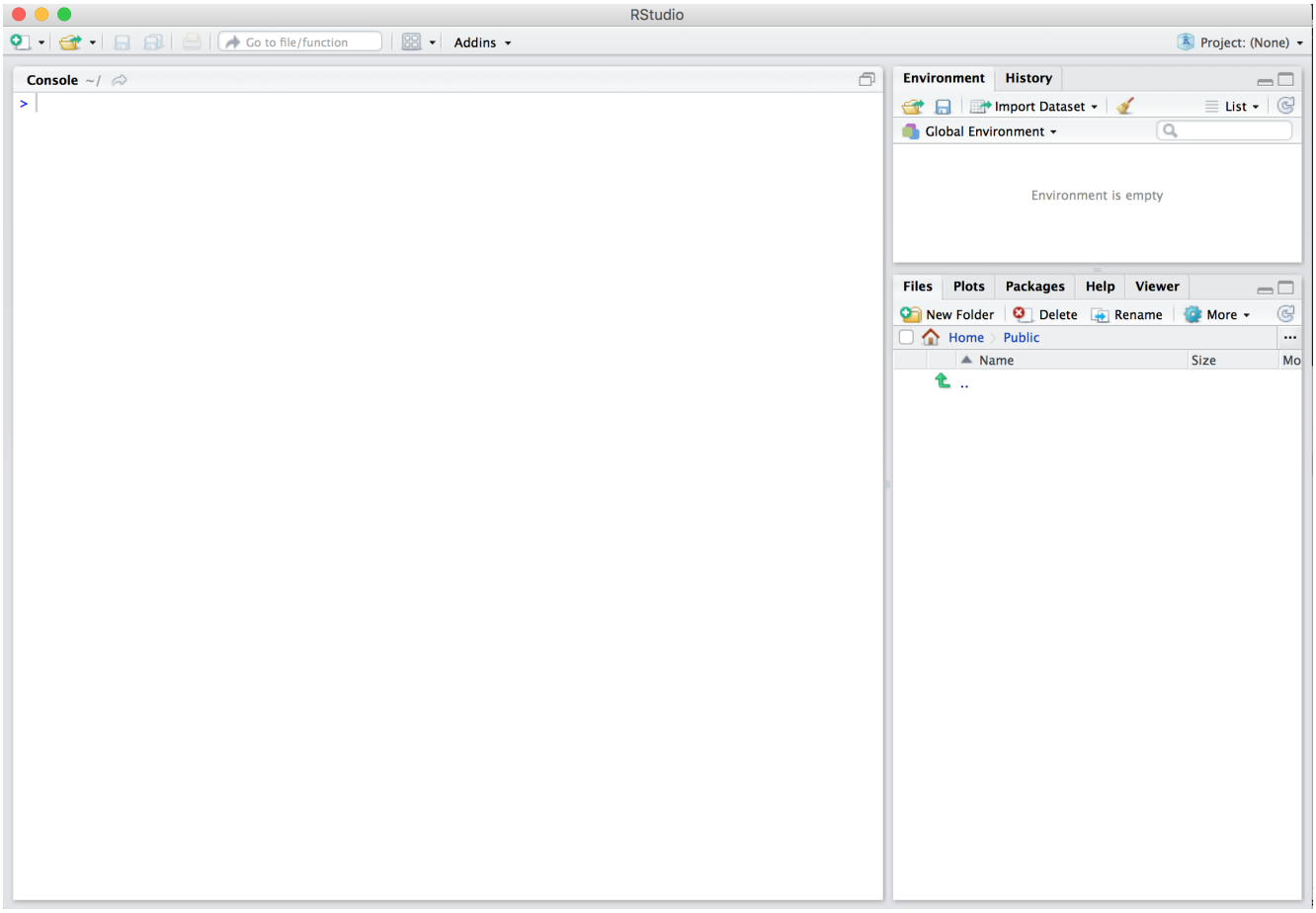

FIGURE 1.3: RStudio interface to R.

Note the three *panes* which are three panels dividing the screen: the *console pane*, the *files pane*, and the *environment pane*. Over the course of this chapter, you'll come to learn what purpose each of these panes serves.

1.2 How do I code in R?

Now that you're set up with R and RStudio, you are probably asking yourself, "OK. Now how do I use R?". The first thing to note is that unlike other statistical software programs like Excel, SPSS, or Minitab that provide point-and-click interfaces, R is an interpreted language. This means you have to type in commands written in *R code*. In other words, you have to code/program in R. Note that we'll use the terms "coding" and "programming" interchangeably in this book.

While it is not required to be a seasoned coder/computer programmer to use R, there is still a set of basic programming concepts that new R users need to understand. Consequently, while this book is not a book on programming, you will still learn just enough of these basic programming concepts needed to explore and analyze data effectively.

1.2.1 Basic programming concepts and terminology

We now introduce some basic programming concepts and terminology. Instead of asking you to memorize all these concepts and terminology right now, we'll guide you so that you'll "learn by doing." To help you learn, we will always use a different font to distinguish regular text from `computer_code`. The best way to master these topics is, in our opinions, through deliberate practice with R and lots of repetition.

