R Basics

Nick Ulle Wesley Brooks

2025-10-29

Table of contents

Overview

This 4-part workshop series provides an introduction to using the R programming language for reproducible data analysis and scientific computing. Topics include programming basics, how to work with tabular data, how to break down programming problems, and how to organize code for clarity and reproducibility.

After this workshop, learners will be able to load tabular data sets into R, compute simple summaries and visualizations, do common data-tidying tasks, write reusable functions, and identify where to go to learn more.

No prior programming experience is necessary. All learners will need access to an internet-connected computer and the latest version of Zoom, R, and RStudio.

1 Getting Started

Learning Goals

After completing this chapter, learners should be able to:

- Run code in the R console
- Call functions and create variables
- Check (in)equality of values
- Describe a file system, directory, and working directory
- Write paths to files or directories
- Get or set the R working directory
- Identify RDS, CSV, TSV files and functions for reading these
- Inspect the structure of a data frame

R is a program for statistical computing. It provides a rich set of built-in tools for cleaning, exploring, modeling, and visualizing data.

The main way you'll interact with R is by writing code or **expressions** in the R programming language. Most people use "R" as a blanket term to refer to both the program and the programming language. Usually, the distinction doesn't matter, but in cases where it does, we'll point it out and be specific.

By writing code, you create an unambiguous record of every step taken in an analysis. This it one of the major advantages of R (and other programming languages) over point-and-click software like Tableau and Microsoft Excel. Code you write in R is **reproducible**: you can share it with someone else, and if they run it with the same inputs, they'll get the same results.

Another advantage of writing code is that it's often **reusable**. This can mean automating a repetitive task within a single analysis, recycling code from one analysis into another, or **packaging** useful code for distribution to the general public. At the time of writing, there were over 17,000 user-contributed packages available for R, spanning a broad range of disciplines.

R is one of many programming languages used in data science. Compared to other programming languages, R's particular strengths are its interactivity, built-in support for handling missing data, the ease with which you can produce high-quality data visualizations, and its broad base of user-contributed packages (due to both its age and growing popularity).

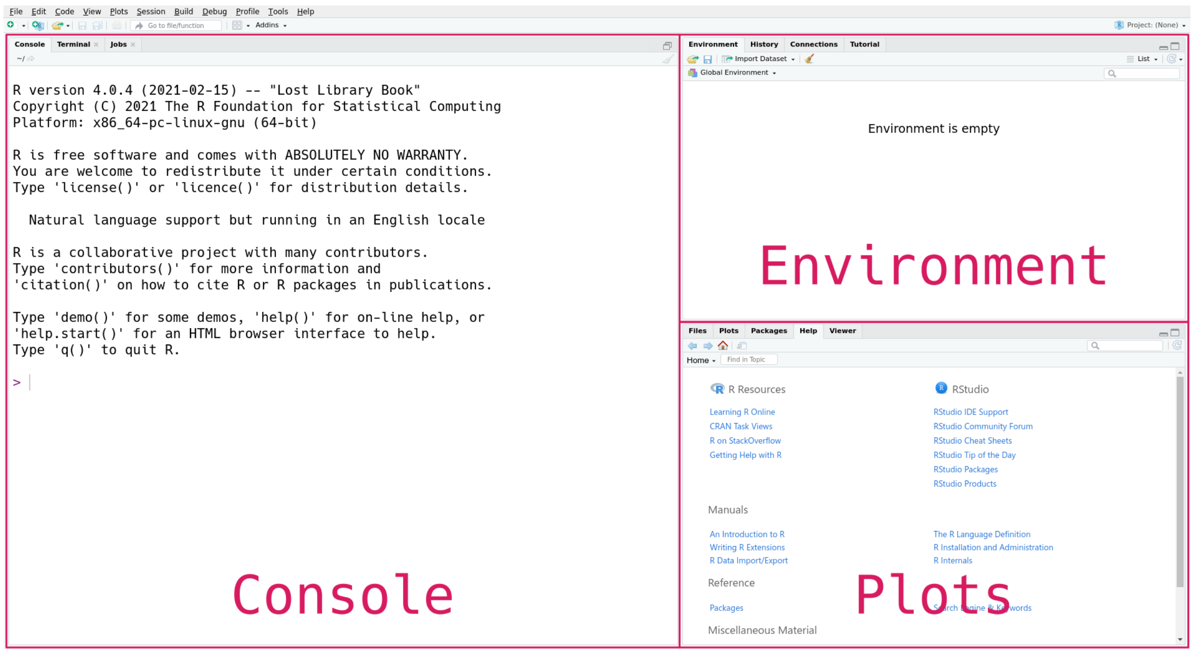
1.1 Prerequisites

You can download R for free here, and can find an install guide here.

In addition to R, you'll need RStudio. RStudio is an **integrated development environment** (IDE), which means it's a comprehensive program for writing, editing, searching, and running code. You can do all of these things without RStudio, but RStudio makes the process easier. You can download RStudio Desktop Open-Source Edition for free here, and can find an install guide here.

1.2 The R Interface

The first time you open RStudio, you'll see a window divided into several panes, like this:



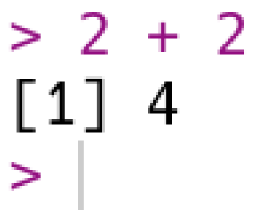
```
R version 4.0.4 (2021-02-15) -- "Lost Library Book"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Figure 1.1: How RStudio typically looks the first time you open it. Don't worry if the text in the panes isn't exactly the same on your computer: it depends on your operating system and versions of R and RStudio.

The console pane, on the left, is the main interface to R. If you type R code into the console and press the `Enter` key on your keyboard, R will run your code and return the result.

On the right are the environment pane and the plots pane. The environment pane shows data in your R workspace. The plots pane shows any plots you make, and also has tabs to browse your file system and to view R's built-in help files. You'll learn more about these gradually, but for now, focus on the console pane.

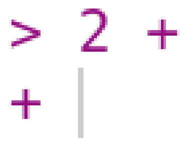
Let's start by using R to do some arithmetic. In the console, you'll see that the cursor is on a line that begins with `>`, called the *prompt*. You can make R compute the sum $2 + 2$ by typing the code `2 + 2` after the prompt and then pressing the `Enter` key. Your code and the result from R should look like this:



```
> 2 + 2
[1] 4
> 
```

R always puts the result on a separate line (or lines) from your code. In this case, the result begins with the tag `[1]`, which is a hint from R that the result is a **vector** and that this line starts with the **element** at position 1. You'll learn more about vectors in Section 2.1, and eventually learn about other data types that are displayed differently. The result of the sum, `4`, is displayed after the tag. In this reader, results from R will usually be typeset in monospace and further prefixed with `##` to indicate that they aren't code.

If you enter an incomplete expression, R will change the prompt to `+`, then wait for you to type the rest of the expression and press the `Enter` key. Here's what it looks like if you only enter `2 +`:



```
> 2 +
+ 
```

You can finish entering the expression, or you can cancel it by pressing the `Esc` key (or `Ctrl-c` if you're using R without RStudio). R can only tell an expression is incomplete if it's missing something, like the second operand in `2 +`. So if you mean to enter `2 + 2` but accidentally enter `2`, which is a complete expression by itself, don't expect R to read your mind and wait for more input!

Try out some other arithmetic in the R console. Besides `+` for addition, the other arithmetic operators are:

- `-` for subtraction

- `*` for multiplication
- `/` for division
- `%%` for remainder division (modulo)
- `^` or `**` for exponentiation

You can combine these and use parentheses to make more complicated expressions, just as you would when writing a mathematical expression. When R computes a result, it follows the standard order of operations: parentheses, exponentiation, multiplication, division, addition, and finally subtraction. For example, to estimate the area of a circle with radius 3, you can write:

```r
3.14 * 3^2
```

```
[1] 28.26
```

You can write R expressions with any number of spaces (including none) around the operators and R will still compute the result. Nevertheless, putting spaces in your code makes it easier for you and others to read, so it's good to make it a habit. Put spaces around most operators, after commas, and after keywords.

1.2.1 Variables

Since R is designed for mathematics and statistics, you might expect that it provides a better appoximation for π than `3.14`. R and most other programming languages allow you to create named values, or **variables**. R provides a built-in variable called `pi` for the value of π. You can display a variable's value by entering its name in the console:

```r
pi
```

```
[1] 3.141593
```

You can also use variables in expressions. For instance, here's a more precise expression for the area of a circle with radius 3:

```r
pi * 3^2
```

```
[1] 28.27433
```

You can define your own variables with the assignment operator `=` or `<-`. In most circumstances these two operators are interchangeable. For clarity, it's best to choose one you like and use it consistently in all of your R code. In this reader, we use `=` for assignment because this is the assignment operator in most programming languages.

The main reason to use variables is to save results so that you can use them on other expressions later. For example, to save the area of the circle in a variable called `area`, you can write:

```r
area = pi * 3^2
```

In R, variable names can contain any combination of letters, numbers, dots `.`, and underscores `_`, but must always start with a letter or a dot. Spaces and other symbols are not allowed in variable names.

Now you can use the `area` variable anywhere you want the computed area. Notice that when you assign a result to a variable, R doesn't automatically display that result. If you want to see the result as well, you have to enter the variable's name as a separate expression:

```r
area
```

```
[1] 28.27433
```

Another reason to use variables is to make an expression more general. For instance, you might want to compute the area of several circles with different radii. Then the expression `pi * 3^2` is too specific. You can rewrite it as `pi * r^2`, and then assign a value to the variable `r` just before you compute the area. Here's the code to compute and display the area of a circle with radius 1 this way:

```r
r = 1
area = pi * r^2
area
```

```
[1] 3.141593
```

Now if you want to compute the area for a different radius, all you have to do is change `r` and run the code again (R will not change `area` until you do this). Writing code that's general enough to reuse across multiple problems can be a big time-saver in the long run. Later on, you'll learn ways to make this code even easier to reuse.

1.2.2 Strings

R treats anything inside single or double quotes as literal text rather than as an expression to evaluate. In programming jargon, a piece of literal text is called a **string**. You can use whichever kind of quotes you prefer, but the quote at the beginning of the string must match the quote at the end.

```
'Hi'
```

```
[1] "Hi"
```

```
"Hello!"
```

```
[1] "Hello!"
```

Numbers and strings are not the same thing, so for example R considers `1` different from `"1"`. As a result, you can't use strings with most of R's arithmetic operators. For instance, this code causes an error:

```
"1" + 3
```

```
Error in "1" + 3: non-numeric argument to binary operator
```

The error message notes that `+` is not defined for non-numeric values.

1.2.3 Comparisons

Besides arithmetic, you can also use R to compare values. The comparison operators are:

- `<` for "less than"
- `>` for "greater than"
- `<=` for "less than or equal to"
- `>=` for "greater than or equal to"
- `==` for "equal to"
- `!=` for "not equal to"

The "equal to" operator uses two equal signs so that R can distinguish it from `=`, the assignment operator.

Let's look at a few examples:

```r
1.5 < 3
```

```
[1] TRUE
```

```r
"a" > "b"
```

```
[1] FALSE
```

```r
pi == 3.14
```

```
[1] FALSE
```

```r
"hi" == 'hi'
```

```
[1] TRUE
```

When you make a comparison, R returns a **logical** value, `TRUE` or `FALSE`, to indicate the result. Logical values are not the same as strings, so they are not quoted.

Logical values are values, so you can use them in other computations. For example:

```r
TRUE
```

```
[1] TRUE
```

```r
TRUE == FALSE
```

```
[1] FALSE
```

Section 2.4.5 describes more ways to use and combine logical values.

Beware that the equality operators don't always return `FALSE` when you compare two different types of data:

```r
"1" == 1
```

```
[1] TRUE
```

```r
"TRUE" <= TRUE
```

```
[1] TRUE
```

```r
"FALSE" <= TRUE
```

```
[1] TRUE
```

Section 2.2.2 explains why this happens, and Section 5.1 explains several other ways to compare values.

1.2.4 Calling Functions

Most of R's features are provided through **functions**, pieces of reusable code. You can think of a function as a machine that takes some inputs and uses them to produce some output. In programming jargon, the inputs to a function are called **arguments**, the output is called the **return value**, and when you use a function, you're **calling** the function.

To call a function, write its name followed by parentheses. Put any arguments to the function inside the parentheses. For example, in R, the sine function is named `sin` (there are also `cos` and `tan`). So you can compute the sine of $\pi/4$ with this code:

```r
sin(pi / 4)
```

```
[1] 0.7071068
```

There are many functions that accept more than one argument. For instance, the `sum` function accepts any number of arguments and adds them all together. When you call a function with multiple arguments, separate the arguments with commas. So another way to compute $2 + 2$ in R is:

```r
sum(2, 2)
```

```
[1] 4
```

When you call a function, R assigns each argument to a **parameter**. Parameters are special variables that represent the inputs to a function and only exist while that function runs. For example, the `log` function, which computes a logarithm, has parameters `x` and `base` for the operand and base of the logarithm, respectively. The next section, Section 1.3, explains how to look up the parameters for a function.

By default, R assigns arguments to parameters based on their order. The first argument is assigned to the function's first parameter, the second to the second, and so on. So you can compute the logarithm of 64, base 2, with this code:

