Chapter 1

A Gentle Introduction to Programming

1.1 Data are everywhere — and they come in all types.

Data are everywhere we look. They are not necessarily values in a spreadsheet that lack context, but embedded in our everyday discussions and communication. How the stock market is doing is rarely communicated to the public in the form of a spreadsheet (perhaps to investors). Rather it is communicated in the form of concise statements. Let's take the example for a CNBC online article about stock market performance: "The Dow Jones industrial average erased earlier losses today to end 56.97 points higher at 21,865.37...".

- Each piece of information corresponds to a different data type (e.g., numeric, string, factors, Boolean, dates).
- "Dow Jones industrial average" references the name of a specific stock-market index.
- "today" references a date (as does "earlier")
- "56.97" and "21,865.37" are numbers (with units)
- There are also data embedded in the phrases "erased", "earlier losses" and "higher".

An investor or an economist can take an isolated statement and draw upon all past information in their memory to make sense of it. Computers, on the other hand, need a little help. They need people to structure data and set guidelines for them to follow. For one thing, each piece of information needs to be associated with a *data type* such as numeric and character. This dictates how each value can be treated. Each morsel of data can be stored as an object. Similar objects can be organized together into data structures that can be more easily analyzed.

Once information is structured and code is written, computers are capable of incredible feats from examining satellite imagery to making sense of text. This chapter focuses on the fundamentals of statistical programming. We cover the basics of how data are stored, what can be done to each type of information, loading and executing code among others. These basic concepts are everything on which data science relies.

1.2 Data types

In R, data are encoded (often implicitly) as one of several types of data e.g., `numeric`, `logical`, `character`, `factor`, or `date`. In this section, we describe the properties of each.

1.2.1 numeric

As you may have guessed, *numeric* data are numbers. The numeric type is R's catch-all for all numbers, encompassing both integer and continuous numbers.

```
4
56.4
4/9
pi
```

1.2.2 character

Words, sentences, letters, among other sequence of letters and numbers are often encoded as `character` data (also strings as *strings*) in datasets of linguistic text (e.g., court transcripts, novels, text messages, or government audit reports). When a value is wrapped in quotation marks, (e.g., `"Text"`), we are indicating to R that the value is a character. Both single quotes (`'`) and double quotes (`"`) work, you just. For example:

```
"text"
"Text is fun!"
"Let's have an example with an interior quotation mark."
```

`logical`. If we want to test whether something is *true* or *false*, then we have stumbled into the world of logic. Because programming and data science use logical data so frequently, they get their own data type. In R, logical data are `TRUE` or `FALSE`, which can be abbreviated as `T` or `F`. As is always the case in R: *the letters' cases matter*, meaning you must use `TRUE` and cannot use `true`. Below are the only four values that logical values can assume:

```
TRUE
T
FALSE
F
```

But how do we produce logical values? Often times, a relational operator is used to test whether a statement is `TRUE` or `FALSE`. For instance:

```
# Is 3 less than 4?
3 < 4

# Is 3 greater than 4?
3 > 4

# Is 3 equal to 4?
3 == 4
```

In combination with a logical operator, we can evaluate two or more relational statements.

```
# is 3 less than 4 AND greater than 1?
(3 < 4) & (3 > 1)

# is 3 less than 4 OR greater than 1?
(3 < 4) | (3 > 1)
```

There are a variety of relational operators like the ones above and logical operators that will come in handy

for filtering and subsetting data in later chapters. The table below presents some of R's most common logical operators.

Table 1.1: Common relational and logical operators.

Operator	Example	Translation
`>`	`x > y`	`x` is greater than `y`.
`<`	`x < y`	`x` is less than `y`.
`>=`	`x >= y`	`x` is greater than or equal to `y`.
`<=`	`x <= y`	`x` is less than or equal to `y`.
`==`	`x == y`	`x` is equal to `y`.
`&`	`x > y & x > z`	Both `x > y` and `x > z` are true.
`|`	`x > y | x > z`	Either `x > y` or `x > z` are true.

These logical operators are particularly common for `numeric` data, but they be used them for any type of data. For instance, if we were to execute each of these statements, we will notice that R is indeed case sensitive: `"a"` is not the same as `"A"`, and, in fact, `"a"` precedes `"A"` in R's ordering of letters. Numbers precede letters.

```
# Is "A" equal to "a"? (Is R case sensitive?)
"A" == "a"

# Is "a" less than "A"?
"a" < "A"

# Is 3 less than "a"?
3 < "a"

# Is numeric 3 equal to character 3?
3 == "3"
```

Finally, it is worth noting that you can transform `TRUE` and `FALSE` to binary numeric (`1` and `0`) by multiplying them by the number `1`. This is particularly useful for creating *dummy variables* quickly.

```
TRUE * 1
1 * FALSE
```

1.2.3 `factor`

Factors are a special data type in R, because they allow you to take one type of variable (e.g., `numeric` or `character`) and add a few more layers of information. We can add labels to the data so that values "Sun" and "Mon" are labeled "Sunday" and "Monday". Also, we have the option to indicate the order of that data, meaning Sunday precedes Monday, etc. You can manipulate these labels, the ordering, and a number of other factor options through the functions `factor` and `as.factor`. We will more fully explore `factor` data in the graphing section.