- Basics:
  - *Console pane*: where you enter in commands.
  - *Running code*: the act of telling R to perform an act by giving it commands in the console.
  - *Objects*: where values are saved in R. We'll show you how to *assign* values to objects and how to display the contents of objects.
  - *Data types*: integers, doubles/numerics, logicals, and characters. Integers are values like -1, 0, 2, 4092. Doubles or numerics are a larger set of values containing both the integers but also fractions and decimal values like -24.932 and 0.8. Logicals are either `TRUE` or `FALSE` while characters are text such as "cabbage", "Hamilton", "The Wire is the greatest TV show ever", and "This ramen is delicious." Note that characters are often denoted with the quotation marks around them.
- *Vectors*: a series of values. These are created using the `c()` function, where `c()` stands for "combine" or "concatenate." For example, `c(6, 11, 13, 31, 90, 92)` creates a six element series of positive integer values .
- *Factors*: *categorical data* are commonly represented in R as factors. Categorical data can also be represented as *strings*. We'll study this difference as we progress through the book.
- *Data frames*: rectangular spreadsheets. They are representations of datasets in R where the rows correspond to *observations* and the columns correspond to *variables* that describe the observations. We'll cover data frames later in Section 1.4.
- *Conditionals*:
  - Testing for equality in R using `==` (and not `=`, which is typically used for assignment). For example, `2 + 1 == 3` compares `2 + 1` to `3` and is correct R code, while `2 + 1 = 3` will return an error.
  - Boolean algebra: `TRUE/FALSE` statements and mathematical operators such as `<` (less than), `<=` (less than or equal), and `!=` (not equal to). For example, `4 + 2 >= 3` will return `TRUE`, but `3 + 5 <= 1` will return `FALSE`.

    - Logical operators: `&` representing "and" as well as `|` representing "or." For example, `(2 + 1 == 3) & (2 + 1 == 4)` returns `FALSE` since both clauses are not `TRUE` (only the first clause is `TRUE`). On the other hand, `(2 + 1 == 3) | (2 + 1 == 4)` returns `TRUE` since at least one of the two clauses is `TRUE`.
- *Functions*, also called *commands*: Functions perform tasks in R. They take in inputs called *arguments* and return outputs. You can either manually specify a function's arguments or use the function's *default values*.
    - For example, the function `seq()` in R generates a sequence of numbers. If you just run `seq()` it will return the value 1. That doesn't seem very useful! This is because the default arguments are set as `seq(from = 1, to = 1)`. Thus, if you don't pass in different values for `from` and `to` to change this behavior, R just assumes all you want is the number 1. You can change the argument values by updating the values after the `=` sign. If we try out `seq(from = 2, to = 5)` we get the result `2 3 4 5` that we might expect.
    - We'll work with functions a lot throughout this book and you'll get lots of practice in understanding their behaviors. To further assist you in understanding when a function is mentioned in the book, we'll also include the `()` after them as we did with `seq()` above.

This list is by no means an exhaustive list of all the programming concepts and terminology needed to become a savvy R user; such a list would be so large it wouldn't be very useful, especially for novices. Rather, we feel this is a minimally viable list of programming concepts and terminology you need to know before getting started. We feel that you can learn the rest as you go. Remember that your mastery of all of these concepts and terminology will build as you practice more and more.

1.2.2 Errors, warnings, and messages

One thing that intimidates new R and RStudio users is how it reports *errors*, *warnings*, and *messages*. R reports errors, warnings, and messages in a glaring red font, which makes it seem like it is scolding you. However, seeing red text in the console is not always bad.

R will show red text in the console pane in three different situations:

- **Errors**: When the red text is a legitimate error, it will be prefaced with "Error in…" and will try to explain what went wrong. Generally when there's an error, the code will not run. For example, we'll see in Subsection 1.3.3 if you see `Error in ggplot(...) : could not find`

`function "ggplot"` , it means that the `ggplot()` function is not accessible because the package that contains the function (`ggplot2`) was not loaded with `library(ggplot2)` . Thus you cannot use the `ggplot()` function without the `ggplot2` package being loaded first.

- **Warnings**: When the red text is a warning, it will be prefaced with "Warning:" and R will try to explain why there's a warning. Generally your code will still work, but with some caveats. For example, you will see in Chapter 2 if you create a scatterplot based on a dataset where two of the rows of data have missing entries that would be needed to create points in the scatterplot, you will see this warning: `Warning: Removed 2 rows containing missing values (geom_point)` . R will still produce the scatterplot with all the remaining non-missing values, but it is warning you that two of the points aren't there.
- **Messages**: When the red text doesn't start with either "Error" or "Warning", it's *just a friendly message*. You'll see these messages when you load *R packages* in the upcoming Subsection 1.3.2 or when you read data saved in spreadsheet files with the `read_csv()` function as you'll see in Chapter 4. These are helpful diagnostic messages and they don't stop your code from working. Additionally, you'll see these messages when you install packages too using `install.packages()` as discussed in Subsection 1.3.1.