```r
log(64, 2)
```

```
[1] 6
```

The argument 64 is assigned to the parameter `x`, and the argument 2 is assigned to the parameter `base`. You can also assign arguments to parameters by name with `=` (not `<-`), overriding their positions. So some other ways you could write the call above are:

```r
log(64, base = 2)
```

```
[1] 6
```

```r
log(x = 64, base = 2)
```

```
[1] 6
```

```r
log(base = 2, x = 64)
```

```
[1] 6
```

```r
log(base = 2, 64)
```

```
[1] 6
```

All of these are equivalent. When you write code, choose whatever seems the clearest to you. Leaving parameter names out of calls saves typing, but including some or all of them can make the code easier to understand.

Parameters are not regular variables, and only exist while their associated function runs. You can't set them before a call, nor can you access them after a call. So this code causes an error:

```r
x = 64
log(base = 2)
```

```
Error in eval(expr, envir, enclos): argument "x" is missing, with no default
```

In the error message, R says that you forgot to assign an argument to the parameter `x`. You can keep the variable `x` and correct the call by making `x` an argument (for the parameter `x`):

```r
log(x, base = 2)
```

```
[1] 6
```

Or, written more explicitly:

```r
log(x = x, base = 2)
```

```
[1] 6
```

In summary, variables and parameters are distinct, even if they happen to have the same name. The variable `x` is not the same thing as the parameter `x`.

1.3 Getting Help

Learning and using a language is hard, so it's important to know how to get help. The first place to look for help is R's built-in documentation. In the console, you can access a specific help page by name with `?` followed by the name of the page.

There are help pages for all of R's built-in functions, usually with the same name as the function itself. So the code to open the help page for the `log` function is:

```r
?log
```

For functions, help pages usually include a brief description, a list of parameters, a description of the return value, and some examples.

There are also help pages for other topics, such as built-in mathematical constants (such as `?pi`), data sets (such as `?iris`), and operators. To look up the help page for an operator, put the operator's name in single or double quotes. For example, this code opens the help page for the arithmetic operators:

```
?"+"
```

It's always okay to put quotes around the name of the page when you use `?`, but they're only required if it contains non-alphabetic characters. So `?sqrt`, `?'sqrt'`, and `?"sqrt"` all open the documentation for `sqrt`, the square root function.

Sometimes you might not know the name of the help page you want to look up. You can do a general search of R's help pages with `??` followed by a string of search terms. For example, to get a list of all help pages related to linear models:

```
??"linear model"
```

This search function doesn't always work well, and it's often more efficient to use an online search engine. When you search for help with R online, include "R" as a search term. Alternatively, you can use RSeek, which restricts the search to a selection of R-related websites.

1.3.1 When Something Goes Wrong

As a programmer, sooner or later you'll run some code and get an error message or result you didn't expect. Don't panic! Even experienced programmers make mistakes regularly, so learning how to diagnose and fix problems is vital.

Try going through these steps:

1. If R returned a warning or error message, read it! If you're not sure what the message means, try searching for it online.
2. Check your code for typographical errors, including incorrect capitalization and missing or extra commas, quotes, and parentheses.
3. Test your code one line at a time, starting from the beginning. After each line that assigns a variable, check that the value of the variable is what you expect. Try to determine the exact line where the problem originates (which may differ from the line that emits an error!).

If none of these steps help, try asking online. Stack Overflow is a popular question and answer website for programmers. Before posting, make sure to read about how to ask a good question.

1.4 File Systems

Most of the time, you won't just write code directly into the R console. Reproducibility and reusability are important benefits of R over point-and-click software, and in order to realize these, you have to save your code to your computer's hard drive. Let's start by reviewing how files on a computer work. You'll need to understand that before you can save your code, and it will also be important later on for loading data sets.

Your computer's **file system** is a collection of **files** (chunks of data) and **directories** (or "folders") that organize those files. For instance, the file system on a computer shared by Ada and Charles, two pioneers of computing, might look like this:

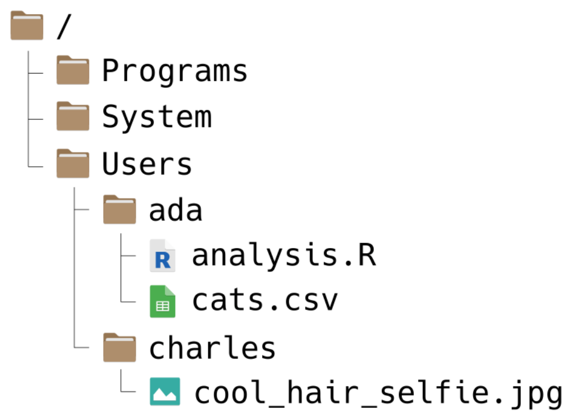


Figure 1.2: A typical file system. Don't worry if your file system looks a bit different from the picture.

File systems have a tree-like structure, with a top-level directory called the **root directory**. On Ada and Charles' computer, the root is called `/`, which is also what it's called on all macOS and Linux computers. On Windows, the root is usually called `C:/`, but sometimes other letters, like `D:/`, are also used depending on the computer's hardware.

A **path** is a list of directories that leads to a specific file or directory on a file system (imagine giving directons to someone as they walk through the file system). Use forward slashes `/` to separate the directories in a path. The root directory includes a forward slash as part of its name, and doesn't need an extra one.

For example, suppose Ada wants to write a path to the file `cats.csv`. She can write the path like this:

```
/Users/ada/cats.csv
```

You can read this path from left-to-right as, "Starting from the root directory, go to the `Users` directory, then from there go to the `ada` directory, and from there go to the file `cats.csv`." Alternatively, you can read the path from right-to-left as, "The file `cats.csv` inside of the `ada` directory, which is inside of the `Users` directory, which is in the root directory."

As another example, suppose Charles wants a path to the `Programs` directory. He can write:

```
/Programs/
```

The `/` at the end of this path is reminder that `Programs` is a directory, not a file. Charles could also write the path like this:

```
/Programs
```

This is still correct, but it's not as obvious that `Programs` is a directory. In other words, when a path leads to a directory, including a **trailing slash** is optional, but makes the meaning of the path clearer. Paths that lead to files never have a trailing slash.

On Windows computers, paths are usually written with backslashes `\` to separate directories instead of forward slashes. Fortunately, R uses forward slashes `/` for all paths, regardless of the operating system. So when you're working in R, use forward slashes and don't worry about the operating system. This is especially convenient when you want to share code with someone that uses a different operating system than you.

1.4.1 Absolute & Relative Paths

A path that starts from the root directory, like all of the ones you've seen so far, is called an **absolute path**. The path is "absolute" because it unambiguously describes where a file or directory is located. The downside is that absolute paths usually don't work well if you share your code.

For example, suppose Ada uses the path `/Programs/ada/cats.csv` to load the `cats.csv` file in her code. If she shares her code with another pioneer of computing, say Gladys, who also has a copy of `cats.csv`, it might not work. Even though Gladys has the file, she might not have it in a directory called `ada`, and might not even have a directory called `ada` on her computer. Because Ada used an absolute path, her code works on her own computer, but isn't portable to others.

On the other hand, a **relative path** is one that doesn't start from the root directory. The path is "relative" to an unspecified starting point, which usually depends on the context.

For instance, suppose Ada's code is saved in the file `analysis.R` (more about `.R` files in Section 1.4.2), which is in the same directory as `cats.csv` on her computer. Then instead of an absolute path, she can use a relative path in her code:

```
cats.csv
```

The context is the location of `analysis.R`, the file that contains the code. In other words, the starting point on Ada's computer is the `ada` directory. On other computers, the starting point will be different, depending on where the code is stored.

Now suppose Ada sends her corrected code in `analysis.R` to Gladys, and tells Gladys to put it in the same directory as `cats.csv`. Since the path `cats.csv` is relative, the code will still work on Gladys' computer, as long as the two files are in the same directory. The name of that directory and its location in the file system don't matter, and don't have to be the same as on Ada's computer. Gladys can put the files in a directory `/Users/gladys/from_ada/` and the path (and code) will still work.

Relative paths can include directories. For example, suppose that Charles wants to write a relative path from the `Users` directory to a cool selfie he took. Then he can write:

```
charles/cool_hair_selfie.jpg
```

You can read this path as, "Starting from wherever you are, go to the `charles` directory, and from there go to the `cool_hair_selfie.jpg` file." In other words, the relative path depends on the context of the code or program that uses it.

When use you paths in R code, they should almost always be relative paths. This ensures that the code is portable to other computers, which is an important aspect of reproducibility. Another benefit is that relative paths tend to be shorter, making your code easier to read (and write).

When you write paths, there are three shortcuts you can use. These are most useful in relative paths, but also work in absolute paths:

- `.` means the current directory.
- `..` means the directory above the current directory.
- `~` means the **home directory**. Each user has their own home directory, whose location depends on the operating system and their username. Home directories are typically found inside `C:/Users/` on Windows, `/Users/` on macOS, and `/home/` on Linux.

As an example, suppose Ada wants to write a (relative) path from the `ada` directory to Charles' cool selfie. Using these shorcuts, she can write:

```
../charles/cool_hair_selfie.jpg
```

Read this as, "Starting from wherever you are, go up one directory, then go to the `charles` directory, and then go to the `cool_hair_selfie.jpg` file." Since `/Users/ada` is Ada's home directory, she could also write the path as:

```
~/../charles/cool_hair_selfie.jpg
```

This path has the same effect, but the meaning is slightly different. You can read it as "Starting from your home directory, go up one directory, then go to the `charles` directory, and then go to the `cool_hair_selfie.jpg` file."

The `..` and `~` shortcut are frequently used and worth remembering. The `.` shortcut is included here in case you see it in someone else's code. Since it means the current directory, a path like `./cats.csv` is identical to `cats.csv`, and the latter is preferable for being simpler. There are a few specific situations where `.` is necessary, but they fall outside the scope of this text.

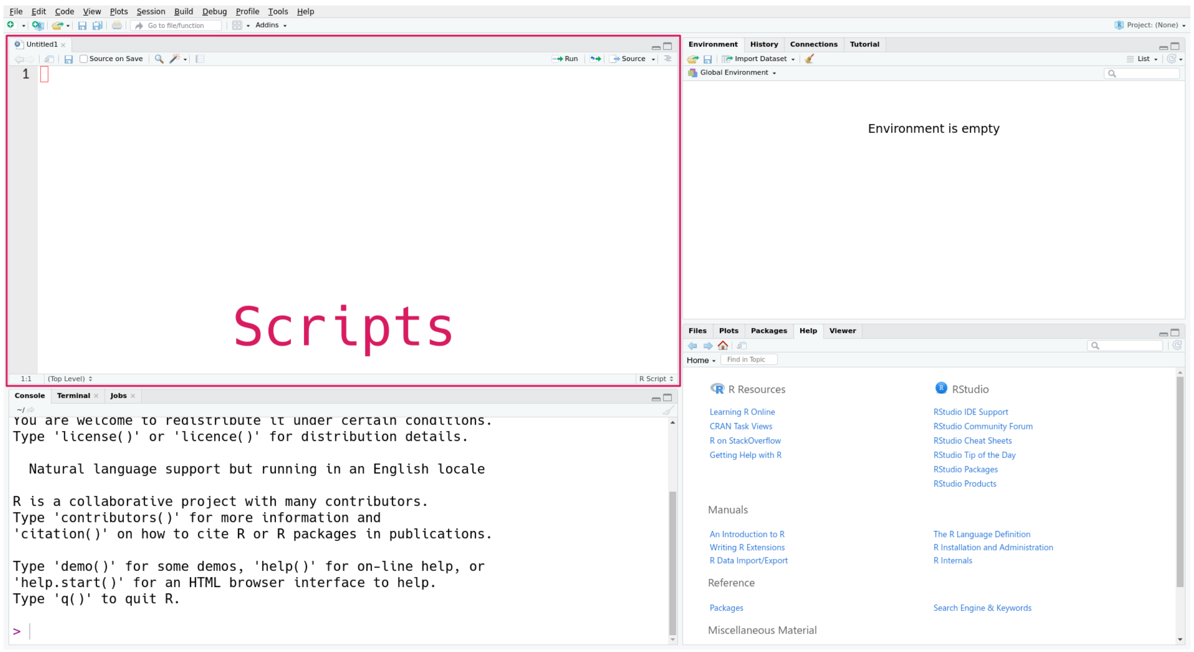
1.4.2 R Scripts

Now that you know how file systems and paths work, you're ready to learn how to save your R code to a file. R code is usually saved into an **R script** (extension `.R`) or an **R Markdown file** (extension `.Rmd`). R scripts are slightly simpler, so let's focus on those.

In RStudio, you can create a new R script with this menu option:

```
File -> New File -> R Script
```

This will open a new pane in RStudio, like this:



```
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 1.3: How RStudio typically looks after opening a new R Script.

The new pane is the scripts pane, which displays all of the R scripts you're editing. Each script appears in a separate tab. In the screenshot, only one script, the new script, is open.

