Chapter 6. Workflow: Scripts and Projects

This chapter will introduce you to two essential tools for organizing your code: scripts and projects.

Scripts

So far, you have used the console to run code. That's a great place to start, but you'll find it gets cramped pretty quickly as you create more complex ggplot2 graphics and longer dplyr pipelines. To give yourself more room to work, use the script editor. Open it by clicking the File menu, selecting New File, and then selecting R script, or using the keyboard shortcut Cmd/Ctrl+Shift+N. Now you'll see four panes, as in Figure 6-1. The script editor is a great place to experiment with your code. When you want to change something, you don't have to retype the whole thing; you can just edit the script and rerun it. And once you have written code that works and does what you want, you can save it as a script file to easily return to later.

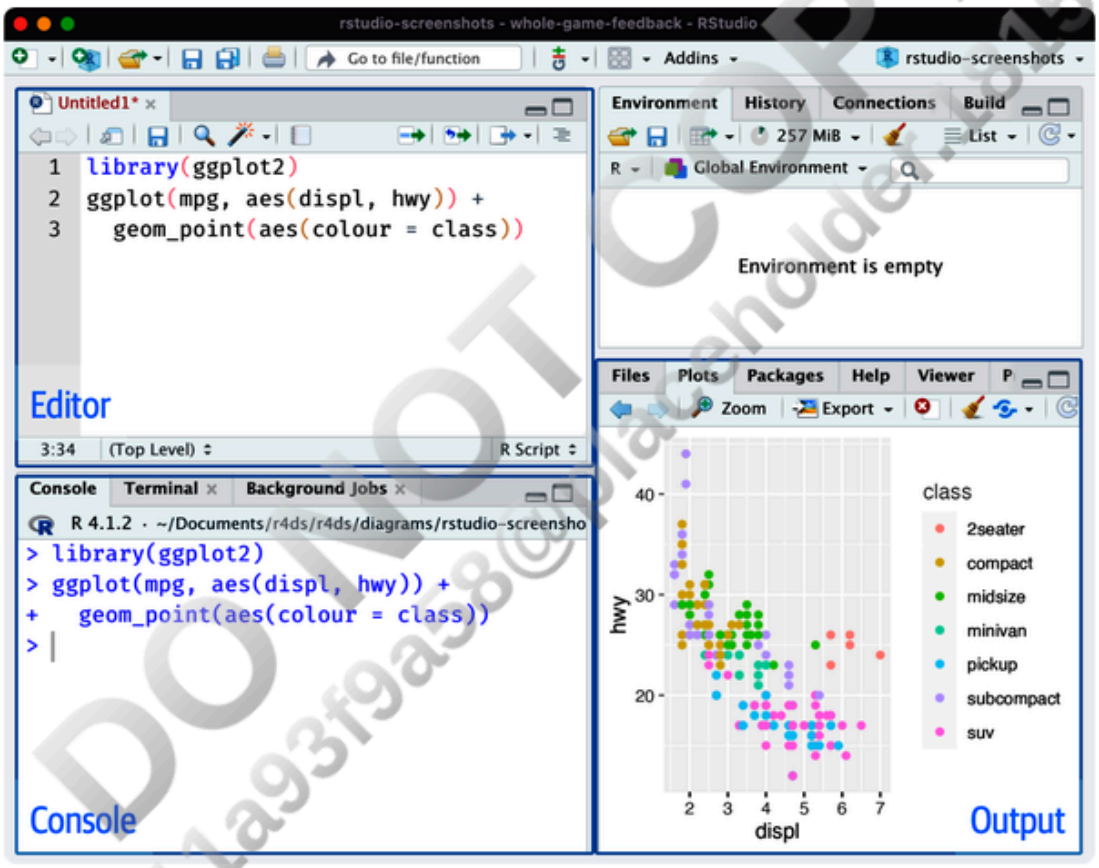


Figure 6-1. Opening the script editor adds a new pane at the top left of the IDE.

Running Code

The script editor is an excellent place for building complex ggplot2 plots or long sequences of dplyr manipulations. The key to using the script editor effectively is to memorize one of the most important keyboard shortcuts: Cmd/Ctrl+Enter. This executes the current R expression in the console. For example, take the following code:

```
library(dplyr)
library(nycflights13)

not_cancelled <- flights |>
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled |>
```

```
  group_by(year, month, day) |>
  summarize(mean = mean(dep_delay))
```

If your cursor is at █, pressing Cmd/Ctrl+Enter will run the complete command that generates `not_cancelled`. It will also move the cursor to the following statement (beginning with `not_cancelled |>`). That makes it easy to step through your complete script by repeatedly pressing Cmd/Ctrl+Enter.

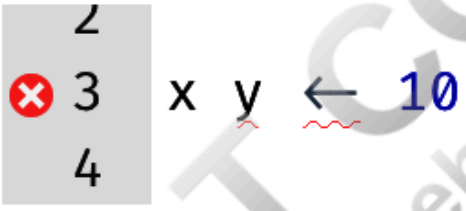
Instead of running your code expression by expression, you can execute the complete script in one step with Cmd/Ctrl+Shift+S. Doing this regularly is a great way to ensure that you've captured all the important parts of your code in the script.

We recommend you always start your script with the packages you need. That way, if you share your code with others, they can easily see which packages they need to install. Note, however, that you should never include `install.packages()` in a script you share. It's inconsiderate to hand off a script that will install something on their computer if they're not being careful!

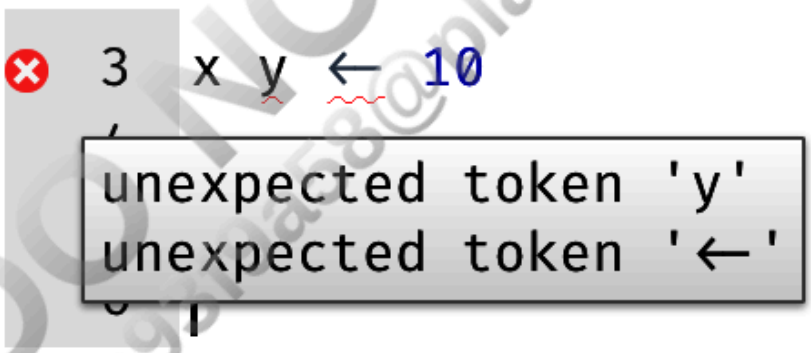
When working through future chapters, we highly recommend starting in the script editor and practicing your keyboard shortcuts. Over time, sending code to the console in this way will become so natural that you won't even think about it.