1.2.4 `date`

While there are many ways to think about dates – `character` data like "January" or `numeric` data like `1` – dates are so common (and oddly challenging to work with) that they get their own type of data, namely, the `date` type. We will dive more into working with dates in the next chapter.

1.2.5 The `class` function

Inferring the type of data by simplying looking at the data can be misleading. The question of type is particularly important when you read data into R from another source – we will not always know whether R read a variable as `numeric`, `character`, `logical`, or perhaps `factor`. The `class` function works wonders.

A Quick aside on functions: This is the first time in this book that we officially use function. We will cover functions in much greater depth throughout the book, but let's cover a few of the basics now. Functions take **arguments** (the data that you *feed* the function) and produce **output**. For example, the class function takes one argument like the number `56.97` and outputs a character that describes the class of whatever argument we inputted into the function – `"numeric"` in our case. Functions help us complete tasks/analyses that may be too difficult or time-consuming to code by ourselves. Functions also excel at repetitive tasks – you can easily tell R to do something a million times so you don't have to copy and paste a million times. You can also write your own functions in R—but that will come later.

Aside complete. Let's see this `class` function in action.

```
# numeric
class(56.97)
class(57)
class(57L)
class(1/3)

# character
class("Dow Jones")

# logical
class(TRUE)
class(F)
class(3 > 2)
class(1 * TRUE)

# factor
class(factor(3))

# date
Sys.Date()
class(Sys.Date())
as.Date("2015-12-31")
class(as.Date("2015-12-31"))

# Fun
class(class)
```

One more note on functions: Many functions in R take many arguments, and sometimes arguments are optional. To keep track of which values you are assigning to which arguments, it is often best to use the

argument's name. In the `class` examples that we just covered, we implicitly assigned the various values that we fed to `class` to its sole argument `x`. We could have also used the argument's name.

```
# Without the argument's name
class(TRUE)

# With the argument's name
class(x = TRUE)
```

Both options produce the same output, but the second example – in which we explicitly use the argument's name – makes it a bit clearer what we are doing. We admit it does not truly matter for `class`, since there is only the possibility of one argument. However, we will soon meet some functions where explicitly naming arguments helps you and your collaborators.

1.3 Objects in R

Simply entering data into R's console is not enough. We need a way to store data in our computer's memory. To be able to access and re-access the data that we store, we give each piece of data a name. In R parlance, we *assign* the values (the data) to objects stored in memory (the names).

R's assignment operator `<-` exists solely for the task: assigning a value (on the right-hand side) to the name (on the left-hand side). If you are reading code out load,[1] `object <- value` is typically read "*object* gets *value.*""

```
a <- 3
string_ex <- "Some funny/clever words."
some.logic <- TRUE
```

As you can see, we have a freedom in the names that can be used for the objects. However, we can mitigate the heartache of troubleshooting erroneous code by using common variable name conventions. Variable names should avoid (1) a number as the first character in the name (e.g. `"1variable"` is not a good idea) as this will not register as an object, (2) any spaces in the name (e.g. `"variable 1"`), (3) a name of an existing function.

Once we assign a value to an object, we can access that object's value – and perform operations on that value. For instance, we can assign a value of three to object `a`, as above, and then we can (1) check the value held in `a` by executing `a` in the console, (2) check the class of `a`, and (3) multiply `a` by 3:

```
# Assign value 3 to object named 'a'
a <- 3

# (1) Check a's value
a

# (2) Check a's class
class(a)

# (3) Multiply a by 3
a * 3
```

[1] Yes, it does happen.

Looks great!

1.4 R's object classes

Just as data in R come in different *types*, objects in R come in different *classes*, e.g., a `vector`, a `matrix`, a `data.frame`, or a `list`. Each of the object classes presents data in a slightly different structure, making it better for some uses and worse for others. We'll review four of the most common data structures now and then will discuss others as they arise throughout the book.

1.4.1 `vector`

`vector` objects are one of the simplest and most used classes of objects in R. Specifically, (in R) a vector is a one-dimensional "list" of values. The values in a `vector` can be `logical`, `numeric`, `factor`, or `character`, but they must all be of the same type. If the objects are not all of the same type, R will *coerce* them to the most conducive type.

The simplest way to create a vector is with the concatenate function: `c`. To create a `vector` of values, you feed the `c` function your values, each separated by a comma (`,`). For instance, to make a vector out of the numbers 1, 2, and 3, we write `c(1, 2, 3)`. And to create a vector of the characters `"apple"` and `"banana"`, we write `c("apple", "banana")`.

```
# Vector of numbers
c(1, 2, 3)

# Vector of characters
c("apple", "banana")
```

So what happens when we create a vector of a number and a character?

```
# Mixed vector
c(1, "banana")
```

R coerces everything to character. This result may not be what you expected—you may have expected R to allow a vector to contain objects of different types.[2] At some point, the folks who developed R made this decision—and millions of other little decisions—that led to the current implementation of R. As you get deeper into R, you will find that there is a logic to these decisions (e.g., why R tells you it is `TRUE` that `"apple pie" < "banana cream pie"`), but R will still occasionally surprise you. And that is part of learning to program in R. So don't be afraid to dive into the rabbit holes that surprise you. You may lose a few hours in the short run, but in the long run you'll become an excellent data scientist.[3]

1.4.2 `matrix`

The next class of objects we'll discuss is R's `matrix` class. In R, matrices are very similar to the matrices you met in your mathematics courses: they have rows and columns filled with data. However, in addition to filling your matrices with numbers, as you did in your math courses, in R You can also fill your `matrix` with other types of data—e.g., logical, character, factor (just as you can with vectors).