Editing a script is similar to editing any other text document. You can write, delete, copy, cut, and paste text. You can also save the file to your computer's file system. When you do, pay attention to where you save the file, as you might need it later.

The contents of an R script should be R code. Anything else you want to write in the script (notes, documentation, etc.) should be in a **comment**. In R, comments begin with `#` and extend to the end of the line:

```
# This is a comment.
```

R will ignore comments when you run your code.

When you start a new project, it's a good idea to create a specific directory for all of the project's files. If you're using R, you should also create one or more R scripts in that directory. As you work, write your code directly into a script. Arrange your code in the order of the steps to solve the problem, even if you write some parts before others. Comment out or delete any lines of code that you try but ultimately decide you don't need. Make sure to save the file periodically so that you don't lose your work. Following these guidelines will help you stay organized and make it easier to share your code with others later.

While editing, you can run the current line in the R console by pressing `Ctrl+Enter` on Windows and Linux, or `Cmd+Enter` on macOS. This way you can test and correct your code as you write it.

If you want, you can instead run (or **source**) the entire R script, by calling the `source` function with the path to the script as an argument. This is also what the "Source on Save" check box refers to in RStudio. The code runs in order, only stopping if an error occurs.

For instance, if you save the script as `my_cool_script.R`, then you can run `source("my_cool_script.R")` in the console to run the entire script (pay attention to the path—it may be different on your computer).

R Markdown files are an alternative format for storing R code. They provide a richer set of formatting options, and are usually a better choice than R scripts if you're writing a report that contains code. You can learn more about R Markdown files here.

1.4.3 The Working Directory

Section 1.4.1 explained that relative paths have a starting point that depends on the context where the path is used. Let's make that idea more concrete for R. The **working directory** is the starting point R uses for relative paths. Think of the working directory as the directory R is currently "at" or watching.

The function `getwd` returns the absolute path for the current working directory, as a string. It doesn't require any arguments:

```r
getwd()
```

```
[1] "/home/nick/mill/datalab/teaching/r_basics"
```

On your computer, the output from `getwd` will likely be different. This is a very useful function for getting your bearings when you write relative paths. If you write a relative path and it doesn't work as expected, the first thing to do is check the working directory.

The related `setwd` function changes the working directory. It takes one argument: a path to the new working directory. Here's an example:

```r
setwd("..")

# Now check the working directory.
getwd()
```

Generally, you should avoid using calls to `setwd` in your R scripts and R Markdown files. Calling `setwd` makes your code more difficult to understand, and can always be avoided by using appropriate relative paths. If you call `setwd` with an absolute path, it also makes your code less portable to other computers. It's fine to use `setwd` interactively (in the R console), but avoid making your saved code dependent on it.

Another function that's useful for dealing with the working directory and file system is `list.files`. The `list.files` function returns the names of all of the files and directories inside of a directory. It accepts a path to a directory as an argument, or assumes the working directory if you don't pass a path. For instance:

```r
# List files and directories in /home/.
list.files("/home/")
```

```
[1] "lost+found" "nick"
```

```r
# List files and directories in the working directory.
list.files()
```

```
 [1] "_freeze"      "_quarto.yml" "assessment"   "chapters"     "cover.png"
 [6] "data"         "docs"         "images"       "index.html"   "index.qmd"
[11] "LICENSE"      "notes"        "R"            "raw"          "README.md"
[16] "site_libs"
```

As usual, since you have a different computer, you're likely to see different output if you run this code. If you call `list.files` with an invalid path or an empty directory, the output is `character(0)`:

```r
list.files("/this/path/is/fake/")
```

```
character(0)
```

Later on, you'll learn about what `character(0)` means more generally.

1.5 Reading Files

Analyzing data sets is one of the most common things to do in R. The first step is to get R to read your data. Data sets come in a variety of file formats, and you need to identify the format in order to tell R how to read the data.

Most of the time, you can guess the format of a file by looking at its **extension**, the characters (usually three) after the last dot `.` in the filename. For example, the extension `.jpg` or `.jpeg` indicates a JPEG image file. Some operating systems hide extensions by default, but you can find instructions to change this setting online by searching for "show file extensions" and your operating system's name. The extension is just part of the file's name, so it should be taken as a hint about the file's format rather than a guarantee.

R has built-in functions for reading a variety of formats. The R community also provides **packages**, shareable and reusable pieces of code, to read even more formats. You'll learn more about packages later, in Section 3.2. For now, let's focus on data sets that can be read with R's built-in functions.

Here are several formats that are frequently used to distribute data, along with the name of a built-in function or contributed package that can read the format:

Name	Extension	Function or Package	Tabular?	Text?
Comma-separated Values	`.csv`	`read.csv`	Yes	Yes
Tab-separated Values	`.tsv`	`read.delim`	Yes	Yes
Fixed-width File	`.fwf`	`read.fwf`	Yes	Yes
Microsoft Excel	`.xlsx`	readr package	Yes	No
Microsoft Excel 1993-2007	`.xls`	readr package	Yes	No
Apache Arrow	`.feather`	arrow package	Yes	No
R Data	`.rds`	`readRDS`	Sometimes	No
R Data	`.rda`	`load`	Sometimes	No
Plaintext	`.txt`	`readLines`	Sometimes	Yes

Name	Extension	Function or Package	Tabular?	Text?
Extensible Markup Language	`.xml`	xml2 package	No	Yes
JavaScript Object Notation	`.json`	jsonlite package	No	Yes

A **tabular** data set is one that's structured as a table, with rows and columns. This reader focuses on tabular data sets, since they're common in practice and present the fewest programming challenges. Here's an example of a tabular data set:

Fruit	Quantity	Price
apple	32	1.49
banana	541	0.79
pear	10	1.99

A **text file** is a file that contains human-readable lines of text. You can check this by opening the file with a text editor such as Microsoft Notepad or macOS TextEdit. Many file formats use text in order to make the format easier to work with.

For instance, a **comma-separated values** (CSV) file records a tabular data using one line per row, with commas separating columns. If you store the table above in a CSV file and open the file in a text editor, here's what you'll see:

```
Fruit,Quantity,Price
apple,32,1.49
banana,541,0.79
pear,10,1.99
```

A **binary file** is one that's not human-readable. You can't just read off the data if you open a binary file in a text editor, but they have a number of other advantages. Compared to text files, binary files are often faster to read and take up less storage space (bytes).

As an example, R's built-in binary format is called RDS (which may stand for "R data serialized"). RDS files are extremely useful for backing up work, since they can store any kind of R object, even ones that are not tabular. You can learn more about how to create an RDS file on the `?saveRDS` help page, and how to read one on the `?readRDS` help page.

1.5.1 Hello, Data!

Let's read our first data set! Over the next few sections, you're going to explore a data set about the people who are depicted on banknotes (paper money) from around the world. Click here to download the data set (you'll need to click the "Download raw file" button).

The banknotes data set is derived from one created by The Pudding, a digital publication that makes awesome stories and visualizations with data. Check out their article about banknotes around the world.

The data set is in a file called `banknotes.csv`, which suggests it's a CSV file. In this case, the extension is correct, so you can read the file into R with the built-in `read.csv` function. The first argument is the path to where you saved the file, which may be different on your computer. The `read.csv` function returns the data set, but R won't keep the data in memory unless you assign the returned result to a variable:

```r
banknotes = read.csv("data/banknotes.csv")
```

The variable name `banknotes` here is arbitrary; you can choose something different if you want. However, in general, it's a good habit to choose variable names that describe the contents of the variable somehow.

If you tried running the line of code above and got an error message, pay attention to what the error message says, and remember the strategies to get help in Section 1.3. The most common mistake when reading a file is incorrectly specifying the path, so first check that you got the path right.

If you ran the line of code and there was no error message, congratulations, you've read your first data set into R!

1.6 Data Frames

Now that you've loaded the data, let's take a look at it. When you're working with a new data set, it's usually not a good idea to print it out directly (by typing `banknotes`, in this case) until you know how big it is. Big data sets can take a long time to print, and the output can be difficult to read.

Instead, you can use the `head` function to print only the beginning, or head, of a data set. Let's take a peek:

```r
head(banknotes)
```

```
  currency_code   country     currency_name                       name gender
1           ARS Argentina  Argentinian Peso                  Eva Perón      F
2           ARS Argentina  Argentinian Peso       Julio Argentino Roca      M
3           ARS Argentina  Argentinian Peso Domingo Faustino Sarmiento      M
4           ARS Argentina  Argentinian Peso       Juan Manuel de Rosas      M
5           ARS Argentina  Argentinian Peso            Manuel Belgrano      M
6           AUD Australia Australian Dollar              David Unaipon      M
  bill_count    profession known_for_being_first current_bill_value
1        1.0      Activist                    No                100
2        1.0 Head of Gov't                    No                100
3        1.0 Head of Gov't                    No                 50
4        1.0    Politician                    No                 20
5        1.0       Founder                   Yes                 10
6        0.5          STEM                   Yes                 50
  prop_total_bills first_appearance_year death_year
1               NA                  2012       1952
2               NA                  1988       1914
3               NA                  1999       1888
4               NA                  1992       1877
5               NA                  1970       1820
6             0.48                  1995       1967

1
2
3
4
5
6 Shares with another person. In 1927 when his book of Aboriginal legends, Hurgarrda was publ
                                        hover_text has_portrait            id
1                                                          true     ARS_Evita
2                                                          true ARS_Argentino
3                                                          true   ARS_Domingo
4                                                          true     ARS_Rosas
5                     Designed first Argentine flag         true  ARS_Belgrano
6 First Australian Aboriginal writer to be published         true   AUD_Unaipon
  scaled_bill_value
1         1.0000000
2         1.0000000
3         0.4444444
4         0.1111111
5         0.0000000
6         0.4736842
```

This data set is tabular—as you might have already guessed, since it came from a CSV file. In R, it's represented by a **data frame**, a table with rows and columns. R uses data frames to represent most (but not all) kinds of tabular data. The `read.csv` function, which you used to read this data, always returns a data frame.

For a data frame, the `head` function only prints the first six rows. If there are lots of columns or the columns are wide, as is the case here, R wraps the output across lines.

When you first read an object into R, you might not know whether it's a data frame. One way to check is visually, by printing it (as you just did with `head`). A better way to check is with the `class` function, which returns information about what an object is. For a data frame, the result will always contain `data.frame`:

```r
class(banknotes)
```

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

You'll learn more about classes in Section 2.2, but for now you can use this function to identify data frames.

By counting the columns in the output from `head(banknotes)`, you can see that this data set has 17 columns. A more convenient way to check the number of columns in a data set is with the `ncol` function:

```r
ncol(banknotes)
```

```
[1] 17
```

The similarly-named `nrow` function returns the number of rows:

```r
nrow(banknotes)
```

```
[1] 279
```

Alternatively, you can get both numbers at the same time with the `dim` (short for "dimensions") function.

Since the columns have names, you might also want to get just these. You can do that with the `names` or `colnames` functions. Both return the same result:

```r
names(banknotes)
```

```
 [1] "currency_code"         "country"               "currency_name"
 [4] "name"                  "gender"                "bill_count"
 [7] "profession"            "known_for_being_first" "current_bill_value"
[10] "prop_total_bills"      "first_appearance_year" "death_year"
[13] "comments"              "hover_text"            "has_portrait"
[16] "id"                    "scaled_bill_value"
```

```r
colnames(banknotes)
```

```
 [1] "currency_code"         "country"               "currency_name"
 [4] "name"                  "gender"                "bill_count"
 [7] "profession"            "known_for_being_first" "current_bill_value"
[10] "prop_total_bills"      "first_appearance_year" "death_year"
[13] "comments"              "hover_text"            "has_portrait"
[16] "id"                    "scaled_bill_value"
```

If the rows have names, you can get those with the `rownames` function. For this particular data set, the rows don't have names.

1.6.1 Summarizing Data

An efficient way to get a sense of what's actually in a data set is to have R compute summary information. This works especially well for data frames, but also applies to other data. R provides two different functions to get summaries: `str` and `summary`.

The `str` function returns a **structural summary** of an object. This kind of summary tells us about the structure of the data—the number of rows, the number and names of columns, what kind of data is in each column, and some sample values. Here's the structural summary for the banknotes data:

```r
str(banknotes)
```

```
'data.frame':	279 obs. of  17 variables:
 $ currency_code        : chr  "ARS" "ARS" "ARS" "ARS" ...
 $ country              : chr  "Argentina" "Argentina" "Argentina" "Argentina" ...
 $ currency_name        : chr  "Argentinian Peso" "Argentinian Peso" "Argentinian Peso" "Arge
 $ name                 : chr  "Eva Perón" "Julio Argentino Roca" "Domingo Faustino Sarmiento
 $ gender               : chr  "F" "M" "M" "M" ...
 $ bill_count           : num  1 1 1 1 1 0.5 0.5 0.5 0.5 0.5 ...
 $ profession           : chr  "Activist" "Head of Gov't" "Head of Gov't" "Politician" ...
 $ known_for_being_first: chr  "No" "No" "No" "No" ...
 $ current_bill_value   : int  100 100 50 20 10 50 10 20 10 50 ...
 $ prop_total_bills     : num  NA NA NA NA NA 0.48 0.08 0.1 0.08 0.48 ...
 $ first_appearance_year: int  2012 1988 1999 1992 1970 1995 1993 1994 1993 1995 ...
 $ death_year           : chr  "1952" "1914" "1888" "1877" ...
 $ comments             : chr  "" "" "" "" ...
 $ hover_text           : chr  "" "" "" "" ...
 $ has_portrait         : chr  "true" "true" "true" "true" ...
 $ id                   : chr  "ARS_Evita" "ARS_Argentino" "ARS_Domingo" "ARS_Rosas" ...
 $ scaled_bill_value    : num  1 1 0.444 0.111 0 ...
```

This summary lists information about each column, and includes most of what you found earlier by using several different functions separately. The summary uses `chr` to indicate columns of text ("characters") and `int` to indicate columns of integers.