RStudio Diagnostics

In the script editor, RStudio will highlight syntax errors with a red squiggly line and a cross in the sidebar:



Hover over the cross to see what the problem is:



RStudio will also let you know about potential problems:

Saving and Naming

RStudio automatically saves the contents of the script editor when you quit and automatically reloads it when you re-open. Nevertheless, it's a good idea to avoid Untitled1, Untitled2, Untitled3, and so on, and instead save your scripts with informative names.

It might be tempting to name your files `code.R` or `myscript.R`, but you should think a bit harder before choosing a name for your file. Three important principles for file naming are as follows:

1. Filenames should be *machine* readable: avoid spaces, symbols, and special characters. Don't rely on case sensitivity to distinguish files.
2. Filenames should be *human* readable: use filenames to describe what's in the file.
3. Filenames should play well with default ordering: start filenames with numbers so that alphabetical sorting puts them in the order they get used.

For example, suppose you have the following files in a project folder:

```
alternative model.R
code for exploratory analysis.r
finalreport.qmd
FinalReport.qmd
fig 1.png
Figure_02.png
model_first_try.R
run-first.r
temp.txt
```

There are a variety of problems here: it's hard to find which file to run first, filenames contain spaces, there are two files with the same name but different capitalization (`finalreport` versus `FinalReport`[1]), and some names don't describe their contents (`run-first` and `temp`).

Here's a better way of naming and organizing the same set of files:

```
01-load-data.R
02-exploratory-analysis.R
03-model-approach-1.R
04-model-approach-2.R
fig-01.png
fig-02.png
report-2022-03-20.qmd
report-2022-04-02.qmd
report-draft-notes.txt
```

Numbering the key scripts makes it obvious in which order to run them, and a consistent naming scheme makes it easier to see what varies. Additionally, the figures are labeled similarly, the reports are distinguished by dates included in the filenames, and `temp` is renamed to `report-draft-notes` to better describe its contents. If you have a lot of files in a directory, taking organization one step further and placing different types of files (scripts, figures, etc.) in different directories is recommended.

Projects

One day, you will need to quit R, go do something else, and return to your analysis later. One day, you will be working on multiple analyses simultaneously and want to keep them separate. One day, you will need to bring data from the outside world into R and send numerical results and figures from R back out into the world.

To handle these real-life situations, you need to make two decisions:

- What is the source of truth? What will you save as your lasting record of what happened?
- Where does your analysis live?

What Is the Source of Truth?

As a beginner, it's OK to rely on your current environment to contain all the objects you have created throughout your analysis. However, to make it easier to work on larger projects or collaborate with others, your source of truth should be the R scripts

However, to make it easier to work on larger projects or collaborate with others, your source of truth should be the R scripts. With your R scripts (and your data files), you can re-create the environment. With only your environment, it's much harder to re-create your R scripts: either you'll have to retype a lot of code from memory (inevitably making mistakes along the way) or you'll have to carefully mine your R history.

To help keep your R scripts as the source of truth for your analysis, we highly recommend that you instruct RStudio not to preserve your workspace between sessions. You can do this either by running `usethis::use_blank_slate()`[2] or by mimicking the options shown in Figure 6-2. This will cause you some short-term pain, because now when you restart RStudio, it will no longer remember the code that you ran last time nor will the objects you created or the datasets you read be available to use. But this short-term pain saves you long-term agony because it forces you to capture all important procedures in your code. There's nothing worse than discovering three months after the fact that you've stored only the results of an important calculation in your environment, not the calculation itself in your code.

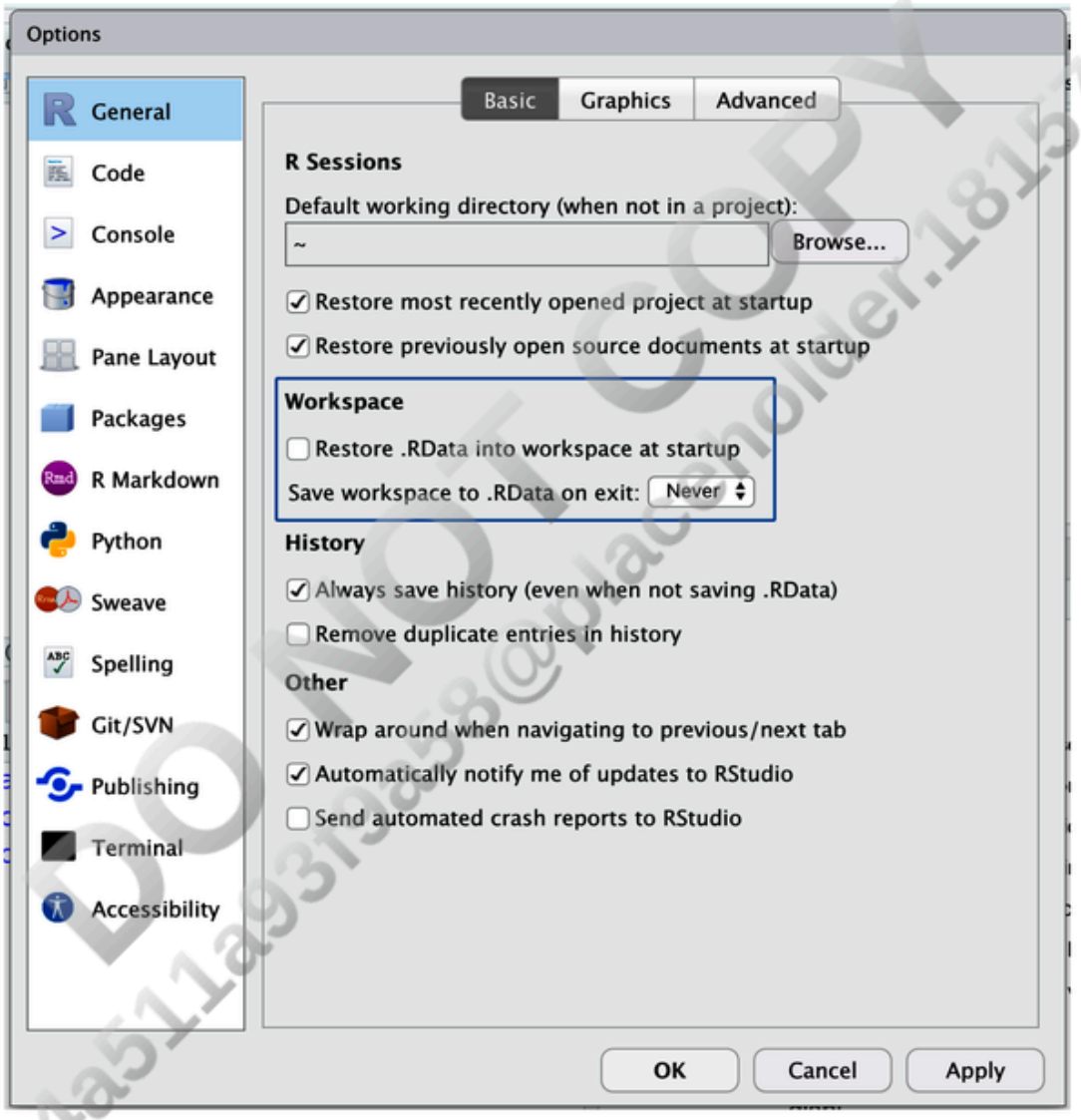


Figure 6-2. Copy these selections in your RStudio options to always start your RStudio session with a clean slate.