[2] Wait a few sections for R's `list`.
[3] And R Wizard.

You generally will need to feed two arguments to R's `matrix` function:

1. The `data` that you want placed into your matrix, as a `vector`
2. The number of rows (`nrow`) in your matrix

For example, let's imagine we want a matrix with 2 rows and 3 columns that contains the integers one through six. Then we need to feed `matrix` a (1) a vector containing the integers one through six, and (2) `nrow = 2`, telling R we want two rows.

Aside on creating numeric sequences: As we covered before, you can easily create short numeric vectors using the `c` function, e.g., `c(1, 2, 3)`. But this construction quickly gets tedious for longer vectors—even creating a vector of the integers between 1 and 100 would be a nightmare with only the `c` function. Lucky for us, R's makers included a quick and easy fix: `a:b` generates a numeric vector containing the integers from `a` to `b` (inclusive of `a` and `b`). E.g., `1:100` will create a vector with the integers 1 to 100.[4]

```
# Old school
c(1, 2, 3, 4, 5, 6)

# New school
1:6
```

Using our new sequence knowledge, we can create our 2×3 matrix[5] with the integers one through six.

```
matrix(data = 1:6, nrow = 2)
```

And there we have it. Notice that because the `matrix` function takes two arguments, we explicitly named the arguments[6] and we separated the two arguments' assignments with a comma. Also notice that by default, R filled the matrix *by its columns* (top to bottom and then left to right). If you would rather fill your matrix *by its rows*, you can add the optional argument `byrow = TRUE` to the `matrix` function:

```
matrix(data = 1:6, nrow = 2, byrow = TRUE)
```

As we mentioned earlier: you can create matrices filled with essentially any type of data.

```
# Character
matrix(data = c("Six", "words", "can", "be", "a", "lot."), ncol = 3, byrow = T)

# Logical
matrix(data = c(T, T, F, F, F, T), ncol = 3)
```

Finally, notice that you can force R to fill in a `matrix` by specifying both the number of rows (`nrow`) **and** the number of columns (`ncol`) for a `matrix`.

```
matrix(data = 1, nrow = 3, ncol = 2)
```

[4] For more complicated tasks, there's also the sequence function `seq`, which takes the starting value for the sequence (`from`), the stopping value for the sequence (`to`), and the *distance* between each of the items in the sequence (`by`).

[5] Two rows and three columns

[6] Not necessary but highly helpful.

1.4.3 `data.frame`

As a budding data scientist, one day you will wake up and find yourself surrounded by `data.frame`s. The `data.frame` is the basic object for data entry and analysis in R—much like a table of data or a spreadsheet (but better). Just like a `matrix`, a `data.frame` has rows and columns. Unlike a `matrix`, a `data.frame` can have different types of data in each of its columns. In addition, you will name each of the columns in your `data.frame`.

In general, you define a `data.frame` using the following conventions:

```
data.frame(
  column1 = values...,
  column2 = values...,
  ...
  columnN = values
)
```

More concretely, let's create a two-column `data.frame`. The first column will be called `some_letters` and will contain... some letters. The second column with be called `some_numbers`...

```
data.frame(
  some_letters = c("a", "A", "b", "B"),
  some_numbers = 1:4
)
```

One of the main requirements for a `data.frame` is that R needs to be able to fill each of the `data.frame`'s columns. In practice this requirement means one of two things:

1. The values you specify for each column have the same number of elements (as they do above).
2. The numbers of elements you specify for each column are multiples of each other.

For an example of the second option:

```
data.frame(
  some_letters = c("a", "A", "b", "B"),
  some_numbers = 1:2
)
```

R repeats the shorter vector (the second column in this example) until the two columns are the same length.

If the numbers of elements are not multiples (e.g., 4 and 3), R will refuse to construct the `data.frame`, and you will get an error. E.g.,

```
data.frame(
  some_letters = c("a", "A", "b", "B"),
  some_numbers = 1:3
)
```

```
## Error in data.frame(some_letters = c("a", "A", "b", "B"), some_numbers = 1:3): arguments imply diffe
```

Notice that R's error message gives us a hint to what the problem is: `arguments imply differing number of rows: 4, 3`. Generally, error messages in R will contain a hint of the problem, but you have not know

a bit about R to understand the hints (e.g., here we need to know the rules about column lengths in a `data.frame`). As a result, error messages can seem a bit cryptic while you are in the early stages of learning R.

Aside: In the `data.frame` examples above, we wrote `data.frame(` and then went to the next line to define the columns. R does not stop at the end of the line, because it 'knows' that the `data.frame` definition is unfinished. You don't have to define a `data.frame` over multiple lines[7], but using multiple lines tends to make your code more legible and easier to troubleshoot.