In contrast to `str`, the `summary` function returns a **statistical summary** of an object. This summary includes summary statistics for each column, choosing appropriate statistics based on the kind of data in the column. For numbers, this is generally the mean, median, and quantiles. For categories, this is the frequencies. Other kinds of statistics are shown for other kinds of data. Here's the statistical summary for the banknotes data:

```
summary(banknotes)
```

```
 currency_code         country          currency_name          name          
 Length:279         Length:279         Length:279         Length:279        
 Class :character   Class :character   Class :character   Class :character  
 Mode  :character   Mode  :character   Mode  :character   Mode  :character  
                                                                              
                                                                              
                                                                              
    gender            bill_count      profession        known_for_being_first
 Length:279         Min.   :0.2500   Length:279         Length:279           
 Class :character   1st Qu.:0.5000   Class :character   Class :character     
 Mode  :character   Median :1.0000   Mode  :character   Mode  :character     
                    Mean   :0.8456                                           
                    3rd Qu.:1.0000                                           
                    Max.   :1.0000                                           
 current_bill_value prop_total_bills first_appearance_year  death_year        
 Min.   :     1      Min.   :0.0100   Min.   :1869          Length:279
 1st Qu.:    20      1st Qu.:0.0550   1st Qu.:1980          Class :character
 Median :   100      Median :0.1000   Median :1996          Mode  :character
 Mean   :  4039      Mean   :0.1669   Mean   :1992
 3rd Qu.:  1000      3rd Qu.:0.2300   3rd Qu.:2012
 Max.   :100000      Max.   :0.7500   Max.   :2021
                     NA's   :220
   comments           hover_text        has_portrait            id
 Length:279         Length:279         Length:279         Length:279
 Class :character   Class :character   Class :character   Class :character
 Mode  :character   Mode  :character   Mode  :character   Mode  :character



 scaled_bill_value
 Min.   :0.00000
 1st Qu.:0.01828
 Median :0.11111
 Mean   :0.30606
 3rd Qu.:0.48914
 Max.   :1.00000
 NA's   :1
```

1.6.2 Selecting Columns

You can select an individual column from a data frame by name with `$`, the dollar sign operator. The syntax is:

```
VARIABLE$COLUMN_NAME
```

For instance, for the banknotes data, `banknotes$country` selects the `country` column, and `banknotes$first_appearance_year` selects the `first_appearance_year` column. So one way to compute the mean of the `first_appearance_year` column is:

```
mean(banknotes$first_appearance_year)
```

```
[1] 1992.319
```

Similarly, to compute the range of the `current_bill_value` column:

```r
range(banknotes$current_bill_value)
```

```
[1]      1 100000
```

You can also use the dollar sign operator to assign values to columns. For instance, to assign `USD` to the entire `currency_code` column:

```r
banknotes$currency_code = "USD"
```

Be careful when you do this, as there is no undo. Fortunately, you haven't saved any transformations to the banknotes data to your computer's hard drive yet, so you can reset the `banknotes` variable back to what it was by reloading the data set:

```r
banknotes = read.csv("data/banknotes.csv")
```

In Section 2.4, you'll learn how to select rows and individual elements from a data frame, as well as other ways to select columns.

1.7 Exercises

1.7.1 Exercise

In a string, an **escape sequence** or escape code consists of a backslash followed by one or more characters. Escape sequences make it possible to:

- Write quotes or backslashes within a string
- Write characters that don't appear on your keyboard (for example, characters in a foreign language)

For example, the escape sequence `\n` corresponds to the newline character. There's a complete list of escape sequences for R in the `?Quotes` help file. Other programming languages also use escape sequences, and many of them are the same as in R.

1. Assign a string that contains a newline to the variable `newline`. Then make R display the value of the variable by entering `newline` at the R prompt.
2. The `message` function prints output to the R console, so it's one way you can make your R code report information as it runs. Use the `message` function to print `newline`.
3. How does the output from part 1 compare to the output from part 2? Why do you think they differ?

1.7.2 Exercise

1. Choose a directory on your computer that you're familiar with, such as one you created. Determine the path to the directory, then use `list.files` to display its contents. Do the files displayed match what you see in your system's file browser?
2. What does the `all.files` parameter of `list.files` do? Give an example.

1.7.3 Exercise

The `read.table` function is another function for reading tabular data. Take a look at the help file for `read.table`. Recall that `read.csv` reads tabular data where the values are separated by commas, and `read.delim` reads tabular data where the values are separated by tabs.

1. What value-separator does `read.table` expect by default?
2. Is it possible to use `read.table` to read a CSV? Explain. If your answer is yes, show how to use `read.table` to load the banknotes data from Section 1.5.1.

2 Data Structures

ℹ Learning Goals

After completing this chapter, learners should be able to:

- Create vectors, including sequences
- Identify whether a function is vectorized or not
- Check the type and class of an object
- Coerce an object to a different type
- Describe matrices and lists
- Describe and differentiate `NA`, `NaN`, `Inf`, `NULL`
- Identify, create, and relevel factors
- Index vectors with empty, integer, string, and logical arguments
- Negate or combine conditions with logic operators

The previous chapter introduced R and gave you enough background to do some simple computations on data sets. This chapter focuses on the foundational knowledge and skills you'll need in order to use R effectively in the long term. Specifically, it begins with a deep dive into R's various data structures and data types, then explains a variety of ways to get and set their elements.

2.1 Vectors

A **vector** is a collection of values. Vectors are the fundamental unit of data in R, and you've already used them in the previous sections.

For instance, each column in a data frame is a vector. So the `first_appearance_year` column in the banknotes data from Section 1.6 is a vector. Take a look at it now. You can use `head` to avoid printing too much. Set the second argument to `10` so that exactly 10 values are printed:

```
head(banknotes$first_appearance_year, 10)
```

```
 [1] 2012 1988 1999 1992 1970 1995 1993 1994 1993 1995
```

Like all vectors, this vector is **ordered**, which just means the values, or **elements**, have specific positions. The value of the 1st element is `2012`, the 2nd is `1988`, the 5th is `1970`, and so on.

Notice that the elements of this vector are all integers. In R, the elements of a vector must all be the same type of data (we say the elements are **homogeneous**). A vector can contain integers, decimal numbers, strings, or several other types of data, but not a mix these all at once.

The other columns in the banknotes data frame are also vectors. For instance, the `name` column is a vector of strings:

```r
head(banknotes$name)
```

```
[1] "Eva Perón"                   "Julio Argentino Roca"
[3] "Domingo Faustino Sarmiento" "Juan Manuel de Rosas"
[5] "Manuel Belgrano"             "David Unaipon"
```

Vectors can contain any number of elements, including 0 or 1 element. Unlike mathematics, R does not distinguish between vectors and **scalars** (solitary values). So as far as R is concerned, a solitary value, like `3`, is a vector with 1 element.

You can check the length of a vector (and other objects) with the `length` function:

```r
length(3)
```

```
[1] 1
```

```r
length("hello")
```

```
[1] 1
```

```r
length(banknotes$first_appearance_year)
```

```
[1] 279
```

Since the last of these is a column from the data frame `banknotes`, the length is the same as the number of rows in `banknotes`.

2.1.1 Creating Vectors

Sometimes you'll want to create your own vectors. You can do this by concatenating several vectors together with the `c` function. It accepts any number of vector arguments, and combines them into a single vector:

```r
c(1, 2, 19, -3)
```

```
[1]  1  2 19 -3
```

```r
c("hi", "hello")
```

```
[1] "hi"    "hello"
```

```r
c(1, 2, c(3, 4))
```

```
[1] 1 2 3 4
```

If the arguments you pass to the `c` function have different data types, R will attempt to convert them to a common data type that preserves the information:

```r
c(1, "cool", 2.3)
```

```
[1] "1"    "cool" "2.3"
```

Section 2.2.2 explains the rules for this conversion in more detail.

The colon operator `:` creates vectors that contain sequences of integers. This is useful for creating "toy" data to test things on, and later we'll see that it's also important in several other contexts. Here are a few different sequences:

```r
1:3
```

```
[1] 1 2 3
```

```r
-3:5
```

```
[1] -3 -2 -1  0  1  2  3  4  5
```

```r
10:1
```

```
 [1] 10  9  8  7  6  5  4  3  2  1
```

Beware that both endpoints are included in the sequence, even in sequences like `1:0`, and that the difference between elements is always `-1` or `1`. If you want more control over the generated sequence, use the `seq` function instead.

2.1.2 Indexing Vectors

You can access individual elements of a vector with the **indexing operator** `[` (also called the square bracket operator). The syntax is:

```
VECTOR[INDEXES]
```

Here `INDEXES` is a vector of positions of elements you want to get or set.

For example, let's make a vector with 5 elements and get the 2nd element:

```r
x = c(4, 8, 3, 2, 1)
x[2]
```

```
[1] 8
```

Now let's get the 3rd and 1st element:

```r
x[c(3, 1)]
```

```
[1] 3 4
```

You can use the indexing operator together with the assignment operator to assign elements of a vector:

```r
x[3] = 0
x
```

```
[1] 4 8 0 2 1
```

Indexing is among the most frequently used operations in R, so take some time to try it out with few different vectors and indexes. We'll revisit indexing in Section 2.4 to learn a lot more about it.

2.1.3 Vectorization

Let's look at what happens if we call a mathematical function, like `sin`, on a vector:

```r
x = c(1, 3, 0, pi)
sin(x)
```

```
[1] 8.414710e-01 1.411200e-01 0.000000e+00 1.224647e-16
```

This gives us the same result as if we had called the function separately on each element. That is, the result is the same as:

```r
c(sin(1), sin(3), sin(0), sin(pi))
```

```
[1] 8.414710e-01 1.411200e-01 0.000000e+00 1.224647e-16
```

Of course, the first version is much easier to type.

Functions that take a vector argument and get applied element-by-element like this are said to be **vectorized**. Most functions in R are vectorized, especially math functions. Some examples include `sin`, `cos`, `tan`, `log`, `exp`, and `sqrt`.

Functions that are not vectorized tend to be ones that combine or aggregate values in some way. For instance, the `sum`, `mean`, `median`, `length`, and `class` functions are not vectorized.

A function can be vectorized across multiple arguments. This is easiest to understand in terms of the arithmetic operators. Let's see what happens if we add two vectors together:

```r
x = c(1, 2, 3, 4)
y = c(-1, 7, 10, -10)
x + y
```

```
[1]  0  9 13 -6
```

The elements are paired up and added according to their positions. The other arithmetic operators are also vectorized:

```r
x - y
```

```
[1]  2 -5 -7 14
```

```r
x * y
```

```
[1]  -1  14  30 -40
```

```r
x / y
```

```
[1] -1.0000000  0.2857143  0.3000000 -0.4000000
```

2.1.4 Recycling

When a function is vectorized across multiple arguments, what happens if the vectors have different lengths? Whenever you think of a question like this as you're learning R, the best way to find out is to create some toy data and test it yourself. Let's try that now:

```r
x = c(1, 2, 3, 4)
y = c(-1, 1)
x + y
```

```
[1] 0 3 2 5
```

The elements of the shorter vector are **recycled** to match the length of the longer vector. That is, after the second element, the elements of `y` are repeated to make a vector with the same length as `x` (because `x` is longer), and then vectorized addition is carried out as usual.

Here's what that looks like written down:

```
   1  2  3  4
+ -1  1 -1  1
  -----------
   0  3  2  5
```

If the length of the longer vector is not a multiple of the length of the shorter vector, R issues a warning, but still returns the result. The warning as meant as a reminder, because unintended recycling is a common source of bugs:

```r
x = c(1, 2, 3, 4, 5)
y = c(-1, 1)
x + y
```

```
Warning in x + y: longer object length is not a multiple of shorter object
length
```

```
[1] 0 3 2 5 4
```

Recycling might seem strange at first, but it's convenient if you want to use a specific value (or pattern of values) with a vector. For instance, suppose you want to multiply all the elements of a vector by `2`. Recycling makes this easy:

```r
2 * c(1, 2, 3)
```

```
[1] 2 4 6
```

When you use recycling, most of the time one of the arguments will be a scalar like this.