There is a great pair of keyboard shortcuts that will work together to make sure you've captured the important parts of your code in the editor:

1. Press Cmd/Ctrl+Shift+0/F10 to restart R.
2. Press Cmd/Ctrl+Shift+S to rerun the current script.

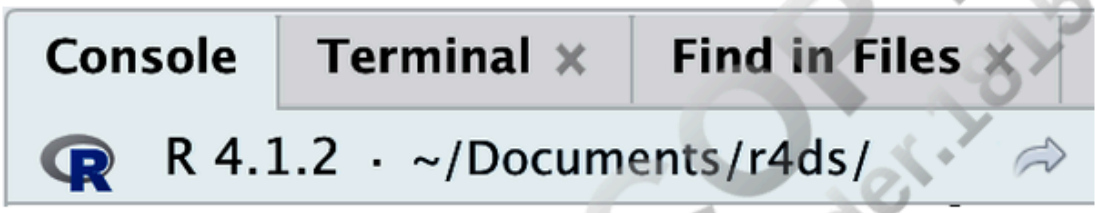
We collectively use this pattern hundreds of times a week.

Alternatively, if you don't use keyboard shortcuts, you can select Session > Restart R and then highlight and rerun your current script.

> **RSTUDIO SERVER**
>
> If you're using RStudio Server, your R session is never restarted by default. When you close your RStudio Server tab, it might feel like you're closing R, but the server actually keeps it running in the background. The next time you return, you'll be in exactly the same place you left. This makes it even more important to regularly restart R so that you're starting with a clean slate.

Where Does Your Analysis Live?

R has a powerful notion of the *working directory*. This is where R looks for files that you ask it to load and where it will put any files that you ask it to save. RStudio shows your current working directory at the top of the console:



You can print this out in R code by running `getwd()`:

```
getwd()
#> [1] "/Users/hadley/Documents/r4ds"
```

In this R session, the current working directory (think of it as "home") is in Hadley's *Documents* folder, in a subfolder called *r4ds*. This code will return a different result when you run it, because your computer has a different directory structure than Hadley's!

As a beginning R user, it's OK to let your working directory be your home directory, documents directory, or any other weird directory on your computer. But you're seven chapters into this book, and you're no longer a beginner. Soon you should evolve to organizing your projects into directories and, when working on a project, set R's working directory to the associated directory.

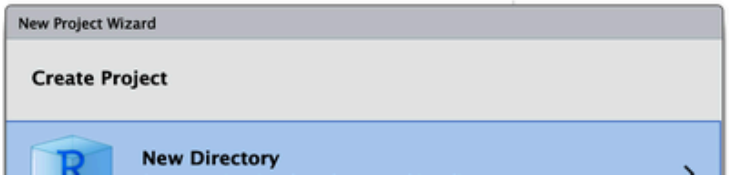
You can set the working directory from within R, but *we do not recommend it*:

```
setwd("/path/to/my/CoolProject")
```

There's a better way—a way that also puts you on the path to managing your R work like an expert. That way is the *RStudio project*.

RStudio Projects

Keeping all the files associated with a given project (input data, R scripts, analytical results, and figures) in one directory is such a wise and common practice that RStudio has built-in support for this via *projects*. Let's make a project for you to use while you're working through the rest of this book. Select File > New Project, and then follow the steps shown in Figure 6-3.



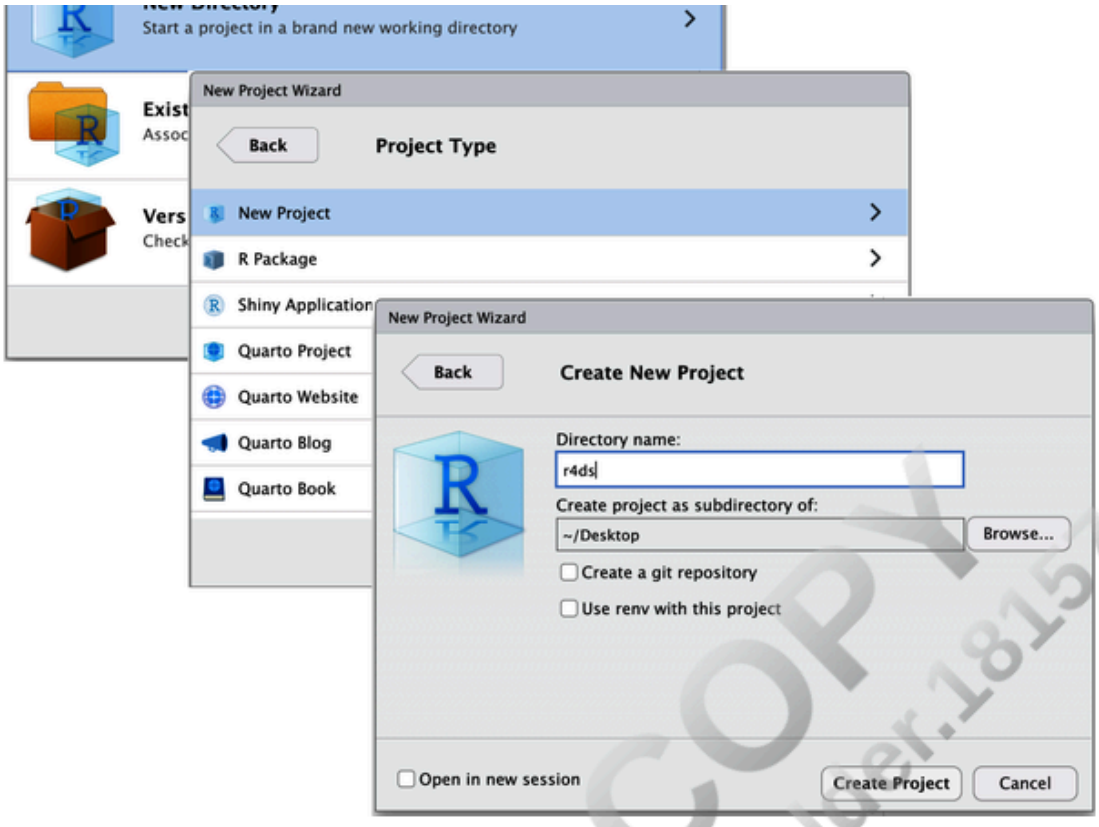


Figure 6-3. To create new project: (top) first click New Directory, then (middle) click New Project, then (bottom) fill in the directory (project) name, choose a good subdirectory for its home, and click Create Project.