1.4.4 `list`

R's `list` class of objects is exactly what its name implies—it lists things. Similar to the `vector`, a `list` has only one dimension—its length. Unlike the `vector`, a `list` can contain objects of many different types. The crazy part of `list` its elements can be *anything*. You are not stuck with simple objects like `1` or `T` or `"Some text"`. A single `list` can have `vector`s, `data.frame`s, and even other `list`s as its elements.

The conventions for creating a `list` mirror what we've discussed so far—especially with the `vector`-creating `c` function: You simply pass the objects you want in the `list` as the arguments to the `list` function:

```
list(
  "apples",
  c(T, F, T),
  matrix(data = 1:4, nrow = 2),
  data.frame(var1 = 9:10, var2 = 11:12)
)
```

Here we have a (very mixed) four-element `list`, whose individual elements are

1. a one-element `character` vector,
2. a three-element `logical` vector,
3. a 2×2 `matrix`, and
4. a two-variable `data.frame`

As with most objects in R—and just as we did with the columns of the `data.frame`—you can name the individual elements of the `list` when you create the `list`. Let's re-create the `list`, naming its elements with super-creative names.

```
list(
  el_1 = "apples",
  el_2 = c(T, F, T),
  el_3 = matrix(data = 1:4, nrow = 2),
  el_4 = data.frame(var1 = 9:10, var2 = 11:12)
)
```

Our knowledge and creativity are impressive, right?

1.4.5 The `class` function, v2

Previously we used the `class` function to learn the data types of various objects (e.g., `class(3.1415)`). You can also use `class` to learn about an object's class. For example,

[7] You can do it in one line: `data.frame(some_letters = c("a", "A", "b", "B"), some_numbers = 1:3)`

```
# Check the class of a matrix
class(matrix(data = 1:4, nrow = 2))
```

```
## [1] "matrix"
```

```
# Check the class of a data.frame
class(data.frame(a = 1:3, b = 1:6))
```

```
## [1] "data.frame"
```

```
# Check the class of a list
class(list(a = 1:3, b = 1:6))
```

```
## [1] "list"
```

However, if you check the class of a `vector`, R will not tell you `"vector"`. Rather, R will tell you what type of vector you have—`numeric`, `logical`, `character`, etc.

```
# Numeric vector class
class(c(1, 2, 3))
```

```
## [1] "numeric"
```

```
# Character vector class
class(c("a", "b", "c"))
```

```
## [1] "character"
```

```
# Logical vector class
class(c(T, F, T))
```

```
## [1] "logical"
```

```
# Coercion to character
class(c(T, 3, "Hi"))
```

```
## [1] "character"
```

The takeaway? R 'thinks' about different types of vectors as different classes of objects.

In addition, each object class also has its own function to check whether an object belongs to the class. For instance, you can use the `is.matrix` to check whether an object is a `matrix`. Type `is.` into the RStudio console and then hit your tab key—you will get a glimpse of all of these `is.*` class functions.

1.4.6 More classes

R offers many other object classes. Many of the object classes are specific to an application (e.g., geographic data like rasters or shape files) or packages (e.g., when you run a regression with the function `lm`, the function produces a special `lm` object).

1.5 Packages

1.5.1 Base R and filling the need for more functions

When you boot up R, you automatically have access to a large set of functions. You can perform mathematical operations (addition `+`, division `/`), conduct statistical summaries (`mean`, `sd`), run a regression (e.g. `lm`), plot a histogram (`hist`) and even make a map (`plot`). This original set of functions that shipped with R is called *base R* – the basic setup when you open the box. Eventually, you will find yourself in a situation in which you need to conduct more complex and specialized tasks that are not possible by any function in base R.

Some may have the adventurous itch to program specialized functions, which may sound daunting. But fear not, there is a whole universe of functions awaiting your use on the Comprehensive R Archive Network (CRAN) as well as on Github. Specifically, functions are available as families of functions, formally called *packages*.

For example, imagine some crazy, hypothetical world (such as the one in which we live) where someone sends you an Excel spreadsheet that you then need to load into R. Herein lies the problem: *base R does not have a function to read directly from a spreadsheet.* But, as its name implies the `readxl` package on CRAN provides exactly the functionality you need.

1.5.2 Installing packages

A large community of R users and developers contribute and maintain a (vast archive of packages)[https://cran.r-project.org/web/packages/available_packages_by_name.html]. And these package span computationally intensive tasks like reading satellite imagery to more visual tasks like creating interactive maps for websites. It is easy to tap into these vast resources, and to get started with a package like `readxl`, we only need two inputs: (A) an internet connection and (B) R's `install.packages` function.

To install `readxl`, you can install the `readxl` package by executing the following line in the R console. To break it down, `install.packages` requires the name of a known package on CRAN in the form of a string value.

```
install.packages("readxl")
```

For some packages, R will also install package dependencies – other packages on which the request package relies. This is one of the beautiful things about R: it is an extensible language that builds on existing functionality. Entire families of packages are dependent on common functions, ensuring that a common logic is employed in making the magic happen with data.

There are sources (e.g., METACRAN) that allow you to search CRAN for packages, but, as with many R-related tasks, Google and asking friends and colleagues are going to be your best bets.