Remember, when you see red text in the console, *don't panic*. It doesn't necessarily mean anything is wrong. Rather:

- If the text starts with "Error", figure out what's causing it. Think of errors as a red traffic light: something is wrong!
- If the text starts with "Warning", figure out if it's something to worry about. For instance, if you get a warning about missing values in a scatterplot and you know there are missing values, you're fine. If that's surprising, look at your data and see what's missing. Think of warnings as a yellow traffic light: everything is working fine, but watch out/pay attention.
- Otherwise, the text is just a message. Read it, wave back at R, and thank it for talking to you. Think of messages as a green traffic light: everything is working fine and keep on going!

1.2.3 Tips on learning to code

Learning to code/program is quite similar to learning a foreign language. It can be daunting and frustrating at first. Such frustrations are common and it is normal to feel discouraged as you learn. However, just as with learning a foreign language, if you put in the effort and are not afraid to make mistakes, anybody can learn and improve.

Here are a few useful tips to keep in mind as you learn to program:

- **Remember that computers are not actually that smart**: You may think your computer or smartphone is "smart," but really people spent a lot of time and energy designing them to appear "smart." In reality, you have to tell a computer everything it needs to do. Furthermore, the instructions you give your computer can't have any mistakes in them, nor can they be ambiguous in any way.
- **Take the "copy, paste, and tweak" approach**: Especially when you learn your first programming language or you need to understand particularly complicated code, it is often much easier to take existing code that you know works and modify it to suit your ends. This is as opposed to trying to type out the code from scratch. We call this the *"copy, paste, and tweak"* approach. So early on, we suggest not trying to write code from memory, but rather take existing examples we have provided you, then copy, paste, and tweak them to suit your goals. After you start feeling more confident, you can slowly move away from this approach and write code from scratch. Think of the "copy, paste, and tweak" approach as training wheels for a child learning to ride a bike. After getting comfortable, they won't need them anymore.
- **The best way to learn to code is by doing**: Rather than learning to code for its own sake, we find that learning to code goes much smoother when you have a goal in mind or when you are working on a particular project, like analyzing data that you are interested in and that is important to you.
- **Practice is key**: Just as the only method to improve your foreign language skills is through lots of practice and speaking, the only method to improving your coding skills is through lots of practice. Don't worry, however, we'll give you plenty of opportunities to do so!

1.3 What are R packages?

Another point of confusion with many new R users is the idea of an R package. R packages extend the functionality of R by providing additional functions, data, and documentation. They are written by a worldwide community of R users and can be downloaded for free from the internet.

For example, among the many packages we will use in this book are the `ggplot2` package (Wickham, Chang, et al. 2023) for data visualization in Chapter 2, the `dplyr` package (Wickham, François, et al. 2023) for data wrangling in Chapter 3, the `moderndive` package (Kim

and Ismay 2022) that accompanies this book, and the `infer` package (Bray et al. 2022) for "tidy" and transparent statistical inference in Chapters 8, 9, and 10.

A good analogy for R packages is they are like apps you can download onto a mobile phone:

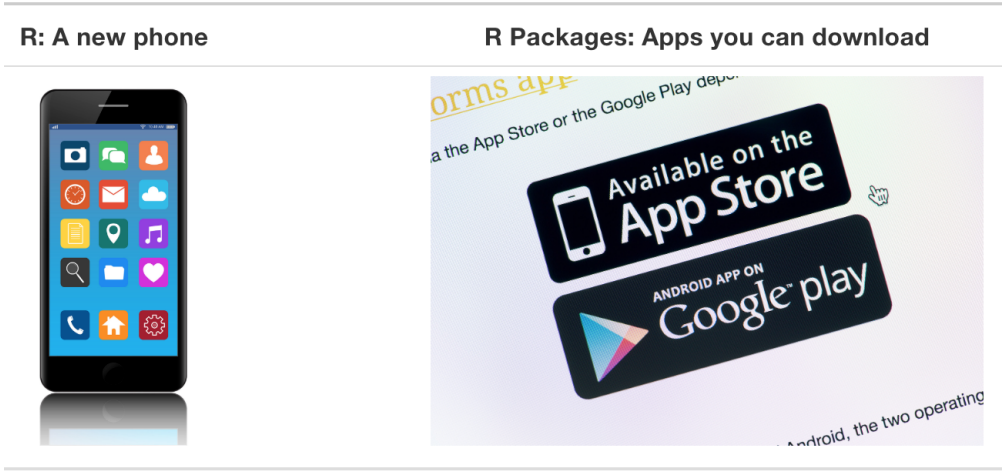

FIGURE 1.4: Analogy of R versus R packages.

So R is like a new mobile phone: while it has a certain amount of features when you use it for the first time, it doesn't have everything. R packages are like the apps you can download onto your phone from Apple's App Store or Android's Google Play.

Let's continue this analogy by considering the Instagram app for editing and sharing pictures. Say you have purchased a new phone and you would like to share a photo you have just taken with friends on Instagram. You need to:

1. *Install the app*: Since your phone is new and does not include the Instagram app, you need to download the app from either the App Store or Google Play. You do this once and you're set for the time being. You might need to do this again in the future when there is an update to the app.
2. *Open the app*: After you've installed Instagram, you need to open it.