Call your project `r4ds` and think carefully about which subdirectory you put the project in. If you don't store it somewhere sensible, it will be hard to find it in the future!

Once this process is complete, you'll get a new RStudio project just for this book. Check that the "home" of your project is the current working directory:

```
getwd()
#> [1] /Users/hadley/Documents/r4ds
```

Now enter the following commands in the script editor and save the file, calling it `diamonds.R`. Then, create a new folder called `data`. You can do this by clicking the New Folder button in the Files pane in RStudio. Finally, run the complete script, which will save a PNG and CSV file into your project directory. Don't worry about the details; you'll learn them later in the book.

```
library(tidyverse)

ggplot(diamonds, aes(x = carat, y = price)) +
  geom_hex()
ggsave("diamonds.png")

write_csv(diamonds, "data/diamonds.csv")
```

Quit RStudio. Inspect the folder associated with your project—notice the `.Rproj` file. Double-click that file to re-open the project. Notice you get back to where you left off: it's the same working directory and command history, and all the files you were working on are still open. Because you followed our instructions, you will, however, have a completely fresh environment, guaranteeing that you're starting with a clean slate.

In your favorite OS-specific way, search your computer for `diamonds.png`, and you will find the PNG (no surprise) but *also the script that created it* (`diamonds.R`). This is a huge win! One day, you will want to remake a figure or just understand where it came from. If you rigorously save figures to files *with R code* and never with the mouse or the clipboard, you will be able to

reproduce old work with ease!

Relative and Absolute Paths

Once you're inside a project, you should only ever use relative paths, not absolute paths. What's the difference? A relative path is relative to the working directory, i.e., the project's home. When Hadley wrote `data/diamonds.csv` earlier, it was a shortcut for `/Users/hadley/Documents/r4ds/data/diamonds.csv`. But importantly, if Mine ran this code on her computer, it would point to `/Users/Mine/Documents/r4ds/data/diamonds.csv`. This is why relative paths are important: they'll work regardless of where the R project folder ends up.

Absolute paths point to the same place regardless of your working directory. They look a little different depending on your operating system. On Windows they start with a drive letter (e.g., `C:`) or two backslashes (e.g., `\\servername`) and on Mac/Linux they start with a slash, / (e.g., `/users/hadley`). You should *never* use absolute paths in your scripts, because they hinder sharing: no one else will have exactly the same directory configuration as you.

There's another important difference between operating systems: how you separate the components of the path. Mac and Linux uses slashes (e.g., `data/diamonds.csv`), and Windows uses backslashes (e.g., `data\diamonds.csv`). R can work with either type (no matter what platform you're currently using), but unfortunately, backslashes mean something special to R, and to get a single backslash in the path, you need to type two backslashes! That makes life frustrating, so we recommend always using the Linux/Mac style with forward slashes.

Exercises

1. Go to the RStudio Tips Twitter account and find one tip that looks interesting. Practice using it!
2. What other common mistakes will RStudio diagnostics report? Read this article on code diagnostics to find out.

Summary

In this chapter, you learned how to organize your R code in scripts (files) and projects (directories). Much like code style, this may feel like busywork at first. But as you accumulate more code across multiple projects, you'll learn to appreciate how a little up-front organization can save you a bunch of time later.

In summary, scripts and projects give you a solid workflow that will serve you well in the future:

- Create one RStudio project for each data analysis project.
- Save your scripts (with informative names) in the project, edit them, and run them in bits or as a whole. Restart R frequently to make sure you've captured everything in your scripts.
- Only ever use relative paths, not absolute paths.

Then everything you need is in one place and cleanly separated from all the other projects you are working on.

So far, we've worked with datasets bundled in R packages. This makes it easier to get some practice on preprepared data, but obviously your data won't be available in this way. So in the next chapter, you're going to learn how load data from disk into your R session using the readr package.

---

[1] Not to mention that you're tempting fate by using "final" in the name. The comic Piled Higher and Deeper has a fun strip on this.

[2] If you don't have this installed, you can install it with `install.packages("usethis")`.

Chapter 7. Data Import

Introduction

Working with data provided by R packages is a great way to learn data science tools, but you want to apply what you've learned to your own data at some point. In this chapter, you'll learn the basics of reading data files into R.

Specifically, this chapter will focus on reading plain-text rectangular files. We'll start with practical advice for handling features such as column names, types, and missing data. You will then learn about reading data from multiple files at once and writing data from R to a file. Finally, you'll learn how to handcraft data frames in R.

Prerequisites

In this chapter, you'll learn how to load flat files in R with the readr package, which is part of the core tidyverse:

```
library(tidyverse)
```

Reading Data from a File

To begin, we'll focus on the most common rectangular data file type: CSV, which is short for "comma-separated values." Here is what a simple CSV file looks like. The first row, commonly called the *header row*, gives the column names, and the following six rows provide the data. The columns are separated, aka *delimited*, by commas.

```
Student ID,Full Name,favourite.food,mealPlan,AGE
1,Sunil Huffmann,Strawberry yoghurt,Lunch only,4
2,Barclay Lynn,French fries,Lunch only,5
3,Jayendra Lyne,N/A,Breakfast and lunch,7
4,Leon Rossini,Anchovies,Lunch only,
5,Chidiegwu Dunkel,Pizza,Breakfast and lunch,five
6,Güvenç Attila,Ice cream,Lunch only,6
```

Table 7-1 represents of the same data as a table.