1.5.3 Loading packages

Like installing a new air conditioning unit, the unit is only useful when it is plugged in and turned on. R packages follow the same logic. Installing a package simply means that you now have access to the package on your machine. To take advantage of the functionality, packages need to be loaded into R environment during each session.

To load a package, you use the `library` function in combination with the package's name. Note that whereas `install.packages` expects the package name in string format, the `library` function can accommodate both strings (e.g. "readxl") or the package as a name (e.g. readxl) as it is now an installed resource.

```
library(readxl)
```

The `library` function will only load one package at a time, meaning that every package will need to be called individually. If we needed to load two installed packages – `readxl` and a package for text manipulation known as `tidytext` – we would need to do the following:

```
library(readxl)
library(tidytext)
```

For highly complex projects requiring many packages, this will be cumbersome. We will have a fix for this shortcoming later on.

When a package is loaded, it is only temporarily help in the computer's memory. Whenever R and RStudio are closed or restarted, all loaded packages are removed from memory. The point: you will need to load the packages again, but you do not need to install them again.

1.5.4 Package management and `pacman`

While installing, loading, and updating packages in R is fairly quick and easy, the `pacman` further streamlines and simplifies these process (and many related processes). What if we needed to load these three packages? The manual option requires one to embrace the tedium:

```
library(readxl)
library(tidytext)
library(ggplot2)
...
```

And the tedium does not end on your computer. When the code is shared with a collaborator, they may not have all of the packages installed, meaning the script will *crash*, requiring them to backtrack through the error log and install all of the missing packages. This scenario happens all too often, but it need not be the case.

Meet: `pacman`.

Upon installing `pacman`, the mental overhead of keeping track of all packages is quickly minimized:

```
library(pacman)
p_load(readxl, tidytext, ggplot2)
```

The `p_load` function within the `pacman` package makes package management easy. Not only does `p_load` install requested but missing packages, it loads them. Now, including this line of code at the top of each R script only requires collaborators to have `pacman` installed and the package does all of the heavy lifting of keeping packages up-to-date and in sync.

Quick Exercise

There are [X] packages that greatly expand the capabilities of R. And these packages will re-appear again and again throughout this book, namely:

Table 1.2: Recommended packages.

Package	Description
readxl	
readr	
haven	
XML	
rjson	
tidyr	
reshape2	
tidytext	
data.table	
lubridate	
dplyr	
ggplot2	
caret	

Write a command to import these core R packages.

1.6 Data I/O

When Captain Jean Luc Picard says "Make it so, Number Two", he is giving the order to execute a command that he specified prior to his statement. All the necessary information needed by the starship's crew to act is embedded in that order, ensuring that a precise output is delivered according to Picard's input. Our interaction with computers is no different. Computer programs cannot guess what we intend to do – although not yet. Computers are designed to interact and respond to humans given specific rules and protocols, and importing data is no different. This is the fundamental idea that underlies data *Input/Output*.

Suppose you need to analyze an Excel file of stock market data. In order to load the data, we need three pieces of information:

- where to look for a file (*directory*)
- how to load the file (*function*), and
- the name of file (*file*).

1.6.1 Directories

R can import any kind of file, but it first needs to know where to look. Directory paths are a special kind of string that dictate how R should traverse folders to get to a folder. Suppose if the Marvel superhero Black Panther wanted to take a look at the fuel bill for his jet vehicle that is stored in a directory on an Apple computer. The path may look something like `/Users/BlackPanther/Tech_and_Travel/Bills/`. On a Windows machine, the path may look like `C:\BlackPanther\Tech_and_Travel\Bills\`.

Rules and Tips with Paths. There are basic rules and best practices that ensure that folder management is smooth. In R, each subfolder in a directory paths should be denoted with a forward slash. If one has a particular affinity for backslashes, then know that only Windows computers support backslashes, but there must be two backslashes for every slash. While it might not matter in R, it is generally good programming practice to avoid adding spaces to paths, like `Black Panther`. Spaces should be replaced with spaces with underscores (`_`).

Name paths that are rememberable

- *Erroneous path*: `\Users\BlackPanther\Tech and Travel\Bills\`

- *Acceptable path on Mac but could be better*: `/Users/Black Panther/Tech and Travel/Bills/`
- *Acceptable path on Mac and Windows Machines*: `/Users/Black_Panther/Tech_and_Travel/Bills/`
- *Acceptable path in Windows*: `C:\\Users\\BlackPanther\\Tech_and_Travel\\Bills\\`

Note that the erroneous path that begins with `\Users` conflates the combination `\U` with special characters rather than as strings representing paths.

To put the paths in - RStudio helps you complete (tab) - For example,

- Navigating directories
- `getwd`, `setwd`, and `dir`

Best practice: Define your directories to objects at the start of your script.

```
#Set directory path
  setwd(main_dir)

#Check path
  getwd()

#Get list of files in current directory
  dir()
```

1.6.2 Load Functions

There are a multitude of data storage formats in use such as extensible markup language (XML), comma separated values (CSV), Excel files among others. Each has its quirks. Some require special software to load while others simply need time and patience to crack the code. Fortunately, R is equipped to load virtually all data formats. Below is a recommended set of functions that are easy to use and flexible.

Table 1.3: Recommended functions to load an assortment of files.