2.2 Data Types & Classes

Data can be categorized into different **types** based on sets of shared characteristics. For instance, statisticians tend to think about whether data are numeric or categorical:

- numeric
  - continuous (real or complex numbers)
  - discrete (integers)
- categorical
  - nominal (categories with no ordering)
  - ordinal (categories with some ordering)

Of course, other types of data, like graphs (networks) and natural language (books, speech, and so on), are also possible. Categorizing data this way is useful for reasoning about which methods to apply to which data.

In R, data objects are categorized in two different ways:

1. The **class** of an R object describes what the object does, or the role that it plays. Sometimes objects can do more than one thing, so objects can have more than one class. The `class` function, which debuted in Section 1.6, returns the classes of its argument.
2. The **type** of an R object describes what the object is. Technically, the type corresponds to how the object is stored in your computer's memory. Each object has exactly one type. The `typeof` function returns the type of its argument.

Of the two, classes tend to be more important than types. If you aren't sure what an object is, checking its classes should be the first thing you do.

The built-in classes you'll use all the time correspond to vectors and lists (which we'll learn more about in Section 2.2.1):

Class	Example	Description
logical	`TRUE`, `FALSE`	Logical (or Boolean) values
integer	`-1L`, `1L`, `2L`	Integer numbers
numeric	`-2.1`, `7`, `34.2`	Real numbers
complex	`3-2i`, `-8+0i`	Complex numbers
character	`"hi"`, `"YAY"`	Text strings
list	`list(TRUE, 1, "hi")`	Ordered collection of heterogeneous elements

R doesn't distinguish between scalars and vectors, so the class of a vector is the same as the class of its elements:

```r
class("hi")
```

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

```r
class(c("hello", "hi"))
```

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

In addition, for most vectors, the class and the type are the same:

```r
x = c(TRUE, FALSE)
class(x)
```

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

```r
typeof(x)
```

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

The exception to this rule is numeric vectors, which have type `double` for historical reasons:

```r
class(pi)
```

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

```r
typeof(pi)
```

```
[1] "double"
```

```r
typeof(3)
```

```
[1] "double"
```

The word "double" here stands for double-precision floating point number, a standard way to represent real numbers on computers.

By default, R assumes any numbers you enter in code are numeric, even if they're integer-valued.

The class `integer` also represents integer numbers, but it's not used as often as `numeric`. A few functions, such as the sequence operator `:` and the `length` function, return integers. You can also force R to create an integer by adding the suffix `L` to a number, but there are no major drawbacks to using the `double` default:

```r
class(1:3)
```

```
[1] "integer"
```

```r
class(3)
```

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

```r
class(3L)
```

```
[1] "integer"
```

Besides the classes for vectors and lists, there are several built-in classes that represent more sophisticated data structures:

Class	Description
function	Functions
factor	Categorical values
matrix	Two-dimensional ordered collection of homogeneous elements
array	Multi-dimensional ordered collection of homogeneous elements
data.frame	Data frames

For these, the class is usually different from the type. We'll learn more about most of these later on.

2.2.1 Lists

A **list** is an ordered data structure where the elements can have different types (they are **heterogeneous**). This differs from a vector, where the elements all have to have the same type, as we saw in Section 2.1. The tradeoff is that most vectorized functions do not work with lists.

You can make an ordinary list with the `list` function:

```r
x = list(1, c("hi", "bye"))
class(x)
```

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

```r
typeof(x)
```

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

For ordinary lists, the type and the class are both `list`. In Section 2.4, we'll learn how to get and set list elements, and in later sections we'll learn more about when and why to use lists.

You've already seen one list, the banknotes data frame:

```r
class(banknotes)
```

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

```r
typeof(banknotes)
```

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

Under the hood, data frames are lists, and each column is a list element. Because the class is `data.frame` rather than `list`, R treats data frames differently from ordinary lists. This difference is apparent in how data frames are printed compared to ordinary lists.

2.2.2 Implicit Coercion

R's types fall into a natural hierarchy of expressiveness:

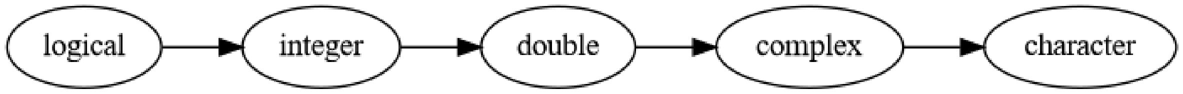


Figure 2.1: R's hierarchy of types.

Each type on the right is more expressive than the ones to its left. That is, with the convention that `FALSE` is `0` and `TRUE` is `1`, we can represent any logical value as an integer. In turn, we can represent any integer as a double, and any double as a complex number. By writing the number out, we can also represent any complex number as a string.

The point is that no information is lost as we follow the arrows from left to right along the types in the hierarchy. In fact, R will automatically and silently convert from types on the left to types on the right as needed. This is called **implicit coercion**.

As an example, consider what happens if we add a logical value to a number:

```r
TRUE + 2
```

```
[1] 3
```

R automatically converts the `TRUE` to the numeric value `1`, and then carries out the arithmetic as usual.

We've already seen implicit coercion at work once before, when we learned the `c` function. Since the elements of a vector all have to have the same type, if you pass several different types to `c`, then R tries to use implicit coercion to make them the same:

```r
x = c(TRUE, "hi", 1, 1+3i)
class(x)
```

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

```r
x
```

```
[1] "TRUE" "hi"   "1"    "1+3i"
```

Implicit coercion is strictly one-way; it never occurs in the other direction. If you want to coerce a type on the right to one on the left, you can do it explicitly with one of the `as.TYPE` functions. For instance, the `as.numeric` (or `as.double`) function coerces to numeric:

```r
as.numeric("3.1")
```

```
[1] 3.1
```

There are a few types that fall outside of the hierarchy entirely, like functions. Implicit coercion doesn't apply to these. If you try to use these types where it doesn't make sense to, R generally returns an error:

```r
sin + 3
```

```
Error in sin + 3: non-numeric argument to binary operator
```

If you try to use these types as elements of a vector, you get back a list instead:

```r
x = c(1, 2, sum)
class(x)
```

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

Understanding how implicit coercion works will help you avoid bugs, and can also be a time-saver. For example, we can use implicit coercion to succinctly count how many elements of a vector satisfy a some condition:

```r
x = c(1, 3, -1, 10, -2, 3, 8, 2)
condition = x < 4
sum(condition)    # or sum(x < 4)
```

```
[1] 6
```

If you still don't quite understand how the code above works, try inspecting each variable. In general, inspecting each step or variable is a good strategy for understanding why a piece of code works (or doesn't work!). Here the implicit coercion happens in the third line.

2.2.3 Matrices & Arrays

A **matrix** is the two-dimensional analogue of a vector. The elements, which are arranged into rows and columns, are ordered and homogeneous.

You can create a matrix from a vector with the `matrix` function. By default, the columns are filled first:

```r
# A matrix with 2 rows and 3 columns:
matrix(1:6, 2, 3)
```

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

The class of a matrix is always `matrix`, and the type matches the type of the elements:

```r
x = matrix(c("a", "b", NA, "c"), 2, 2)
x
```

```
     [,1] [,2]
[1,] "a"  NA  
[2,] "b"  "c" 
```

```r
class(x)
```

```
[1] "matrix" "array"
```

```
typeof(x)
```

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

You can use the matrix multiplication operator `%*%` to multiply two matrices with compatible dimensions.

An **array** is a further generalization of matrices to higher dimensions. You can create an array from a vector with the `array` function. The characteristics of arrays are almost identical to matrices, but the class of an array is always `array`.

2.2.4 Factors

A feature is **categorical** if it measures a qualitative category. For example, the genres `rock`, `blues`, `alternative`, `folk`, `pop` are categories.

R uses the class `factor` to represent categorical data. Visualizations and statistical models sometimes treat factors differently than other data types, so it's important to make sure you have the right data type. If you're ever unsure, remember that you can check the class of an object with the `class` function.

When you load a data set, R usually can't tell which features are categorical. That means identifying and converting the categorical features is up to you. For beginners, it can be difficult to understand whether a feature is categorical or not. The key is to think about whether you want to use the feature to divide the data into groups.

For example, if we want to know how many songs are in the `rock` genre, we first need to divide the songs by genre, and then count the number of songs in each group (or at least the `rock` group).

As a second example, months recorded as numbers can be categorical or not, depending on how you want to use them. You might want to treat them as categorical (for example, to compute max rainfall in each month) or you might want to treat them as numbers (for example, to compute the number of months time between two events).

The bottom line is that you have to think about what you'll be doing in the analysis. In some cases, you might treat a feature as categorical only for part of the analysis.

Let's think about which features are categorical in banknotes data set. To refresh our memory of what's in the data set, we can look at the structural summary:

```
str(banknotes)
```

```
'data.frame':	279 obs. of  17 variables:
 $ currency_code        : chr  "ARS" "ARS" "ARS" "ARS" ...
 $ country              : chr  "Argentina" "Argentina" "Argentina" "Argentina" ...
 $ currency_name        : chr  "Argentinian Peso" "Argentinian Peso" "Argentinian Peso" "Arg
 $ name                 : chr  "Eva Perón" "Julio Argentino Roca" "Domingo Faustino Sarmient
 $ gender               : chr  "F" "M" "M" "M" ...
 $ bill_count           : num  1 1 1 1 1 0.5 0.5 0.5 0.5 0.5 ...
 $ profession           : chr  "Activist" "Head of Gov't" "Head of Gov't" "Politician" ...
 $ known_for_being_first: chr  "No" "No" "No" "No" ...
 $ current_bill_value   : int  100 100 50 20 10 50 10 20 10 50 ...
 $ prop_total_bills     : num  NA NA NA NA NA 0.48 0.08 0.1 0.08 0.48 ...
 $ first_appearance_year: int  2012 1988 1999 1992 1970 1995 1993 1994 1993 1995 ...
 $ death_year           : chr  "1952" "1914" "1888" "1877" ...
 $ comments             : chr  "" "" "" "" ...
 $ hover_text           : chr  "" "" "" "" ...
 $ has_portrait         : chr  "true" "true" "true" "true" ...
 $ id                   : chr  "ARS_Evita" "ARS_Argentino" "ARS_Domingo" "ARS_Rosas" ...
 $ scaled_bill_value    : num  1 1 0.444 0.111 0 ...
```

The columns `bill_count`, `current_bill_value`, and `prop_total_bills` are quantitative rather than categorical, since they measure values and proportions of bills.

The `currency_code`, `country`, `currency_name`, `gender`, `profession`, `known_for_being_first`, and `has_portrait` columns are all categorical. We can see this better if we use the `table` function to compute frequencies for the values in the columns:

```
table(banknotes$currency_code)
```

```

ARS AUD BDT BOB CAD CLP COP CRC CVE CZK DOP GBP GEL IDR ILS ISK JMD JPY KGS KRW
  5   9   9  15   9   5  11   6   5   6  11   8   9  12   4   5   5   3   7   4
MWK MXN NGN NZD PEN PGK PHP RMB RSD SEK STD TND TRY UAH USD UYU VES VES ZAR
  7  11   9   5   7   1   9   6   9   6   8   4  12  10   7   7   2   6   5
```

```
table(banknotes$country)
```

```

            Argentina             Australia            Bangladesh
                    5                     9                     9
              Bolivia                Canada            Cape Verde
                   15                     9                     5
```

```
                  Chile                   China                Colombia
                      5                       6                      11
             Costa Rica          Czech Republic      Dominican Republic
                      6                       6                      11
                England                 Georgia                 Iceland
                      8                       9                       5
              Indonesia                  Israel                 Jamaica
                     12                       4                       5
                  Japan              Kyrgyzstan                  Malawi
                      3                       7                       7
                 Mexico             New Zealand                 Nigeria
                     11                       5                       9
       Papua New Guinea                    Peru             Philippines
                      1                       7                       9
  São Tomé and Príncipe                  Serbia            South Africa
                      8                       9                       5
            South Korea                  Sweden                 Tunisia
                      4                       6                       4
                 Turkey                 Ukraine           United States
                     12                      10                       7
                Uruguay               Venezuela
                      7                       8
```

```r
table(banknotes$gender)
```

```

  F   M 
 59 220 
```

```r
table(banknotes$known_for_being_first)
```

```

 No Yes 
185  94 
```

Each column has only a few unique values, repeated many times. These are ideal for grouping the data. If age had been recorded as a number, rather than a range, it would probably be better to treat it as quantitative, since there would be far more unique values. Columns with many unique values don't make good categorical features, because each group will only have a few elements!

The `name`, `comments`, `hover_text`, and `id` columns contain qualitative strings rather than categories, where values are rarely or never repeated. This means using them for grouping is generally unhelpful, since most of the groups will just be single observations.

That leaves us with the `first_appearance_year` and `death_year` columns. It's easy to imagine grouping the data by year, but these are also clearly numbers. These columns can be treated as quantitative or categorical data, depending on how we want to use them to analyze the data.