Table 7-1. Data from the students.csv file as a table

Student ID	Full Name	favourite.food	mealPlan	AGE
1	Sunil Huffmann	Strawberry yoghurt	Lunch only	4
2	Barclay Lynn	French fries	Lunch only	5
3	Jayendra Lyne	N/A	Breakfast and lunch	7
4	Leon Rossini	Anchovies	Lunch only	NA
5	Chidiegwu Dunkel	Pizza	Breakfast and lunch	five
6	Güvenç Attila	Ice cream	Lunch only	6

We can read this file into R using `read_csv()`. The first argument is the most important: the path to the file. You can think about the path as the address of the file: the file is called `students.csv`, and it lives in the `data` folder.

```
students <- read_csv("data/students.csv")
#> Rows: 6 Columns: 5
#> ── Column specification ────────────────────────────────────────────────
```

```
#> ── Column specification ────────────────────────────────────────────────────────
#> Delimiter: ","
#> chr (4): Full Name, favourite.food, mealPlan, AGE
#> dbl (1): Student ID
#>
#> ℹ Use `spec()` to retrieve the full column specification for this data.
#> ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

The previous code will work if you have the `students.csv` file in a `data` folder in your project. You can download the `students.csv` file or you can read it directly from that URL with this:

```
students <- read_csv("https://pos.it/r4ds-students-csv")
```

When you run `read_csv()`, it prints out a message telling you the number of rows and columns of data, the delimiter that was used, and the column specifications (names of columns organized by the type of data the column contains). It also prints out some information about retrieving the full column specification and how to quiet this message. This message is an integral part of readr, and we'll return to it in "Controlling Column Types".

Practical Advice

Once you read data in, the first step usually involves transforming it in some way to make it easier to work with in the rest of your analysis. Let's take another look at the `students` data with that in mind:

```
students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan             AGE
#>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only           4
#> 2            2 Barclay Lynn     French fries       Lunch only           5
#> 3            3 Jayendra Lyne    N/A                Breakfast and lunch 7
#> 4            4 Leon Rossini     Anchovies          Lunch only
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6            6 Güvenç Attila    Ice cream          Lunch only           6
```

In the `favourite.food` column, there are a bunch of food items, and then the character string `N/A`, which should have been a real `NA` that R will recognize as "not available." This is something we can address using the `na` argument. By default `read_csv()` recognizes only empty strings (`""`) in this dataset as `NA`s; we want it to also recognize the character string `"N/A"`:

```
students <- read_csv("data/students.csv", na = c("N/A", ""))

students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan             AGE
#>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only           4
#> 2            2 Barclay Lynn     French fries       Lunch only           5
#> 3            3 Jayendra Lyne                     Breakfast and lunch 7
#> 4            4 Leon Rossini     Anchovies          Lunch only
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6            6 Güvenç Attila    Ice cream          Lunch only           6
```

You might also notice that the `Student ID` and `Full Name` columns are surrounded by backticks. That's because they contain spaces, breaking R's usual rules for variable names; they're *nonsyntactic* names. To refer to these variables, you need to surround them with backticks, `` ` ``:

```
students |>
  rename(
    student_id = `Student ID`,
    full_name = `Full Name`
  )
#> # A tibble: 6 × 5
#>   student_id full_name        favourite.food     mealPlan             AGE
```

```
#>   student_id full_name        favourite.food     mealplan             age
#>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only           4
#> 2          2 Barclay Lynn     French fries       Lunch only           5
#> 3          3 Jayendra Lyne                    Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only           6
```

An alternative approach is to use `janitor::clean_names()` to use some heuristics to turn them all into snake case at once:[1]

```
students |> janitor::clean_names()
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne                    Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

Another common task after reading in data is to consider variable types. For example, `meal_plan` is a categorical variable with a known set of possible values, which in R should be represented as a factor:

```
students |>
  janitor::clean_names() |>
  mutate(meal_plan = factor(meal_plan))
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne                    Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

Note that the values in the `meal_plan` variable have stayed the same, but the type of variable denoted underneath the variable name has changed from character () to factor (). You'll learn more about factors in Chapter 16.

Before you analyze these data, you'll probably want to fix the `age` and `id` columns. Currently, `age` is a character variable because one of the observations is typed out as `five` instead of a numeric `5`. We discuss the details of fixing this issue in Chapter 20.

```
students <- students |>
  janitor::clean_names() |>
  mutate(
    meal_plan = factor(meal_plan),
    age = parse_number(if_else(age == "five", "5", age))
  )

students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne                    Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

A new function here is `if_else()`, which has three arguments. The first argument `test` should be a logical vector. The result will contain the value of the second argument, `yes`, when `test` is `TRUE`, and the value of the third argument, `no`, when it is `FALSE`. Here we're saying if `age` is the character string `"five"`, make it `"5"`, and if not, leave it as `age`. You will learn more

about `if_else()` and logical vectors in Chapter 12.

Other Arguments

There are a couple of other important arguments that we need to mention, and they'll be easier to demonstrate if we first show you a handy trick: `read_csv()` can read text strings that you've created and formatted like a CSV file:

```
read_csv(
  "a,b,c
  1,2,3
  4,5,6"
)
#> # A tibble: 2 × 3
#>       a     b     c
#>
#> 1     1     2     3
#> 2     4     5     6
```

Usually, `read_csv()` uses the first line of the data for the column names, which is a common convention. But it's not uncommon for a few lines of metadata to be included at the top of the file. You can use `skip = n` to skip the first `n` lines or use `comment = "#"` to drop all lines that start with, for example, `#`:

```
read_csv(
  "The first line of metadata
  The second line of metadata
  x,y,z
  1,2,3",
  skip = 2
)
#> # A tibble: 1 × 3
#>       x     y     z
#>
#> 1     1     2     3

read_csv(
  "# A comment I want to skip
  x,y,z
  1,2,3",
  comment = "#"
)
#> # A tibble: 1 × 3
#>       x     y     z
#>
#> 1     1     2     3
```

In other cases, the data might not have column names. You can use `col_names = FALSE` to tell `read_csv()` not to treat the first row as headings and instead label them sequentially from `X1` to `Xn`:

```
read_csv(
  "1,2,3
  4,5,6",
  col_names = FALSE
)
#> # A tibble: 2 × 3
#>      X1    X2    X3
#>
#> 1     1     2     3
#> 2     4     5     6
```

Alternatively, you can pass `col_names` a character vector, which will be used as the column names:

```
read_csv(
  "1,2,3
  4,5,6",
  col_names = c("x", "y", "z")
)
```

```
  col_names = c("x", "y", "z")
)
#> # A tibble: 2 × 3
#>       x     y     z
#>
#> 1     1     2     3
#> 2     4     5     6
```

These arguments are all you need to know to read the majority of CSV files that you'll encounter in practice. (For the rest, you'll need to carefully inspect your `.csv` file and read the documentation for `read_csv()`'s many other arguments.)

Other File Types

Once you've mastered `read_csv()`, using readr's other functions is straightforward; it's just a matter of knowing which function to reach for:

`read_csv2()`

Reads semicolon-separated files. These use `;` instead of `,` to separate fields and are common in countries that use `,` as the decimal marker.

`read_tsv()`

Reads tab-delimited files.