Function	Package	Description
read_excel	readxl	Load Excel files into a data frame. Note that with excel files, generally need to specify which sheet to load using the *sheet* argument and may need to skip a number of rows at the beginning of worksheet to indicate the header row using *skip*.
read_csv	readr	Load any comma separated file into data frame format.
read_delim	readr	Load delimited file using any delimiter such as tab delimited ("\") using the *sep* argument.
readLines	base R	Read a text file, line by line. This is helpful for working with free form text.
xmlToDataFrame	XML	Read XML into a data frame if the structure is simple and flat. Note that XML files tend to have a more complex hierarchical structure.
xmlToList	XML	If the XML is complicated, read each element as a list.
fromJSON	rjson	Read JSON into list. If JSON is a flat, non-hierarchical file, then the resulting object can be simply converted into a data frame.
read_dta	haven	Load Stata data file.
read_sas	haven	Load SAS data files.
read_spss	haven	Load SPSS data files.
load	base R	Load saved R data set that can contain multiple objects. Note that this does not need to be assigned to another object.

Function	Package	Description
readRDS	base R	Read individually saved R object.

In the course of writing this text, the authors have come across a number of misconceptions about how to work with data, leading to haphazardous manual editing of data files. In particular, we have found that:

- Analysts will often manually open files and delete unnecessary rows at the top of a file as functions may not be able to read in files.
- The number of columns will differ, requiring columns to be deleted or empty columns added before loading.
- Column names differ, requiring manual editing of headers.
- The edits often overwrite the original file. This means that an edit were erroneous, there is no hope in restoring the original data.
- Manual edits are learned and shared through as an oral history.

These manual steps become tedious endeavor when presented with hundreds if not thousands of files. When loading and assembling data sets, the raw data should be stored as a copy. It then should be read into the programming environment where all processing is conducted. Since the processing is coded, it can be reproduced. The result is then stored as a new file.

In this section, we illustrate a simple workflow that is efficient and accurate. Our goal is to see if Energy Information Administration (EIA) spot gas prices recorded in New York Harbor and the US Gulf Coast are correlated. To do this, we will need to assemble a time series data set taking the following:

- Load a CSV of spot gas prices from the US Gulf Coast as a data frame.
- Load two Excel worksheets of spot gas prices from NY Harbor from two different time windows as data frames, then append one to the other.
- Combine the two into a single data frame as illustrated below, then calculate the Pearson Correlation coefficient.

Table 1.4: Target format

date	price1	price2
2016-12-28	1.739	1.702
2016-12-29	1.729	1.703
2016-12-30	1.722	1.699

Loading CSVs. Before reading in a file, it is generally a good idea to open the file in a code editor to take a peek at the structure, but "blind loading" the file can also give clues as well. Let's start with the CSV `doe_usgulf.csv` using the `read_csv` function from the `readr` package, assign the data to the object `gulf`, then inspect the first few rows using `head`.

```
pacman::p_load(readr)
gulf <- read_csv("data/doe_usgulf.csv")
head(gulf, 3)
```

```
## # A tibble: 3 x 2
##   Date   `U.S. Gulf Coast Conventional Gasoline Regular Spot Price FOB (Do~
##   <chr>                                                                <dbl>
## 1 1/2/14                                                                2.52
## 2 1/3/14                                                                2.49
## 3 1/6/14                                                                2.52
```

It appears that the data was successfully loaded. The data is read in as a *tibble*, which is a variant of a data frame designed for more efficient data manipulation. The headers require cosmetic changes. Generally, it is good practice to keep column names as the same case (lower case) and avoid the use of spaces and punctuation with the exception of periods. We can edit the column headers, which can be called as a vector using `colnames(gulf)`, by assigning a new string vector that turns all letters into lower case (`tolower`) and replaces substrings (`gsub`) that are eiher spaces or punctuation (`[[:space:][:punct:]]`) with periods. These string manipulation commands will be covered in more detail later in this chapter.

```
colnames(gulf) <- tolower(gsub("[[:space:][:punct:]]", ".", colnames(gulf)))
```

However, as these column names are quite long, we can opt to overwrite them with something more concise. Notice that we did not need to open the CSV and manually edit the column names.

```
colnames(gulf) <- c("date", "gulf.price")
```

Loading an Excel file. Excel files tend to have a few more curve balls than CSVs. While Excel files are more readable by aesthetic and cosmetic styling, parsing is challenging. Using `read_excel` from the `readxl` package, we take a first look at the first sheet (`sheet = 1`) of our Excel file `doe_ny.xlsx`, then use `head` to examine what was loaded. Unlike our CSV, our first attempt requires some tweaks: The `Date` header is treated as the first row of data and there are two additional empty columns.

```
pacman::p_load(readxl)
attempt1 <- read_excel("data/doe_ny.xlsx", sheet = 1)
head(attempt1, 3)
```

```
## # A tibble: 3 x 4
##   Sourcekey EER_EPMRU_PF4_Y35NY_DPG                              ...3  ...4
##   <chr>     <chr>                                                <lgl> <lgl>
## 1 Date      New York Harbor Conventional Gasoline Regular Spot~ NA    NA
## 2 41641     2.718                                                NA    NA
## 3 41642     2.6709999999999998                                   NA    NA
```