Let's convert the `currency_code` column to a factor. To do this, use the `factor` function:

```
currency_codes = factor(banknotes$currency_code)
head(currency_codes)
```

```
[1] ARS ARS ARS ARS ARS AUD
39 Levels: ARS AUD BDT BOB CAD CLP COP CRC CVE CZK DOP GBP GEL IDR ILS ... ZAR
```

Notice that factors are printed differently than strings.

The categories of a factor are called **levels**. You can list the levels with the `levels` function:

```
levels(currency_codes)
```

```
 [1] "ARS" "AUD" "BDT" "BOB" "CAD" "CLP" "COP" "CRC" "CVE" "CZK" "DOP" "GBP"
[13] "GEL" "IDR" "ILS" "ISK" "JMD" "JPY" "KGS" "KRW" "MWK" "MXN" "NGN" "NZD"
[25] "PEN" "PGK" "PHP" "RMB" "RSD" "SEK" "STD" "TND" "TRY" "UAH" "USD" "UYU"
[37] "VES" "VES" "ZAR"
```

Factors remember all possible levels even if you take a subset:

```
currency_codes[1:3]
```

```
[1] ARS ARS ARS
39 Levels: ARS AUD BDT BOB CAD CLP COP CRC CVE CZK DOP GBP GEL IDR ILS ... ZAR
```

This is another way factors are different from strings. Factors "remember" all possible levels even if they aren't present. This ensures that if you plot a factor, the missing levels will still be represented on the plot.

You can make a factor forget levels that aren't present with the `droplevels` function:

```r
droplevels(currency_codes[1:3])
```

```
[1] ARS ARS ARS
Levels: ARS
```

2.3 Special Values

R has four special values to represent missing or invalid data.

2.3.1 Missing Values

The value `NA`, called the **missing value**, represents missing entries in a data set. It's implied that the entries are missing due to how the data was collected, although there are exceptions. As an example, imagine the data came from a survey, and respondents chose not to answer some questions. In the data set, their answers for those questions can be recorded as `NA`.

The missing value is a chameleon: it can be a logical, integer, numeric, complex, or character value. By default, the missing value is logical, and the other types occur through coercion (Section 2.2.2):

```r
class(NA)
```

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

```r
class(c(1, NA))
```

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

```r
class(c("hi", NA, NA))
```

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

The missing value is also contagious: it represents an unknown quantity, so using it as an argument to a function usually produces another missing value. The idea is that if the inputs to a computation are unknown, generally so is the output:

```r
NA - 3
```

```
[1] NA
```

```r
mean(c(1, 2, NA))
```

```
[1] NA
```

As a consequence, testing whether an object is equal to the missing value with `==` doesn't return a meaningful result:

```r
5 == NA
```

```
[1] NA
```

```r
NA == NA
```

```
[1] NA
```

You can use the `is.na` function instead:

```r
is.na(5)
```

```
[1] FALSE
```

```r
is.na(NA)
```

```
[1] TRUE
```

```r
is.na(c(1, NA, 3))
```

```
[1] FALSE  TRUE FALSE
```

Missing values are a feature that sets R apart from most other programming languages.

2.3.2 Infinity

The value `Inf` represents infinity, and can be numeric or complex. You're most likely to encounter it as the result of certain computations:

```r
13 / 0
```

```
[1] Inf
```

```r
class(Inf)
```

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

You can use the `is.infinite` function to test whether a value is infinite:

```r
is.infinite(3)
```

```
[1] FALSE
```

```r
is.infinite(c(-Inf, 0, Inf))
```

```
[1]  TRUE FALSE  TRUE
```

2.3.3 Not a Number

The value `NaN` ("not a number") represents a quantity that's undefined mathematically. For instance, dividing 0 by 0 is undefined:

```r
0 / 0
```

```
[1] NaN
```

```r
class(NaN)
```

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

Like `Inf`, `NaN` can be numeric or complex.

You can use the `is.nan` function to test whether a value is `NaN`:

```r
is.nan(c(10.1, log(-1), 3))
```

```
Warning in log(-1): NaNs produced
```

```
[1] FALSE  TRUE FALSE
```

2.3.4 Null

The value `NULL` represents a quantity that's undefined in R. Most of the time, `NULL` indicates the absence of a result. For instance, vectors don't have dimensions, so the `dim` function returns `NULL` for vectors:

```r
dim(c(1, 2))
```

```
NULL
```

```r
class(NULL)
```

```
[1] "NULL"
```

```r
typeof(NULL)
```

```
[1] "NULL"
```

Unlike the other special values, `NULL` has its own unique type and class.

You can use the `is.null` function to test whether a value is `NULL`:

```r
is.null("null")
```

```
[1] FALSE
```

```r
is.null(NULL)
```

```
[1] TRUE
```

2.4 Indexing

The way to get and set elements of a data structure is by **indexing**. Sometimes this is also called **subsetting** or (element) **extraction**. Indexing is a fundamental operation in R, key to reasoning about how to solve problems with the language.

We first saw indexing in Section 1.6, where we used `$`, the dollar sign operator, to get and set data frame columns. We saw indexing again in Section 2.1.2, where we used `[`, the indexing or square bracket operator, to get and set elements of vectors.

The indexing operator `[` is R's primary operator for indexing. It works in four different ways, depending on the type of the index you use. These four ways to select elements are:

1. All elements, with no index
2. By position, with a numeric index
3. By name, with a character index
4. By condition, with a logical index

Let's examine each in more detail. We'll use this vector as an example, to keep things concise:

```
x = c(a = 10, b = 20, c = 30, d = 40, e = 50)
x
```

```
 a  b  c  d  e 
10 20 30 40 50 
```

Even though we're using a vector here, the indexing operator works with almost all data structures, including factors, lists, matrices, and data frames. We'll look at unique behavior for some of these later on.

2.4.1 All Elements

The first way to use `[` to select elements is to leave the index blank. This selects all elements:

```
x[]
```

```
 a  b  c  d  e 
10 20 30 40 50 
```

This way of indexing is rarely used for getting elements, since it's the same as entering the variable name without the indexing operator. Instead, its main use is for setting elements. Suppose we want to set all the elements of `x` to `5`. You might try writing this:

```
x = 5
x
```

```
[1] 5
```

Rather than setting each element to `5`, this sets `x` to the scalar `5`, which is not what we want. Let's reset the vector and try again, this time using the indexing operator:

```
x = c(a = 10, b = 20, c = 30, d = 40, e = 50)
x[] = 5
x
```

```
a b c d e 
5 5 5 5 5 
```

As you can see, now all the elements are `5`. So the indexing operator is necessary to specify that we want to set the elements rather than the whole variable.

Let's reset `x` one more time, so that we can use it again in the next example:

```
x = c(a = 10, b = 20, c = 30, d = 40, e = 50)
```

2.4.2 By Position

The second way to use `[` is to select elements by position. This happens when you use an integer or numeric index. We already saw the basics of this in Section 2.1.2.

The positions of the elements in a vector (or other data structure) correspond to numbers starting from 1 for the first element. This way of indexing is frequently used together with the sequence operator `:` to get ranges of values. For instance, let's get the 2nd through 4th elements of `x`:

```
x[2:4]
```

```
 b  c  d 
20 30 40 
```

You can also use this way of indexing to set specific elements or ranges of elements. For example, let's set the 3rd and 5th elements of `x` to `9` and `7`, respectively:

```r
x[c(3, 5)] = c(9, 7)
x
```

```
 a  b  c  d  e 
10 20  9 40  7 
```

When getting elements, you can repeat numbers in the index to get the same element more than once. You can also use the order of the numbers to control the order of the elements:

```r
x[c(2, 1, 2, 2)]
```

```
 b  a  b  b 
20 10 20 20 
```

Finally, if the index contains only negative numbers, the elements at those positions are excluded rather than selected. For instance, let's get all elements except the 1st and 5th:

```r
x[-c(1, 5)]
```

```
 b  c  d 
20  9 40 
```

When you index by position, the index should always be all positive or all negative. Using a mix of positive and negative numbers causes R to emit error rather than returning elements, since it's unclear what the result should be:

```r
x[c(-1, 2)]
```

```
Error in x[c(-1, 2)]: only 0's may be mixed with negative subscripts
```

2.4.3 By Name

The third way to use `[` is to select elements by name. This happens when you use a character vector as the index, and only works with named data structures.

Like indexing by position, you can use indexing by name to get or set elements. You can also use it to repeat elements or change the order. Let's get elements `a`, `c`, `d`, and `a` again from the vector `x`:

```
y = x[c("a", "c", "d", "a")]
y
```

```
 a  c  d  a 
10  9 40 10 
```

Element names are generally unique, but if they're not, indexing by name gets or sets the first element whose name matches the index:

```
y["a"]
```

```
 a 
10 
```

Let's reset `x` again to prepare for learning about the final way to index:

```
x = c(a = 10, b = 20, c = 30, d = 40, e = 50)
```

2.4.4 By Condition

The fourth and final way to use `[` is to select elements based on a condition. This happens when you use a logical vector as the index. The logical vector should have the same length as what you're indexing, and will be recycled if it doesn't.

Congruent Vectors

To understand indexing by condition, we first need to learn about congruent vectors. Two vectors are **congruent** if they have the same length and they correspond element-by-element.

For example, suppose you do a survey that records each respondent's favorite animal and age. These are two different vectors of information, but each person will have a response for both. So you'll have two vectors that are the same length:

```
animal = c("dog", "cat", "iguana")
age = c(31, 24, 72)
```

The 1st element of each vector corresponds to the 1st person, the 2nd to the 2nd person, and so on. These vectors are congruent.

Notice that columns in a data frame are always congruent!

Back to Indexing

When you index by condition, the index should generally be congruent to the object you're indexing. Elements where the index is `TRUE` are kept and elements where the index is `FALSE` are dropped.

If you create the index from a condition on the object, it's automatically congruent. For instance, let's make a condition based on the vector `x`:

```
is_small = x < 25
is_small
```

```
    a     b     c     d     e 
 TRUE  TRUE FALSE FALSE FALSE 
```

The 1st element in the logical vector `is_small` corresponds to the 1st element of `x`, the 2nd to the 2nd, and so on. The vectors `x` and `is_small` are congruent.

It makes sense to use `is_small` as an index for `x`, and it gives us all the elements less than 25:

```
x[is_small]
```

```
 a  b 
10 20 
```

Of course, you can also avoid using an intermediate variable for the condition:

```r
x[x > 10]
```

```
 b  c  d  e 
20 30 40 50 
```

If you create index some other way (not using the object), make sure that it's still congruent to the object. Otherwise, the subset returned from indexing might not be meaningful.

You can also use indexing by condition to set elements, just as the other ways of indexing can be used to set elements. For instance, let's set all the elements of `x` that are greater than `10` to the missing value `NA`:

```r
x[x > 10] = NA
x
```

```
 a  b  c  d  e 
10 NA NA NA NA 
```

2.4.5 Logic

All of the conditions we've seen so far have been written in terms of a single test. If you want to use more sophisticated conditions, R provides operators to negate and combine logical vectors. These operators are useful for working with logical vectors even outside the context of indexing.

Negation

The **NOT operator** `!` converts `TRUE` to `FALSE` and `FALSE` to `TRUE`:

```r
x = c(TRUE, FALSE, TRUE, TRUE, NA)
x
```

```
[1]  TRUE FALSE  TRUE  TRUE    NA
```

```r
!x
```

```
[1] FALSE  TRUE FALSE FALSE    NA
```

You can use `!` with a condition:

```
y = c("hi", "hello")
!(y == "hi")
```

```
[1] FALSE  TRUE
```

The NOT operator is vectorized.

Combinations

R also has operators for combining logical values.

The **AND operator** `&` returns `TRUE` only when both arguments are `TRUE`. Here are some examples:

```
FALSE & FALSE
```

```
[1] FALSE
```

```
TRUE & FALSE
```

```
[1] FALSE
```

```
FALSE & TRUE
```

```
[1] FALSE
```

```
TRUE & TRUE
```

```
[1] TRUE
```

```
c(TRUE, FALSE, TRUE) & c(TRUE, TRUE, FALSE)
```

```
[1]  TRUE FALSE FALSE
```

The **OR operator** `|` returns `TRUE` when at least one argument is `TRUE`. Let's see some examples:

```r
FALSE | FALSE
```

```
[1] FALSE
```

```r
TRUE | FALSE
```

```
[1] TRUE
```

```r
FALSE | TRUE
```

```
[1] TRUE
```

```r
TRUE | TRUE
```

```
[1] TRUE
```

```r
c(TRUE, FALSE) | c(TRUE, TRUE)
```

```
[1] TRUE TRUE
```

Be careful: everyday English is less precise than logic. You might say:

> I want all subjects with age over 50 and all subjects that like cats.

But in logic this means:

`(subject age over 50) OR (subject likes cats)`

So think carefully about whether you need both conditions to be true (AND) or at least one (OR).

Rarely, you might want *exactly one* condition to be true. The **XOR (eXclusive OR) function** `xor()` returns `TRUE` when exactly one argument is `TRUE`. For example:

```r
xor(FALSE, FALSE)
```

```
[1] FALSE
```

```r
xor(TRUE, FALSE)
```

```
[1] TRUE
```

```r
xor(TRUE, TRUE)
```

```
[1] FALSE
```

The AND, OR, and XOR operators are vectorized.