Once Instagram is open on your phone, you can then proceed to share your photo with your friends and family. The process is very similar for using an R package. You need to:

1. *Install the package*: This is like installing an app on your phone. Most packages are not installed by default when you install R and RStudio. Thus if you want to use a package for the first time, you need to install it first. Once you've installed a package, you likely won't install it again unless you want to update it to a newer version.
2. *"Load" the package*: "Loading" a package is like opening an app on your phone. Packages are not "loaded" by default when you start RStudio on your computer; you need to "load"

each package you want to use every time you start RStudio.

Let's perform these two steps for the `ggplot2` package for data visualization.

1.3.1 Package installation

> **Note about RStudio Server or RStudio Cloud**: If your instructor has provided you with a link and access to RStudio Server or RStudio Cloud, you might not need to install packages, as they might be preinstalled for you by your instructor. That being said, it is still a good idea to know this process for later on when you are not using RStudio Server or Cloud, but rather RStudio Desktop on your own computer.

There are two ways to install an R package: an easy way and a more advanced way. Let's install the `ggplot2` package the easy way first as shown in Figure 1.5. In the Files pane of RStudio:

a. Click on the "Packages" tab.
b. Click on "Install" next to Update.
c. Type the name of the package under "Packages (separate multiple with space or comma):" In this case, type `ggplot2`.
d. Click "Install."

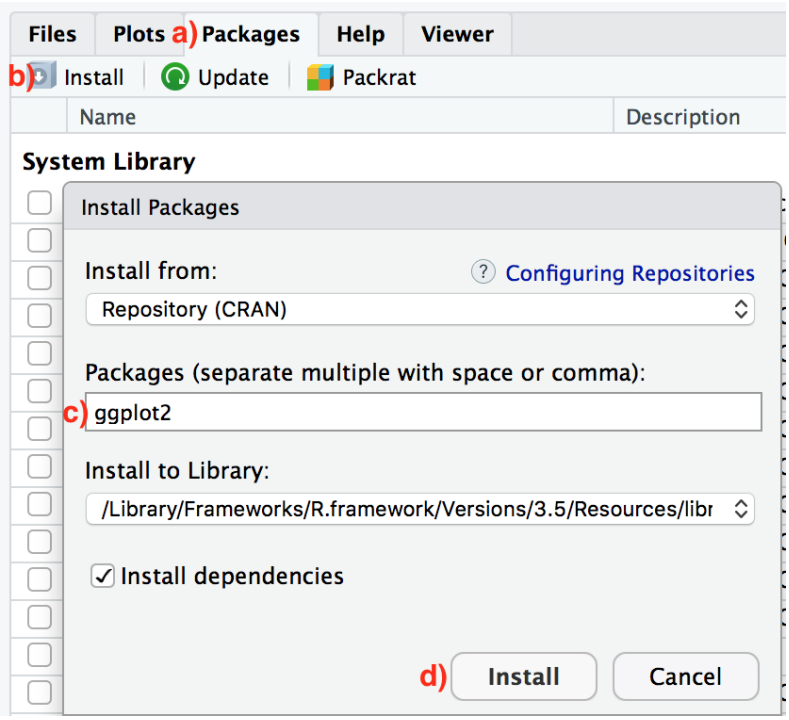

FIGURE 1.5: Installing packages in R the easy way.

An alternative but slightly less convenient way to install a package is by typing `install.packages("ggplot2")` in the console pane of RStudio and pressing Return/Enter on your keyboard. Note you must include the quotation marks around the name of the package.

Much like an app on your phone, you only have to install a package once. However, if you want to update a previously installed package to a newer version, you need to reinstall it by repeating the earlier steps.

Learning check

(LC1.1) Repeat the earlier installation steps, but for the `dplyr`, `nycflights13`, and `knitr` packages. This will install the earlier mentioned `dplyr` package for data wrangling, the `nycflights13` package containing data on all domestic flights leaving a NYC airport in 2013, and the `knitr` package for generating easy-to-read tables in R. We'll use these packages in the next section.

Note that if you'd like your output on your computer to match up exactly with the output presented throughout the book, you may want to use the exact versions of the packages that we used. You can find a full listing of these packages and their versions in Appendix E. This likely won't be relevant for novices, but we included it for reproducibility reasons.

1.3.2 Package loading

Recall that after you've installed a package, you need to "load it." In other words, you need to "open it." We do this by using the `library()` command.

For example, to load the `ggplot2` package, run the following code in the console pane. What do we mean by "run the following code"? Either type or copy-and-paste the following code into the console pane and then hit the Enter key.

```
library(ggplot2)
```

If after running the earlier code, a blinking cursor returns next to the `>` "prompt" sign, it means you were successful and the `ggplot2` package is now loaded and ready to use. If, however, you get a red "error message" that reads `...`

```
Error in library(ggplot2) : there is no package called 'ggplot2'
```

`...` it means that you didn't successfully install it. This is an example of an "error message" we discussed in Subsection 1.2.2. If you get this error message, go back to Subsection 1.3.1 on R package installation and make sure to install the `ggplot2` package before proceeding.

Learning check

(LC1.2) "Load" the `dplyr`, `nycflights13`, and `knitr` packages as well by repeating the earlier steps.