At this point, some may be tempted to manually delete the first row. Fortunately, we can skip rows when using most loading functions using a `skip` argument. Below, we read in the first and second sheets of `doe_ny.xlsx`, skipping the first row in each, then rename the headers.

```
#Load first two sheets
sheet1 <- read_excel("data/doe_ny.xlsx", sheet = 1, skip = 1)
sheet2 <- read_excel("data/doe_ny.xlsx", sheet = 2, skip = 1)
#Rename columns
colnames(sheet1) <- c("date", "ny.price")
colnames(sheet2) <- c("date", "ny.price")
```

Appending data. It turns out that *sheet1* and *sheet2* are from the same time series of spot covering different time windows. We can construct a new, more complete data frame by appending one data frame to the other using the `rbind` function.

```
  ny <- rbind(sheet1, sheet2)
```

The above scenario is the ideal: two data frames with the same number of columns and the same column names. But what if one data frame has at least one more column than the other? The `rbind.fill` function in the `plyr` package appends the two data frames together, filling any additional missing columns with `NA` values. Below, rather than loading the entire `plyr` library, we can selectively load functions using the double colon operator `::`.

```
  ny <- plyr::rbind.fill(sheet1, sheet2)
```

Note that if two columns in two different data frames represent the same concept but have different names, `rbind.fill` does not know to make the connection. Column names are interpreted literally, meaning that it is of the utmost importance that headers are exactly labeled.

Combining data by rows. The two data frames `gulf` and `ny` contain the same number of rows. If they are in the same exact desired order and have the same number of rows, we can use the `cbind` function to join the datasets by rows. The `cbind` function will join all columns in both data frames together, including the dates. To ensure the data is neatly organized, we keep all columns from the `gulf` dataset and only the second column from the `ny` data set. The resulting data frame contains three columns containing daily spot prices from two sources.

We calculate the correlation using the `cor` function finding a value of $\rho = 0.989617$ – the two sets of prices are highly correlated. Note that `cor` only accepts numeric values, thus the date column is temporarily dropped.

```
#Combine data
  prices <- cbind(gulf, ny[,2])
#Calculate correlation
  cor(prices[,-1])
```

```
##             gulf.price  ny[, 2]
## gulf.price   1.000000 0.989617
## ny[, 2]      0.989617 1.000000
```

1.7 Finding help

1.7.1 Help function

Developers interested in driving adoption of their packages understand that ease of use and clear documentation win the day. Thus, each R function is equipped with documentation to illustrate its syntax and use. As a first line of support, you can access any function's help file using `?<package_name>` in the console.

For instance, suppose you are interested in importing a spreadsheet using the `read_xlsx` from the `readxl` package, but you would like to have a better understand how to use it. Assuming that `readxl` has been loaded, simply execute the following in the console `?read_xlsx`. If the package has not already been loaded, include the package's name, double colons (`::`), and the function's name, e.g., `?readxl::read_xlsx`. The double colon trick essentially is short hand for obtaining a specific function from a library without importing the entire library and is not restricted to looking up help files.

1.7.2 Google and online communities

One of the best aspects of R is the vibrant and (generally) helpful online community. By simply searching "*R* ", you will likely discover that you are not the first to have faced the challenge at hand. To illustrate the point:

- *Leading zeroes* often are required to convert a number to a certain format. One of the zipcodes for Somerville, MA is `02144`, which sometimes is erroneously converted into a number `2144`. To find a solution to this formatting issue, simply search "*R leading zeroes*".
- *JSON* or JavaScript Object Notation is a data format that is increasingly used for web data. It requires a package to read and import. Search "*R import JSON*" to find a solution.

It quickly becomes apparent that Stack Overflow is a key resource for programming and Cross Validated for statistics and machine learning. More often than not, these online communities will have an answer and R coders hunting for an answer will copy and paste code without checking what it does. User beware, the answer might not be to your question, be the most efficient, or accurate. Nonetheless, these online resources are a great starting point.

1.8 Best practices: Commenting your scripts

As you may have noticed, many of the code snippets in this chapter contained comments preceded by a hash (`#`).[8] The hash tells R to ignore whatever follows the hash *on the same line as the* `#`. Thus, the hash allows you to make notes (generally called comments) to yourself, your collaborators, or anyone else who may have to read through your code. These comments help you figure out what you (or your friend) were doing two years ago when you wrote the R script. For instance:

```
# Make sure the packages are installed and loaded.
p_load(readxl, dplyr)

# Navigate to the data directory to load the data.
setwd("/Users/Someone/OnlyData")

# Load the xlsx file (named 'fake_data.xlsx')
fake_df <- read_xlsx("fake_data.xlsx")
```

In the example above, each comment line briefly describes what we will do in the next line and why we want to do it. Often the *why* is the most important part: with some experience, you will be able to figure out what each line of an R script does, but it is much more difficult to figure out *why* the author wrote the 641 lines of code in the middle of a 1200-line R script.