Short-circuiting

The second argument is irrelevant in some conditions:

- `FALSE &` is always `FALSE`
- `TRUE |` is always `TRUE`

Now imagine you have `FALSE & long_computation()`. You can save time by skipping `long_computation()`. A **short-circuit operator** does exactly that.

R has two short-circuit operators:

- `&&` is a short-circuited `&`
- `||` is a short-circuited `|`

These operators only evaluate the second argument if it is necessary to determine the result. Here are some of these:

```r
TRUE && FALSE
```

```
[1] FALSE
```

```r
TRUE && TRUE
```

```
[1] TRUE
```

```r
TRUE || TRUE
```

```
[1] TRUE
```

The short-circuit operators are not vectorized—they only accept length-1 arguments:

```r
c(TRUE, FALSE) && c(TRUE, TRUE)
```

```
Error in c(TRUE, FALSE) && c(TRUE, TRUE): 'length = 2' in coercion to 'logical(1)'
```

Because of this, you can't use short-circuit operators for indexing. Their main use is in writing conditions for if-expressions, which we'll learn about later on.

Note

Prior to R 4.3.0, short-circuit operators didn't raise an error for inputs with length greater than 1 (and thus were a common source of bugs).

2.5 Exercises

2.5.1 Exercise

The `rep` function is another way to create a vector. Read the help file for the `rep` function.

1. What does the `rep` function do to create a vector? Give an example.
2. The `rep` function has parameters `times` and `each`. What does each do, and how do they differ? Give examples for both.
3. Can you set both of `times` and `each` in a single call to `rep`? If the function raises an error, explain what the error message means. If the function returns a result, explain how the result corresponds to the arguments you chose.

2.5.2 Exercise

Considering how implicit coercion works (Section 2.2.2):

1. Why does `"3" + 4` raise an error?
2. Why does `"TRUE" == TRUE` return `TRUE`?
3. Why does `"FALSE" < TRUE` return TRUE?

2.5.3 Exercise

1. Section 2.3.1 described the missing value as a "chameleon" because it can have many different types. Is `Inf` also a chameleon? Use examples to justify your answer.

2. The missing value is also "contagious" because using it as an argument usually produces another missing value. Is `Inf` contagious? Again, use examples to justify your answer.

2.5.4 Exercise

The `table` function is useful for counting all sorts of things, not just level frequencies for a factor. For instance, you can use `table` to count how many `TRUE` and `FALSE` values there are in a logical vector.

1. For the banknotes data, how many bills have value below 1,000?
2. How many bills with value above 1,000 depict women?

3 Exploring Data

Learning Goals

After completing this chapter, learners should be able to:

- Describe when to use `[` versus `[[`
- Index data frames to get specific rows, columns, or subsets
- Install and load packages
- Describe the grammar of graphics
- Make a plot
- Save a plot to an image file
- Call a function repeatedly with `sapply` or `lapply`
- Split data into groups and apply a function to each

Now that you have a solid foundation in the basic functions and data structures of R, you can move on to its most popular application: data analysis. In this chapter, you'll learn how to efficiently explore and summarize data with visualizations and statistics. Along the way, you'll also learn how to use apply functions, which are essential to fluency in R.

3.1 Indexing Data Frames

This section explains how to get and set data in a data frame, expanding on the indexing techniques you learned in Section 2.4. Under the hood, every data frame is a list, so first you'll learn about indexing lists.

3.1.1 Indexing Lists

Lists are a **container** for other types of R objects. When you select an element from a list, you can either keep the container (the list) or discard it. The indexing operator `[` almost always keeps containers.

As an example, let's get some elements from a small list:

```r
x = list(first = c(1, 2, 3), second = sin, third = c("hi", "hello"))
y = x[c(1, 3)]
y
```

```
$first
[1] 1 2 3

$third
[1] "hi"    "hello"
```

```r
class(y)
```

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

The result is still a list. Even if we get just one element, the result of indexing a list with `[` is a list:

```r
class(x[1])
```

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

Sometimes this will be exactly what we want. But what if we want to get the first element of `x` so that we can use it in a vectorized function? Or in a function that only accepts numeric arguments? We need to somehow get the element and discard the container.

The solution to this problem is the **extraction operator** `[[`, which is also called the double square bracket operator. The extraction operator is the primary way to get and set elements of lists and other containers.

Unlike the indexing operator `[`, the extraction operator always discards the container:

```r
x[[1]]
```

```
[1] 1 2 3
```

```r
class(x[[1]])
```

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

The tradeoff is that the extraction operator can only get or set one element at a time. Note that the element can be a vector, as above. Because it can only get or set one element at a time, the extraction operator can only index by position or name. Blank and logical indexes are not allowed.

The final difference between the index operator `[` and the extraction operator `[[` has to do with how they handle invalid indexes. The index operator `[` returns `NA` for invalid vector elements, and `NULL` for invalid list elements:

```r
c(1, 2)[10]
```

```
[1] NA
```

```r
x[10]
```

```
$<NA>
NULL
```

On the other hand, the extraction operator `[[` raises an error for invalid elements:

```r
x[[10]]
```

```
Error in x[[10]]: subscript out of bounds
```

The indexing operator `[` and the extraction operator `[[` both work with any data structure that has elements. However, you'll generally use the indexing operator `[` to index vectors, and the extraction operator `[[` to index containers (such as lists).

3.1.2 Two-dimensional Indexing

For two-dimensional objects, like matrices and data frames, you can pass the indexing operator `[` or the extraction operator `[[` a separate index for each dimension. The rows come first:

```
DATA[ROWS, COLUMNS]
```

For instance, let's get the first 3 rows and all columns of the banknotes data:

```r
banknotes[1:3, ]
```

```
  currency_code   country    currency_name                         name gender
1           ARS Argentina Argentinian Peso                    Eva Perón      F
2           ARS Argentina Argentinian Peso        Julio Argentino Roca      M
3           ARS Argentina Argentinian Peso Domingo Faustino Sarmiento      M
  bill_count    profession known_for_being_first current_bill_value
1          1      Activist                    No                100
2          1 Head of Gov't                    No                100
3          1 Head of Gov't                    No                 50
  prop_total_bills first_appearance_year death_year comments hover_text
1               NA                  2012       1952
2               NA                  1988       1914
3               NA                  1999       1888
  has_portrait            id scaled_bill_value
1         true     ARS_Evita         1.0000000
2         true ARS_Argentino         1.0000000
3         true   ARS_Domingo         0.4444444
```

As we saw in Section 2.4.1, leaving an index blank means all elements.

As another example, let's get the 3rd and 5th row, and the 2nd and 4th column:

```r
banknotes[c(3, 5), c(2, 4)]
```

```
    country                       name
3 Argentina Domingo Faustino Sarmiento
5 Argentina            Manuel Belgrano
```

Mixing several different ways of indexing is allowed. So for example, we can get the same above, but use column names instead of positions:

```r
banknotes[c(3, 5), c("currency_name", "name")]
```

```
     currency_name                       name
3 Argentinian Peso Domingo Faustino Sarmiento
5 Argentinian Peso            Manuel Belgrano
```

For data frames, it's especially common to index the rows by condition and the columns by name. For instance, let's get the `name` and `profession` columns for all women in the banknotes data set:

```
result = banknotes[banknotes$gender == "F", c("name", "profession")]
head(result)
```

```
                 name profession
1           Eva Perón   Activist
7        Mary Gilmore     Writer
10        Edith Cowan Politician
11       Nellie Melba   Musician
13        Mary Reibey      Other
14 Queen Elizabeth II    Monarch
```

Also see Section 5.2 for a case where the `[` operator behaves in a surprising way.

3.2 Packages

A **package** is a collection of functions for use in R. Packages usually include documentation, and can also contain examples, vignettes, and data sets. Most packages are developed by members of the R community, so quality varies. There are also a few packages that are built into R but provide extra features. We'll use a package in Section 3.3, so we're learning about them now.

The Comprehensive R Archive Network, or CRAN, is the main place people publish packages. As of writing, there were 18,619 packages posted to CRAN. This number has been steadily increasing as R has grown in popularity.

Packages span a wide variety of topics and disciplines. There are packages related to statistics, social sciences, geography, genetics, physics, biology, pharmacology, economics, agriculture, and more. The best way to find packages is to search online, but the CRAN website also provides "task views" if you want to browse popular packages related to a specific discipline.

The `install.packages` function installs one or more packages from CRAN. Its first argument is the packages to install, as a character vector.

When you run `install.packages`, R will ask you to choose which **mirror** to download the package from. A mirror is a web server that has the same set of files as some other server. Mirrors are used to make downloads faster and to provide redundancy so that if a server stops working, files are still available somewhere else. CRAN has dozens of mirrors; you should choose one that's geographically nearby, since that usually produces the best download speeds. If you aren't sure which mirror to choose, you can use the 0-Cloud mirror, which attempts to automatically choose a mirror near you.

As an example, here's the code to install the remotes package:

```r
install.packages("remotes")
```

If you run the code above, you'll be asked to select a mirror, and then see output that looks something like this:

```
--- Please select a CRAN mirror for use in this session ---
trying URL 'https://cloud.r-project.org/src/contrib/remotes_2.3.0.tar.gz'
Content type 'application/x-gzip' length 148405 bytes (144 KB)
==================================================
downloaded 144 KB

* installing *source* package ‘remotes’ ...
** package ‘remotes’ successfully unpacked and MD5 sums checked
** using staged installation
** R
** inst
** byte-compile and prepare package for lazy loading
** help
*** installing help indices
** building package indices
** installing vignettes
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (remotes)

The downloaded source packages are in
        ‘/tmp/Rtmp8t6iGa/downloaded_packages’
```

R goes through a variety of steps to install a package, even installing other packages that the package depends on. You can tell that a package installation succeeded by the final line `DONE`. When a package installation fails, R prints an error message explaining the problem instead.

Once a package is installed, it stays on your computer until you remove it or remove R. This means you only need to install each package once. However, most packages are periodically updated. You can reinstall a package using `install.packages` the same way as above to get the latest version.

Alternatively, you can update all of the R packages you have installed at once by calling the `update.packages` function. Beware that this may take a long time if you have a lot of packages installed.

The function to remove packages is `remove.packages`. Like `install.packages`, this function's first argument is the packages to remove, as a character vector.

If you want to see which packages are installed, you can use the `installed.packages` function. It does not require any arguments. It returns a matrix with one row for each package and columns that contain a variety of information. Here's an example:

```r
packages = installed.packages()
# Just print the version numbers for 10 packages.
packages[1:10, "Version"]
```

```
     abind        ape    askpass assertthat  backports  base64enc         BH 
   "1.4-5"      "5.8"    "1.2.0"    "0.2.1"    "1.4.1"    "0.1-3" "1.84.0-0" 
       bit      bit64     bitops 
   "4.0.5"    "4.0.5"    "1.0-7" 
```

You'll see a different set of packages, since you have a different computer.

Before you can use the functions (or other resources) in an installed package, you must load the package with the `library` function. R doesn't load packages automatically because each package you load uses memory and may conflict with other packages. Thus you should only load the packages you need for whatever it is that you want to do. When you restart R, the loaded packages are cleared and you must again load any packages you want to use.

Let's load the remotes package we installed earlier:

```r
library("remotes")
```

The `library` function works with or without quotes around the package name, so you may also see people write things like `library(remotes)`. We recommend using quotes to make it unambiguous that you are not referring to a variable.

A handful of packages print out a message when loaded, but the vast majority do not. Thus you can assume the call to `library` was successful if nothing is printed. If something goes wrong while loading a package, R will print out an error message explaining the problem.

Finally, not all R packages are published to CRAN. GitHub is another popular place to publish R packages, especially ones that are experimental or still in development. Unlike CRAN, GitHub is a general-purpose website for publishing code written in any programming language, so it contains much more than just R packages and is not specifically R-focused.

The remotes package that we just installed and loaded provides functions to install packages from GitHub. It is generally better to install packages from CRAN when they are available there, since the versions on CRAN tend to be more stable and intended for a wide audience. However, if you want to install a package from GitHub, you can learn more about the remotes package by reading its online documentation.

3.3 Data Visualization

There are three popular systems for creating visualizations in R:

1. The base R functions (primarily the `plot` function)
2. The lattice package
3. The ggplot2 package

These three systems are not interoperable! Consequently, it's best to choose one to use exclusively. Compared to base R, both lattice and ggplot2 are better at handling grouped data and generally require less code to create a nice-looking visualization.

The ggplot2 package is so popular that there are now knockoff packages for other data-science-oriented programming languages like Python and Julia. The package is also part of the Tidyverse, a popular collection of R packages designed to work well together. Because of these advantages, we'll use ggplot2 for visualizations in this and all future lessons.

ggplot2 has detailed documentation and also a cheatsheet.