1.3.3 Package use

One very common mistake new R users make when wanting to use particular packages is they forget to "load" them first by using the `library()` command we just saw. Remember: *you have to load each package you want to use every time you start RStudio.* If you don't first "load" a package, but attempt to use one of its features, you'll see an error message similar to:

```
Error: could not find function
```

This is a different error message than the one you just saw on a package not having been installed yet. R is telling you that you are trying to use a function in a package that has not yet been "loaded." R doesn't know where to find the function you are using. Almost all new users forget to do this when starting out, and it is a little annoying to get used to doing it. However, you'll remember with practice and after some time it will become second nature for you.

1.4 Explore your first datasets

Let's put everything we've learned so far into practice and start exploring some real data! Data comes to us in a variety of formats, from pictures to text to numbers. Throughout this book, we'll focus on datasets that are saved in "spreadsheet"-type format. This is probably the most common way data are collected and saved in many fields. Remember from Subsection 1.2.1 that these "spreadsheet"-type datasets are called *data frames* in R. We'll focus on working with data saved as data frames throughout this book.

Let's first load all the packages needed for this chapter, assuming you've already installed them. Read Section 1.3 for information on how to install and load R packages if you haven't already.

```
library(nycflights13)
library(dplyr)
library(knitr)
```

At the beginning of all subsequent chapters in this book, we'll always have a list of packages that you should have installed and loaded in order to work with that chapter's R code.

1.4.1 `nycflights13` package

Many of us have flown on airplanes or know someone who has. Air travel has become an ever-present aspect of many people's lives. If you look at the Departures flight information board at an airport, you will frequently see that some flights are delayed for a variety of reasons. Are there ways that we can understand the reasons that cause flight delays?

We'd all like to arrive at our destinations on time whenever possible. (Unless you secretly love hanging out at airports. If you are one of these people, pretend for a moment that you are very much anticipating being at your final destination.) Throughout this book, we're going to analyze data related to all domestic flights departing from one of New York City's three main airports in 2013: Newark Liberty International (EWR), John F. Kennedy International (JFK), and LaGuardia Airport (LGA). We'll access this data using the `nycflights13` R package, which contains five datasets saved in five data frames:

- `flights` : Information on all 336,776 flights.

- `airlines` : A table matching airline names and their two-letter International Air Transport Association (IATA) airline codes (also known as carrier codes) for 16 airline companies. For example, “DL” is the two-letter code for Delta.
- `planes` : Information about each of the 3,322 physical aircraft used.
- `weather` : Hourly meteorological data for each of the three NYC airports. This data frame has 26,115 rows, roughly corresponding to the $365 \times 24 \times 3 = 26,280$ possible hourly measurements one can observe at three locations over the course of a year.
- `airports` : Names, codes, and locations of the 1,458 domestic destinations.

1.4.2 `flights` data frame

We’ll begin by exploring the `flights` data frame and get an idea of its structure. Run the following code in your console, either by typing it or by cutting-and-pasting it. It displays the contents of the `flights` data frame in your console. Note that depending on the size of your monitor, the output may vary slightly.

```
flights
```

```
# A tibble: 336,776 × 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# ℹ 336,766 more rows
# ℹ 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Let's unpack this output:

- `A tibble: 336,776 x 19` : A `tibble` is a specific kind of data frame in R. This particular data frame has
  - `336,776` rows corresponding to different *observations*. Here, each observation is a flight.
  - `19` columns corresponding to 19 *variables* describing each observation.
- `year` , `month` , `day` , `dep_time` , `sched_dep_time` , `dep_delay` , and `arr_time` are the different columns, in other words, the different variables of this dataset.
- We then have a preview of the first 10 rows of observations corresponding to the first 10 flights. R is only showing the first 10 rows, because if it showed all `336,776` rows, it would overwhelm your screen.
- `...` `with 336,766 more rows, and 11 more variables:` indicating to us that 336,766 more rows of data and 11 more variables could not fit in this screen.

Unfortunately, this output does not allow us to explore the data very well, but it does give a nice preview. Let's look at some different ways to explore data frames.

1.4.3 Exploring data frames

There are many ways to get a feel for the data contained in a data frame such as `flights`. We present three functions that take as their "argument" (their input) the data frame in question. We also include a fourth method for exploring one particular column of a data frame:

1. Using the `View()` function, which brings up RStudio's built-in data viewer.
2. Using the `glimpse()` function, which is included in the `dplyr` package.
3. Using the `kable()` function, which is included in the `knitr` package.
4. Using the `$` "extraction operator," which is used to view a single variable/column in a data frame.

1. `View()`:

Run `View(flights)` in your console in RStudio, either by typing it or cutting-and-pasting it into the console pane. Explore this data frame in the resulting pop up viewer. You should get into the habit of viewing any data frames you encounter. Note the uppercase `V` in `View()`. R is case-sensitive, so you'll get an error message if you run `view(flights)` instead of `View(flights)`.

Learning check

(LC1.3) What does any *ONE* row in this `flights` dataset refer to?

- A. Data on an airline
- B. Data on a flight
- C. Data on an airport
- D. Data on multiple flights

By running `View(flights)`, we can explore the different *variables* listed in the columns. Observe that there are many different types of variables. Some of the variables like `distance`, `day`, and `arr_delay` are what we will call *quantitative* variables. These variables are numerical in nature. Other variables here are *categorical*.

Note that if you look in the leftmost column of the `View(flights)` output, you will see a column of numbers. These are the row numbers of the dataset. If you glance across a row with the same number, say row 5, you can get an idea of what each row is representing. This will

allow you to identify what object is being described in a given row by taking note of the values of the columns in that specific row. This is often called the *observational unit*. The observational unit in this example is an individual flight departing from New York City in 2013. You can identify the observational unit by determining what "thing" is being measured or described by each of the variables. We'll talk more about observational units in Subsection 1.4.4 on *identification* and *measurement* variables.

2. `glimpse()` :

The second way we'll cover to explore a data frame is using the `glimpse()` function included in the `dplyr` package. Thus, you can only use the `glimpse()` function after you've loaded the `dplyr` package by running `library(dplyr)`. This function provides us with an alternative perspective for exploring a data frame than the `View()` function:

```
glimpse(flights)
```

```
Rows: 336,776
Columns: 19
$ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2…
$ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1…
$ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1…
$ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, …
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, …
$ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -1…
$ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,…
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,…
$ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -1…
$ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "…
$ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4…
$ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394…
$ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",…
$ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",…
$ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1…
$ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, …
$ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6…
$ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0…
$ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 0…
```

Observe that `glimpse()` will give you the first few entries of each variable in a row after the variable name. In addition, the *data type* (see Subsection 1.2.1) of the variable is given immediately after each variable's name inside `< >`. Here, `int` and `dbl` refer to "integer" and "double", which are computer coding terminology for quantitative/numerical variables. "Doubles" take up twice the size to store on a computer compared to integers.

In contrast, `chr` refers to "character", which is computer terminology for text data. In most forms, text data, such as the `carrier` or `origin` of a flight, are categorical variables. The `time_hour` variable is another data type: `dttm`. These types of variables represent date and time combinations. However, we won't work with dates and times in this book; we leave this topic for other data science books like *Introduction to Data Science* by Tiffany-Anne Timbers, Melissa Lee, and Trevor Campbell or *R for Data Science* (Grolemund and Wickham 2017).

Learning check

(LC1.4) What are some other examples in this dataset of *categorical* variables? What makes them different than *quantitative* variables?

3. `kable()` :

The final way to explore the entirety of a data frame is using the `kable()` function from the `knitr` package. Let's explore the different carrier codes for all the airlines in our dataset two ways. Run both of these lines of code in the console:

```
airlines
kable(airlines)
```

At first glance, it may not appear that there is much difference in the outputs. However, when using tools for producing reproducible reports such as R Markdown, the latter code produces output that is much more legible and reader-friendly. You'll see us use this reader-friendly style in many places in the book when we want to print a data frame as a nice table.

4. `$` operator

Lastly, the `$` operator allows us to extract and then explore a single variable within a data frame. For example, run the following in your console

```
airlines$name
```

We used the `$` operator to extract only the `name` variable and return it as a vector of length 16. We'll only be occasionally exploring data frames using the `$` operator, instead favoring the `View()` and `glimpse()` functions.

1.4.4 Identification and measurement variables

There is a subtle difference between the kinds of variables that you will encounter in data frames. There are *identification variables* and *measurement variables*. For example, let's explore the `airports` data frame by showing the output of `glimpse(airports)`:

```
glimpse(airports)
```

```
Rows: 1,458
Columns: 8
$ faa   <chr> "04G", "06A", "06C", "06N", "09J", "0A9", "0G6", "0G7", "0P2", "…
$ name  <chr> "Lansdowne Airport", "Moton Field Municipal Airport", "Schaumbur…
$ lat   <dbl> 41.1, 32.5, 42.0, 41.4, 31.1, 36.4, 41.5, 42.9, 39.8, 48.1, 39.6…
$ lon   <dbl> -80.6, -85.7, -88.1, -74.4, -81.4, -82.2, -84.5, -76.8, -76.6, -…
$ alt   <dbl> 1044, 264, 801, 523, 11, 1593, 730, 492, 1000, 108, 409, 875, 10…
$ tz    <dbl> -5, -6, -6, -5, -5, -5, -5, -5, -5, -8, -5, -6, -5, -5, -5, -5, …
$ dst   <chr> "A", "A", "A", "A", "A", "A", "A", "A", "U", "A", "A", "U", "A",…
$ tzone <chr> "America/New_York", "America/Chicago", "America/Chicago", "Ameri…
```

The variables `faa` and `name` are what we will call *identification variables*, variables that uniquely identify each observational unit. In this case, the identification variables uniquely identify airports. Such variables are mainly used in practice to uniquely identify each row in a data frame. `faa` gives the unique code provided by the FAA for that airport, while the `name` variable gives the longer official name of the airport. The remaining variables (`lat`, `lon`, `alt`, `tz`, `dst`, `tzone`) are often called *measurement* or *characteristic* variables: variables that describe properties of each observational unit. For example, `lat` and `long` describe the latitude and longitude of each airport.

Furthermore, sometimes a single variable might not be enough to uniquely identify each observational unit: combinations of variables might be needed. While it is not an absolute rule, for organizational purposes it is considered good practice to have your identification variables in the leftmost columns of your data frame.

Learning check

(LC1.5) What properties of each airport do the variables `lat`, `lon`, `alt`, `tz`, `dst`, and `tzone` describe in the `airports` data frame? Take your best guess.

(LC1.6) Provide the names of variables in a data frame with at least three variables where one of them is an identification variable and the other two are not. Further, create your own tidy data frame that matches these conditions.

1.4.5 Help files

Another nice feature of R are help files, which provide documentation for various functions and datasets. You can bring up help files by adding a `?` before the name of a function or data frame and then run this in the console. You will then be presented with a page showing the corresponding documentation if it exists. For example, let's look at the help file for the `flights` data frame.