`read_delim()`

Reads in files with any delimiter, attempting to automatically guess the delimiter if you don't specify it.

`read_fwf()`

Reads fixed-width files. You can specify fields by their widths with `fwf_widths()` or by their positions with `fwf_positions()`.

`read_table()`

Reads a common variation of fixed-width files where columns are separated by whitespace.

`read_log()`

Reads Apache-style log files.

Exercises

1. What function would you use to read a file where fields were separated with |?
2. Apart from `file`, `skip`, and `comment`, what other arguments do `read_csv()` and `read_tsv()` have in common?
3. What are the most important arguments to `read_fwf()`?
4. Sometimes strings in a CSV file contain commas. To prevent them from causing problems, they need to be surrounded by a quoting character, like `"` or `'`. By default, `read_csv()` assumes that the quoting character will be `"`. To read the following text into a data frame, what argument to `read_csv()` do you need to specify?

```
"x,y\n1,'a,b'"
```

5. Identify what is wrong with each of the following inline CSV files. What happens when you run the code?

```
read_csv("a,b\n1,2,3\n4,5,6")
read_csv("a,b,c\n1,2\n1,2,3,4")
read_csv("a,b\n\"1")
```

```
read_csv("a,b\n\"1")
read_csv("a,b\n1,2\na,b")
read_csv("a;b\n1;3")
```

6. Practice referring to nonsyntactic names in the following data frame by:

   a. Extracting the variable called `1`.

   b. Plotting a scatterplot of `1` versus `2`.

   c. Creating a new column called `3`, which is `2` divided by `1`.

   d. Renaming the columns to `one`, `two`, and `three`:

```
annoying <- tibble(
  `1` = 1:10,
  `2` = `1` * 2 + rnorm(length(`1`))
)
```

Controlling Column Types

A CSV file doesn't contain any information about the type of each variable (i.e., whether it's a logical, number, string, etc.), so readr will try to guess the type. This section describes how the guessing process works, how to resolve some common problems that cause it to fail, and, if needed, how to supply the column types yourself. Finally, we'll mention a few general strategies that are useful if readr is failing catastrophically and you need to get more insight into the structure of your file.

Guessing Types

readr uses a heuristic to figure out the column types. For each column, it pulls the values of 1,000[2] rows spaced evenly from the first row to the last, ignoring missing values. It then works through the following questions:

- Does it contain only `F`, `T`, `FALSE`, or `TRUE` (ignoring case)? If so, it's a logical.
- Does it contain only numbers (e.g., `1`, `-4.5`, `5e6`, `Inf`)? If so, it's a number.
- Does it match the ISO8601 standard? If so, it's a date or date-time. (We'll return to date-times in more detail in "Creating Date/Times".)
- Otherwise, it must be a string.

You can see that behavior in action in this simple example:

```
read_csv("
  logical,numeric,date,string
  TRUE,1,2021-01-15,abc
  false,4.5,2021-02-15,def
  T,Inf,2021-02-16,ghi
")
#> # A tibble: 3 × 4
#>   logical numeric date       string
#>
#> 1 TRUE        1   2021-01-15 abc
#> 2 FALSE       4.5 2021-02-15 def
#> 3 TRUE      Inf   2021-02-16 ghi
```

This heuristic works well if you have a clean dataset, but in real life, you'll encounter a selection of weird and beautiful failures.

Missing Values, Column Types, and Problems

The most common way column detection fails is that a column contains unexpected values, and you get a character column instead of a more specific type. One of the most common causes for this is a missing value, recorded using something other than the `NA` that readr expects.

Take this simple one-column CSV file as an example:

```
simple_csv <- "
  x
  10
  .
  20
  30"
```

If we read it without any additional arguments, `x` becomes a character column:

```
read_csv(simple_csv)
#> # A tibble: 4 × 1
#>   x
#>
#> 1 10
#> 2 .
#> 3 20
#> 4 30
```

In this small case, you can easily see the missing value `.`. But what happens if you have thousands of rows with only a few missing values represented by `.`s sprinkled among them? One approach is to tell readr that `x` is a numeric column and then see where it fails. You can do that with the `col_types` argument, which takes a named list where the names match the column names in the CSV file:

```
df <- read_csv(
  simple_csv,
  col_types = list(x = col_double())
)
#> Warning: One or more parsing issues, call `problems()` on your data frame for
#> details, e.g.:
#>   dat <- vroom(...)
#>   problems(dat)
```

Now `read_csv()` reports that there was a problem and tells us we can find out more with `problems()`:

```
problems(df)
#> # A tibble: 1 × 5
#>     row   col expected actual file
#>
#> 1     3     1 a double .      /private/tmp/RtmpAYlSop/file392d445cf269
```

This tells us that there was a problem in row 3, column 1 where readr expected a double but got a `.`. That suggests this dataset uses `.` for missing values. So then we set `na = "."`, and the automatic guessing succeeds, giving us the numeric column that we want:

```
read_csv(simple_csv, na = ".")
#> # A tibble: 4 × 1
#>       x
#>
#> 1    10
#> 2    NA
#> 3    20
#> 4    30
```

Column Types

readr provides a total of nine column types for you to use:

- `col_logical()` and `col_double()` read logicals and real numbers. They're relatively rarely needed (except as shown previously), since readr will usually guess them for you.
- `col_integer()` reads integers. We seldom distinguish integers and doubles in this book because they're functionally equivalent, but reading integers explicitly can occasionally be useful because they occupy half the memory of doubles.
- `col_character()` reads strings. This can be useful to specify explicitly when you have a column that is a numeric identifier, i.e., long series of digits that identifies an object but doesn't make sense to apply mathematical operations to. Examples include phone numbers, Social Security numbers, credit card numbers, and so on.
- `col_factor()`, `col_date()`, and `col_datetime()` create factors, dates, and date-times, respectively; you'll learn more about those when we get to those data types in Chapter 16 and Chapter 17.
- `col_number()` is a permissive numeric parser that will ignore non-numeric components and is particularly useful for currencies. You'll learn more about it in Chapter 13.
- `col_skip()` skips a column so it's not included in the result, which can be useful for speeding up reading the data if you have a large CSV file and you want to use only some of the columns.