Another popular use of comments in R—and in many other languages—is to flag questions, issues, or unfinished items. The idea here is to start the comment with a quick flag, for instance `# TODO`. This flag helps you find the unfinished parts of your document—or the parts that you want your collaborators to finish for you. For instance, we might start a document

```
# Make sure the packages are installed and loaded
# FIXME: There are better packages for large datasets.
p_load(readxl, dplyr)

# Navigate to the data directory to load the data
setwd("/Users/Someone/OnlyData")

# Load the xlsx file (named 'fake_data.xlsx')
fake_df <- read_xlsx("fake_data.xlsx")

# TODO: Clean the dataset. Check out `stringr` and `data.table`
```

[8] Also called a *pound sign* or *number sign* or *octothorpe* in the olden days before Twitter.

You can also use commenting (`#`) to tell R to ignore lines of code. This usage of the hash is called *commenting out* your code, as it allows you to turn off a line of code, e.g.,

```
# Standard line of code
two_plus_two <- 2 + 2
# Commented out line of code
# two_times_two <- 2 * 2
```

In general, commenting lines of code out is most helpful for temporary changes and/or testing code. If you really do not need a code, just delete it.

Finally, it is worth saying that the style and amount of commenting, flagging, and commenting-out are highly subjective. If you find them burdensome, then you probably should cut back a bit or try a different strategy. That said, it's pretty rare for 0 comments to be an optimal number of comments in an R script.

1.9 DIY: Loading solar energy data from the web

Solar energy generation has been rapidly gaining ground. In the period between 2008 and 2017, annual net generation of solar energy produced by utility scale facilities had grown by 61.3-times, from 864 thousand megawatt hours to 52,958 thousand megawatt hours.[9]. At the same time, solar also became more economical: photovoltaic module costs fell from $3.37/peak watt to $0.48/peak watt – a 86% reduction in cost.

The increased affordability of solar among other advanced technologies opens the possibility for constructing buildings that are hyper energy efficient. For example, the Net Zero Energy Residential Test Facility is a house that produces as much energy as it uses over the course of a year. Engineered and constructed by the National Institute of Standards and Technology (NIST), the lab home was designed to be approximately 60 percent more energy efficient than typical homes. In order to achieve net zero energy, the lab home needs to produce an energy surplus and overcome the energy consumption of a simulated family of four. In fact, within its first year, the facility produced an energy surplus that is enough to power an electric car for 1400 miles.

The test bed also generates an extraordinary amount of data that help engineers study how to make more energy efficient homes and may one day inform building standards and policies. As a first step with R, we tap into one slice of the net zero house's photovoltaic time series data, which is made available publicly at https://s3.amazonaws.com/nist-netzero/2015-data-files/PV-hour.csv. This data set contains hourly estimates of solar energy production and exposure on the Net Zero home's solar panels.

Suppose we wanted to know how much energy the solar arrays are exposed to in an average hour and how variable that exposure is. With only a few lines of code, we can import and summarize the information. To start, we load the `pacman` package and use `p_load` to load the `readr` package.

```
#Load libraries
  pacman::p_load(readr)
```

With the basic functions loaded, `read_csv` is used to import a comma separated values (CSV) file stored at the given `url`, which is then loaded into memory as a data frame labeled `solar`. We can check the dimensions of the data frame using `dim`: it has n = 8,737 with 32 columns.

```
#Create string object with URL
  url <- "https://s3.amazonaws.com/nist-netzero/2015-data-files/PV-hour.csv"
```

[9] https://www.eia.gov/electricity/monthly/epm_table_grapher.php?t=epmt_1_01

```
#Read data direct from web
  solar <- read_csv(url)
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Timestamp = col_datetime(format = "")
## )
```

```
## See spec(...) for full column specifications.
```

```
#Check dimensions
  dim(solar)
```

```
## [1] 8737   32
```

As we are primarily interested in the total amount of sunlight hitting the solar arrays at any given hour (kWh), we focus in on the variable `PV_PVInsolationHArray` in data frame `solar`. To take a sneak peek of the data, we can use `head` to retrieve the first six observations.

```
  head(solar$PV_PVInsolationHArray)
```

```
## [1] 0.05654048 0.06022133 0.05883229 0.05876344 0.05934065 0.05871897
```

To answer our questions, we can obtain a concise `summary` to gauge the hourly variability. As it turns out, the summary indicates that the photovoltaic arrays are exposed to fairly small amounts of energy for the majority of hours as indicated by the small median relative to the mean, but there are occasional periods with intense energy exposure.

```
  summary(solar$PV_PVInsolationHArray)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##  0.00000  0.05576  0.32245  9.42320 14.73864 54.85163
```

This can be more concisely summarized as the coefficient of variation ($CV = \frac{\text{standard deviation}}{\text{mean}}$) – the ratio of the standard deviation and the mean. Values of the CV that exceed $CV = 1$ indicate greater dispersion. Otherwise stated, the data may be *wider than it is tall*, therefore greater vales of CV indicate there may be more noise and less consistency in the data. We compute the `mean` and standard deviation `sd`. The resulting CV indicates that one standard deviation is 1.5-times as wide as the mean, suggesting some visible degree of variability in hourly sun exposure.

```
  a <- mean(solar$PV_PVInsolationHArray)
  b <- sd(solar$PV_PVInsolationHArray)
  print(b/a)
```

```
## [1] 1.541016
```

With only a few lines of code, we have shown how programming can make data analysis an efficient process. In the context of policy, programming removes the tedium of spreadsheeting, allowing an analysis to be delivered using only a few decisive keystrokes.