```
?flights
```

The help file should pop up in the Help pane of RStudio. If you have questions about a function or data frame included in an R package, you should get in the habit of consulting the help file right away.

Learning check

(LC1.7) Look at the help file for the `airports` data frame. Revise your earlier guesses about what the variables `lat`, `lon`, `alt`, `tz`, `dst`, and `tzone` each describe.

1.5 Conclusion

We've given you what we feel is a minimally viable set of tools to explore data in R. Does this chapter contain everything you need to know? Absolutely not. To try to include everything in this chapter would make the chapter so large it wouldn't be useful! As we said earlier, the best way to add to your toolbox is to get into RStudio and run and write code as much as possible.

1.5.1 Additional resources

If you are new to the world of coding, R, and RStudio and feel you could benefit from a more detailed introduction, we suggest you check out the short book, *Getting Used to R, RStudio, and R Markdown* (Ismay and Kennedy 2016). It includes screencast recordings that you can follow along and pause as you learn. This book also contains an introduction to R Markdown, a tool used for reproducible research in R.

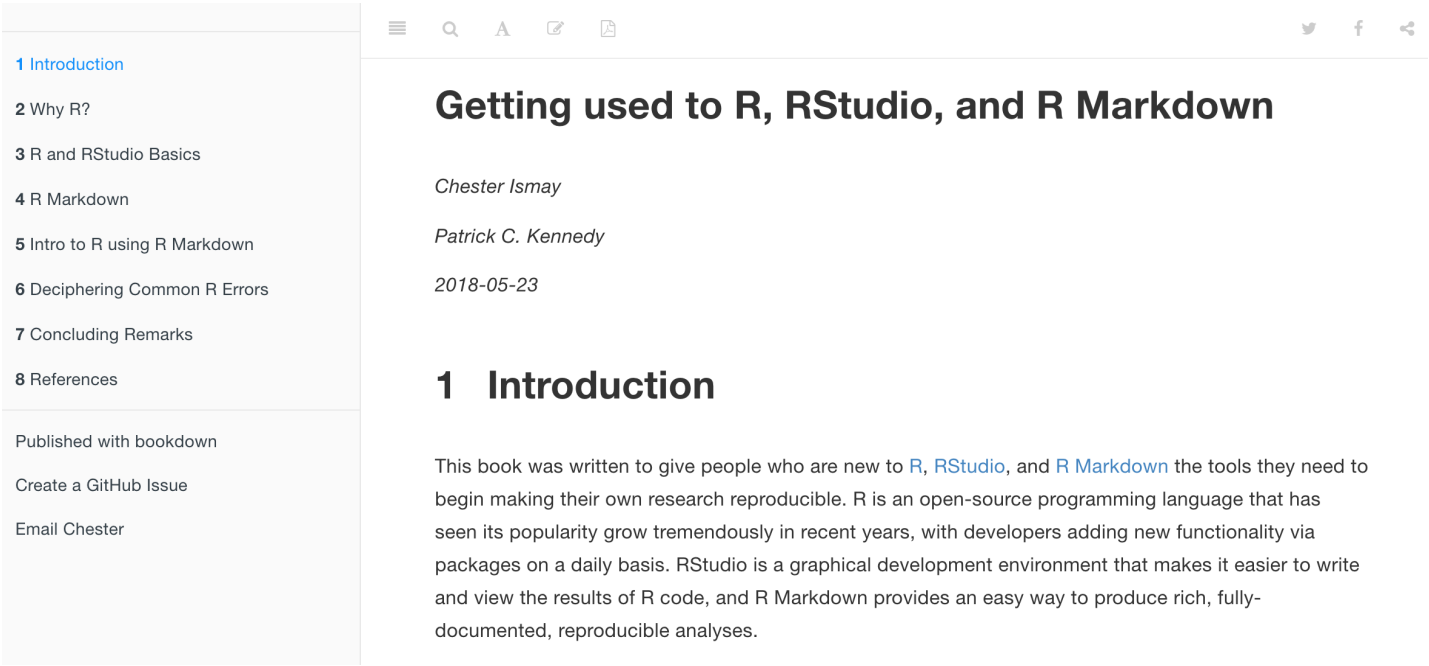

FIGURE 1.6: Preview of *Getting Used to R, RStudio, and R Markdown*.

1.5.2 What's to come?