It's also possible to override the default column by switching from `list()` to `cols()` and specifying `.default`:

```
another_csv <- "
x,y,z
1,2,3"

read_csv(
  another_csv,
  col_types = cols(.default = col_character())
)
#> # A tibble: 1 × 3
#>   x     y     z
#>
#> 1 1     2     3
```

Another useful helper is `cols_only()`, which will read in only the columns you specify:

```
read_csv(
  another_csv,
  col_types = cols_only(x = col_character())
)
#> # A tibble: 1 × 1
#>   x
#>
#> 1 1
```

Reading Data from Multiple Files

Sometimes your data is split across multiple files instead of being contained in a single file. For example, you might have sales data for multiple months, with each month's data in a separate file: `01-sales.csv` for January, `02-sales.csv` for February, and `03-sales.csv` for March. With `read_csv()` you can read these data in at once and stack them on top of each other in a single data frame.

```
sales_files <- c("data/01-sales.csv", "data/02-sales.csv", "data/03-sales.csv")
read_csv(sales_files, id = "file")
#> # A tibble: 19 × 6
#>   file              month    year brand  item     n
#>
#> 1 data/01-sales.csv January  2019     1  1234     3
```

```
#> 1 data/01-sales.csv January  2019     1  1234     3
#> 2 data/01-sales.csv January  2019     1  8721     9
#> 3 data/01-sales.csv January  2019     1  1822     2
#> 4 data/01-sales.csv January  2019     2  3333     1
#> 5 data/01-sales.csv January  2019     2  2156     9
#> 6 data/01-sales.csv January  2019     2  3987     6
#> # … with 13 more rows
```

Once again, the previous code will work if you have the CSV files in a `data` folder in your project. You can download these files from *https://oreil.ly/jVd8o*, *https://oreil.ly/RYsgM*, and *https://oreil.ly/4uZOm* or you can read them directly with:

```
sales_files <- c(
  "https://pos.it/r4ds-01-sales",
  "https://pos.it/r4ds-02-sales",
  "https://pos.it/r4ds-03-sales"
)
read_csv(sales_files, id = "file")
```

The `id` argument adds a new column called `file` to the resulting data frame that identifies the file the data come from. This is especially helpful in circumstances where the files you're reading in do not have an identifying column that can help you trace the observations back to their original sources.

If you have many files you want to read in, it can get cumbersome to write out their names as a list. Instead, you can use the base `list.files()` function to find the files for you by matching a pattern in the filenames. You'll learn more about these patterns in Chapter 15.

```
sales_files <- list.files("data", pattern = "sales\\.csv$", full.names = TRUE)
sales_files
#> [1] "data/01-sales.csv" "data/02-sales.csv" "data/03-sales.csv"
```

Writing to a File

readr also comes with two useful functions for writing data to disk: `write_csv()` and `write_tsv()`. The most important arguments to these functions are `x` (the data frame to save) and `file` (the location to save it). You can also specify how missing values are written with `na`, as well as whether you want to `append` to an existing file.

```
write_csv(students, "students.csv")
```

Now let's read that CSV file back in. Note that the variable type information that you just set up is lost when you save to CSV because you're starting over with reading from a plain-text file again:

```
students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne                    Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
write_csv(students, "students-2.csv")
read_csv("students-2.csv")
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne                    Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

This makes CSVs a little unreliable for caching interim results—you need to re-create the column specification every time you load in. There are two main alternatives:

- `write_rds()` and `read_rds()` are uniform wrappers around the base functions `readRDS()` and `saveRDS()`. These store data in R's custom binary format called RDS. This means that when you reload the object, you are loading the *exact same* R object that you stored.

```
write_rds(students, "students.rds")
read_rds("students.rds")
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne                    Breakfast and lunch      7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

- The arrow package allows you to read and write parquet files, a fast binary file format that can be shared across programming languages. We'll return to arrow in more depth in Chapter 22.

```
library(arrow)
write_parquet(students, "students.parquet")
read_parquet("students.parquet")
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    NA                 Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

Parquet tends to be much faster than RDS and is usable outside of R but does require the arrow package.

Data Entry

Sometimes you'll need to assemble a tibble "by hand" doing a little data entry in your R script. There are two useful functions to help you do this, which differ in whether you lay out the tibble by columns or by rows. `tibble()` works by column:

```
tibble(
  x = c(1, 2, 5),
  y = c("h", "m", "g"),
  z = c(0.08, 0.83, 0.60)
)
#> # A tibble: 3 × 3
#>       x y         z
#>
#> 1     1 h      0.08
#> 2     2 m      0.83
#> 3     5 g      0.6
```

Laying out the data by column can make it hard to see how the rows are related, so an alternative is `tribble()`, short for *tr*ansposed t*ibble*, which lets you lay out your data row by row. `tribble()` is customized for data entry in code: column headings start with ~ and entries are separated by commas. This makes it possible to lay out small amounts of data in an easy-to-read form:

```
tribble(
  ~x, ~y, ~z,
  1, "h", 0.08,
  2, "m", 0.83,
  5, "g", 0.60
)
#> # A tibble: 3 × 3
#>   x         y     z
#>
#> 1         1 h  0.08
#> 2         2 m  0.83
#> 3         5 g  0.6
```

Summary

In this chapter, you learned how to load CSV files with `read_csv()` and to do your own data entry with `tibble()` and `tribble()`. You've learned how CSV files work, some of the problems you might encounter, and how to overcome them. We'll come to data import a few times in this book: Chapter 20 will show you how to load data from Excel and Google Sheets, Chapter 21 from databases, Chapter 22 from parquet files, Chapter 23 from JSON, and Chapter 24 from websites.

We're just about at the end of this section of the book, but there's one important last topic to cover: how to get help. So in the next chapter, you'll learn some good places to look for help, how to create a reprex to maximize your chances of getting good help, and some general advice on keeping up with the world of R.

---

1 The janitor package is not part of the tidyverse, but it offers handy functions for data cleaning and works well within data pipelines that use `|>`.

2 You can override the default of 1,000 with the `guess_max` argument.

Chapter 8. Workflow: Getting Help

This book is not an island; there is no single resource that will allow you to master R. As you begin to apply the techniques described in this book to your own data, you will soon find questions that we do not answer. This section describes a few tips on how to get help and to help you keep learning.

Google Is Your Friend

If you get stuck, start with Google. Typically adding "R" to a query is enough to restrict it to relevant results: if the search isn't useful, it often means that there aren't any R-specific results available. Additionally, adding package names like "tidyverse" or "ggplot2" will help narrow down the results to code that will feel more familiar to you as well, e.g., "how to make a boxplot in R" versus "how to make a boxplot in R with ggplot2." Google is particularly useful for error messages. If you get an error message and you have no idea what it means, try googling it! Chances are that someone else has been confused by it in the past, and there will be help somewhere on the web. (If the error message isn't in English, run `Sys.setenv(LANGUAGE = "en")` and rerun the code; you're more likely to find help for English error messages.)

If Google doesn't help, try Stack Overflow. Start by spending a little time searching for an existing answer, including `[R]`, to restrict your search to questions and answers that use R.

Making a reprex

If your googling doesn't find anything useful, it's a really good idea to prepare a *reprex*, short for minimal *repr*oducible *ex*ample. A good reprex makes it easier for other people to help you, and often you'll figure out the problem yourself in the course of making it. There are two parts to creating a reprex:

- First, you need to make your code reproducible. This means you need to capture everything, i.e., include any `library()` calls and create all necessary objects. The easiest way to make sure you've done this is using the reprex package.
- Second, you need to make it minimal. Strip away everything that is not directly related to your problem. This usually involves creating a much smaller and simpler R object than the one you're facing in real life or even using built-in data.

That sounds like a lot of work! And it can be, but it has a great payoff:

- 80% of the time, creating an excellent reprex reveals the source of your problem. It's amazing how often the process of writing up a self-contained and minimal example allows you to answer your own question.
- The other 20% of the time, you will have captured the essence of your problem in a way that is easy for others to play with. This substantially improves your chances of getting help!

When creating a reprex by hand, it's easy to accidentally miss something, meaning your code can't be run on someone else's computer. Avoid this problem by using the reprex package, which is installed as part of the tidyverse. Let's say you copy this code onto your clipboard (or, on RStudio Server or Cloud, select it):