We're now going to start the "Data Science with `tidyverse`" portion of this book in Chapter 2 as shown in Figure 1.7 with what we feel is the most important tool in a data scientist's toolbox: data visualization. We'll continue to explore the data included in the `moderndive` and

`nycflights13` packages using the `ggplot2` package for data visualization. You'll see that data visualization is a powerful tool to add to your toolbox for data exploration that provides additional insight to what the `View()` and `glimpse()` functions can provide.

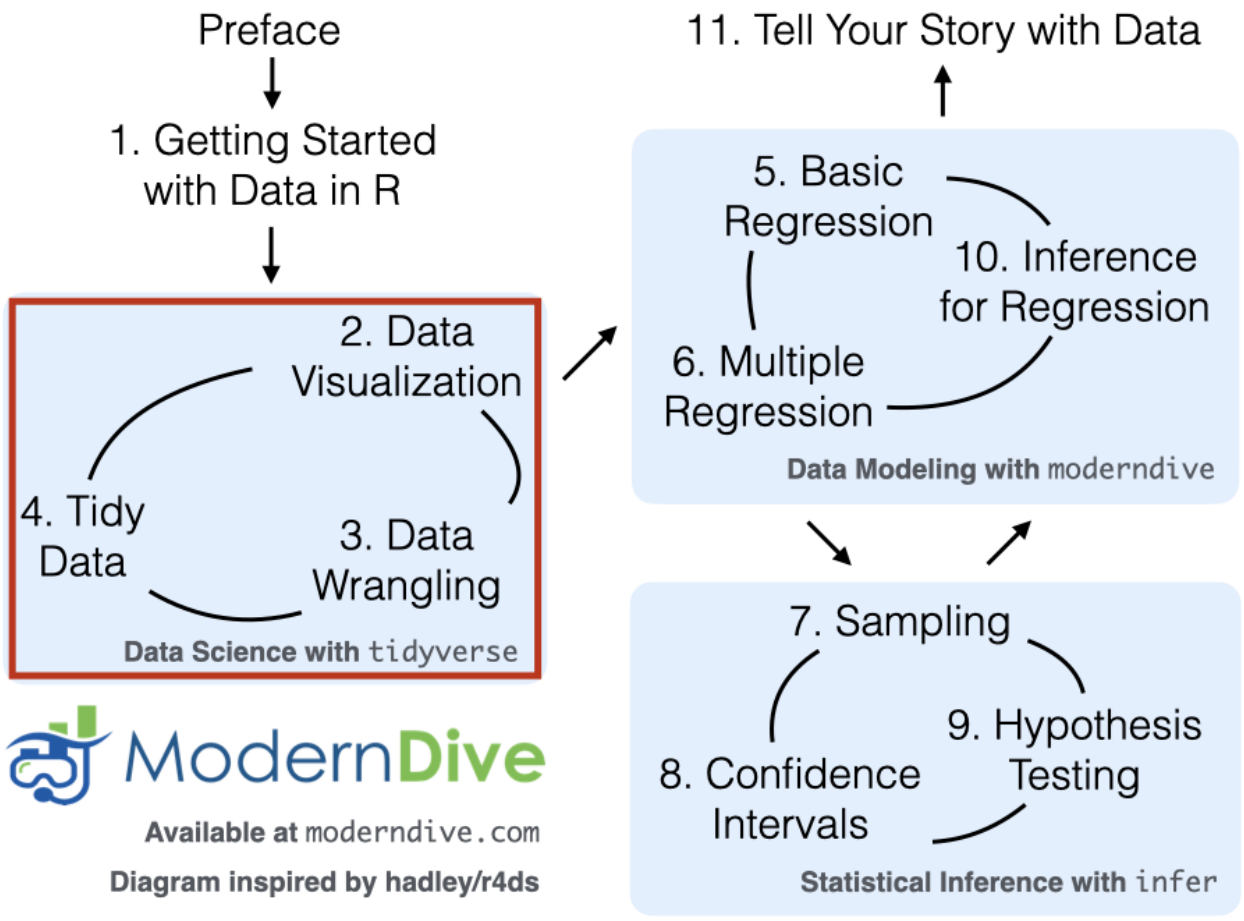

FIGURE 1.7: *ModernDive* flowchart - on to Part I!

References

Bray, Andrew, Chester Ismay, Evgeni Chasnovski, Simon Couch, Ben Baumer, and Mine Cetinkaya-Rundel. 2022. *Infer: Tidy Statistical Inference*.

Grolemund, Garrett, and Hadley Wickham. 2017. *R for Data Science*. First. Sebastopol, CA: O'Reilly Media. https://r4ds.had.co.nz/.

Ismay, Chester, and Patrick C. Kennedy. 2016. *Getting Used to r, RStudio, and r Markdown*. https://rbasics.netlify.com.

Kim, Albert Y., and Chester Ismay. 2022. *Moderndive: Tidyverse-Friendly Introductory Linear Regression*.

Wickham, Hadley, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, and Dewey Dunnington. 2023. *Ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*.

Wickham, Hadley, Romain François, Lionel Henry, Kirill Müller, and Davis Vaughan. 2023. *Dplyr: A Grammar of Data Manipulation*.