```
y <- 1:4
mean(y)
```

Then call `reprex()`, where the default output is formatted for GitHub:

```
reprex::reprex()
```

A nicely rendered HTML preview will display in RStudio's Viewer (if you're in RStudio) or your default browser otherwise.

The reprex is automatically copied to your clipboard (on RStudio Server or Cloud, you will need to copy this yourself):

```
``` r
y <- 1:4
mean(y)
#> [1] 2.5
```
```

This text is formatted in a special way, called Markdown, which can be pasted to sites like StackOverflow or GitHub, which will automatically render it to look like code. Here's what that Markdown would look like rendered on GitHub:

```r
y <- 1:4
mean(y)
#> [1] 2.5
```

Anyone else can copy, paste, and run this immediately.

There are three things you need to include to make your example reproducible: required packages, data, and code.

- *Packages* should be loaded at the top of the script so it's easy to see which ones the example needs. This is a good time to check that you're using the latest version of each package; you may have discovered a bug that's been fixed since you installed or last updated the package. For packages in the tidyverse, the easiest way to check is to run `tidyverse_update()`.
- The easiest way to include *data* is to use `dput()` to generate the R code needed to re-create it. For example, to re-create the `mtcars` dataset in R, perform the following steps:
  - Run `dput(mtcars)` in R.
  - Copy the output.
  - In reprex, type `mtcars <-`, and then paste.

  Try to use the smallest subset of your data that still reveals the problem.
- Spend a little bit of time ensuring that your *code* is easy for others to read:
  - Make sure you've used spaces and your variable names are concise yet informative.
  - Use comments to indicate where your problem lies.
  - Do your best to remove everything that is not related to the problem.

  The shorter your code is, the easier it is to understand and the easier it is to fix.

Finish by checking that you have actually made a reproducible example by starting a fresh R session and copying and pasting your script.

Creating reprexes is not trivial, and it will take some practice to learn to create good, truly minimal reprexes. However, learning to ask questions that include the code and investing the time to make it reproducible will continue to pay off as you learn and master R.

Investing in Yourself

You should also spend some time preparing yourself to solve problems before they occur. Investing a little time in learning R each day will pay off handsomely in the long run. One way is to follow what the tidyverse team is doing on the tidyverse blog. To keep up with the R community more broadly, we recommend reading R Weekly: it's a community effort to aggregate the most interesting news in the R community each week.

Summary

This chapter concludes the “Whole Game” part of the book. You’ve now seen the most important parts of the data science process: visualization, transformation, tidying, and importing. Now that you’ve gotten a holistic view of the whole process, we can start to get into the details of small pieces.

The next part of the book, “Visualize,” does a deeper dive into the grammar of graphics and creating data visualizations with ggplot2, showcases how to use the tools you’ve learned so far to conduct exploratory data analysis, and introduces good practices for creating plots for communication.

Part II. Visualize

After reading the first part of the book, you understand (at least superficially) the most important tools for doing data science. Now it's time to start diving into the details. In this part of the book, you'll learn about visualizing data in further depth in Figure II-1.

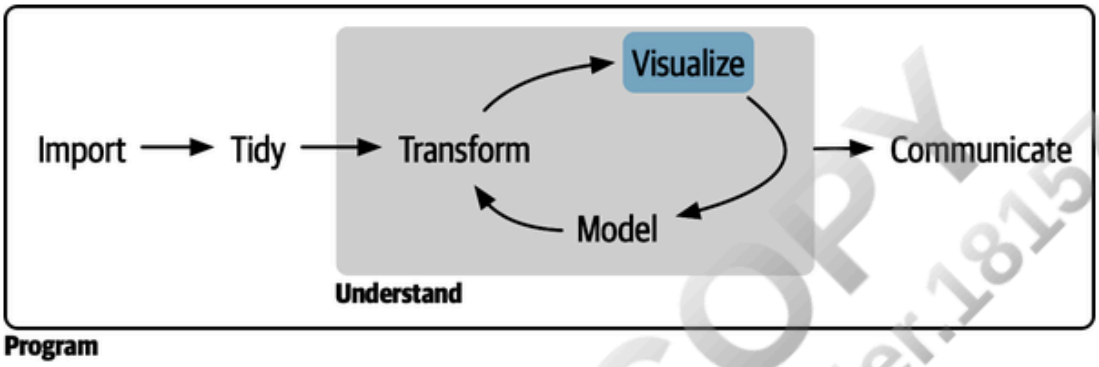


Figure II-1. Data visualization is often the first step in data exploration.

Each chapter addresses one to a few aspects of creating a data visualization:

- In Chapter 9 you will learn about the layered grammar of graphics.
- In Chapter 10, you'll combine visualization with your curiosity and skepticism to ask and answer interesting questions about data.
- Finally, in Chapter 11 you will learn how to take your exploratory graphics, elevate them, and turn them into expository graphics, graphics that help the newcomer to your analysis understand what's going on as quickly and easily as possible.

These three chapters get you started in the world of visualization, but there is much more to learn. The absolute best place to learn more is the ggplot2 book: *ggplot2: Elegant Graphics for Data Analysis* (Springer). It goes into much more depth about the underlying theory and has many more examples of how to combine the individual pieces to solve practical problems. Another great resource is the ggplot2 extensions gallery. This site lists many of the packages that extend ggplot2 with new geoms and scales. It's a great place to start if you're trying to do something that seems hard with ggplot2.