The "gg" in ggplot2 stands for **grammar of graphics**. The idea of a grammar of graphics is that visualizations can be built up in layers. In ggplot2, the three layers every plot must have are:

- Data
- Geometry
- Aesthetics

There are also several optional layers. Here are a few:

Layer	Description
scales	Title, label, and axis value settings
facets	Side-by-side plots
guides	Axis and legend position settings
annotations	Shapes that are not mapped to data
coordinates	Coordinate systems (Cartesian, logarithmic, polar)

As an example, let's plot the banknotes data. First, we need to load ggplot2. As always, if this is your first time using the package, you'll have to install it. Then you can load the package:

```r
# install.packages("ggplot2")
library("ggplot2")
```

What kind of plot should we make? It depends on what we want to know about the data set. Suppose we want to understand the relationship between a banknote's value and how long ago the person on the banknote died, as well as whether this is affected by gender. One way to show this is to make a scatter plot.

Before plotting, we need to make sure that the `death_year` column is numeric:

```r
is_blank = banknotes$death_year %in% c("", "-")
banknotes$death_year[is_blank] = NA
banknotes$death_year = as.numeric(banknotes$death_year)
```

Now we're ready to make the plot.

3.3.1 Layer 1: Data

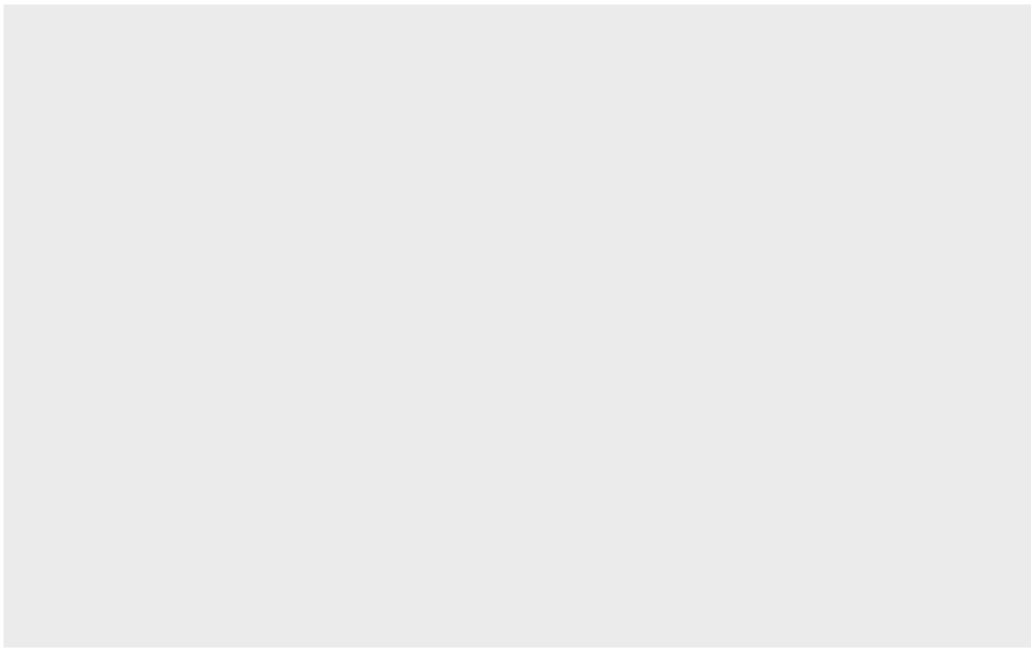
The data layer determines the data set used to make the plot. ggplot and most other Tidyverse packages are designed for working with **tidy** data frames. Tidy means:

1. Each observation has its own row.
2. Each feature has its own column.
3. Each value has its own cell.

Tidy data sets are convenient in general. A later lesson will cover how to make an untidy data set tidy. Until then, we'll take it for granted that the data sets we work with are tidy.

To set up the data layer, call the `ggplot` function on a data frame:

```r
ggplot(banknotes)
```



This returns a blank plot. We still need to add a few more layers.

3.3.2 Layer 2: Geometry

The **geom**etry layer determines the shape or appearance of the visual elements of the plot. In other words, the geometry layer determines what kind of plot to make: one with points, lines, boxes, or something else.

There are many different geometries available in ggplot2. The package provides a function for each geometry, always prefixed with `geom_`.

To add a geometry layer to the plot, choose the `geom_` function you want and add it to the plot with the `+` operator:

```
ggplot(banknotes) + geom_point()
```

```
Error in `geom_point()`:
! Problem while setting up geom.
i Error occurred in the 1st layer.
Caused by error in `compute_geom_1()`:
! `geom_point()` requires the following missing aesthetics: x and y.
```

This returns an error message that we're missing aesthetics `x` and `y`. We'll learn more about aesthetics in the next section, but this error message is especially helpful: it tells us exactly what we're missing. When you use a geometry you're unfamiliar with, it can be helpful to run the code for just the data and geometry layer like this, to see exactly which aesthetics need to be set.

As we'll see later, it's possible to add multiple geometries to a plot.

3.3.3 Layer 3: Aesthetics

The **aes**thetic layer determines the relationship between the data and the geometry. Use the aesthetic layer to map features in the data to **aesthetics** (visual elements) of the geometry.

The `aes` function creates an aesthetic layer. The syntax is:

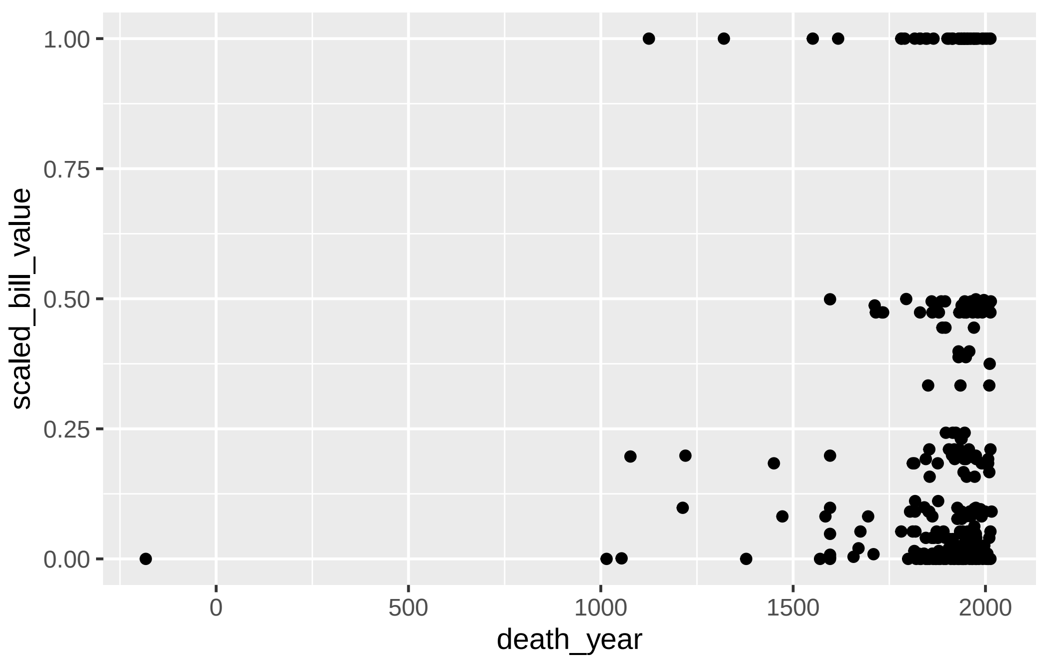
```
aes(AESTHETIC = FEATURE, ...)
```

The names of the aesthetics depend on the geometry, but some common ones are `x`, `y`, `color`, `fill`, `shape`, and `size`. There is more information about and examples of aesthetic names in the documentation.

For example, we want to put `death_year` on the x-axis and `scaled_bill_value` on the y-axis. It's best to use `scaled_bill_value` here rather than `current_bill_value` because the different countries use different scales of currency. For example, 1 United States Dollar is worth approximately 100 Japanese Yen. So the aesthetic layer should be:

```r
ggplot(banknotes) +
  geom_point() +
  aes(x = death_year, y = scaled_bill_value)
```

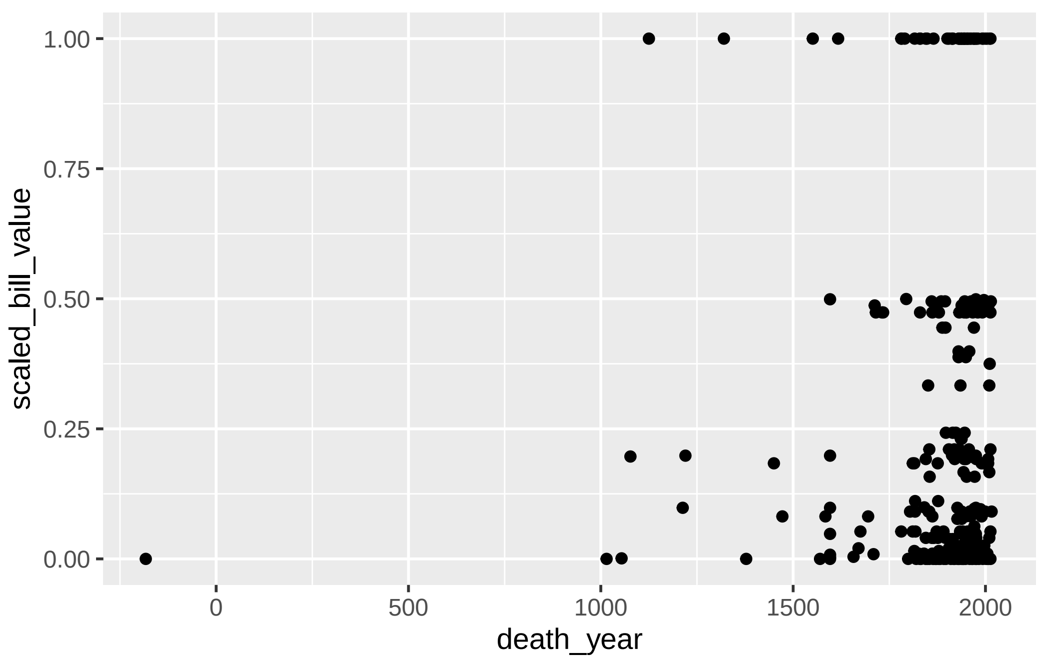
```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_point()`).
```



In the `aes` function, column names are never quoted. In older versions of ggplot2, you must pass the aesthetic layer as the second argument of the `ggplot` function rather than using `+` to add it to the plot. This syntax is still widely used:

```
ggplot(banknotes, aes(x = death_year, y = scaled_bill_value)) +
  geom_point()
```

```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_point()`).
```

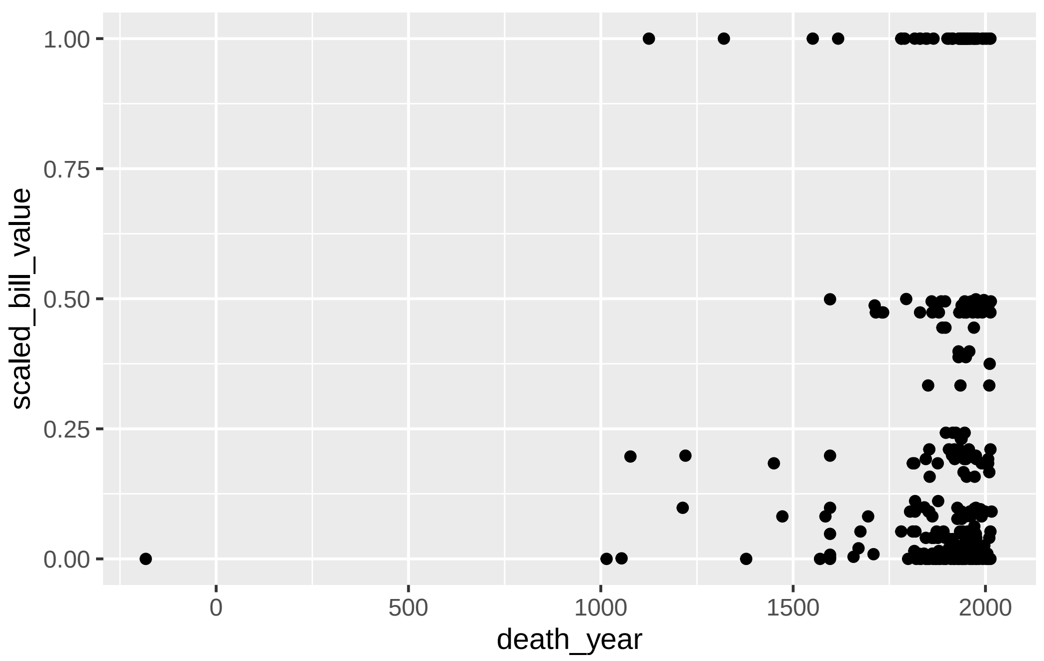


Per-geometry Aesthetics

When you add the aesthetic layer or pass it to the `ggplot` function, it applies to the entire plot. You can also set an aesthetic layer individually for each geometry, by passing the layer as the first argument in the `geom_` function:

```
ggplot(banknotes) +
  geom_point(aes(x = death_year, y = scaled_bill_value))
```

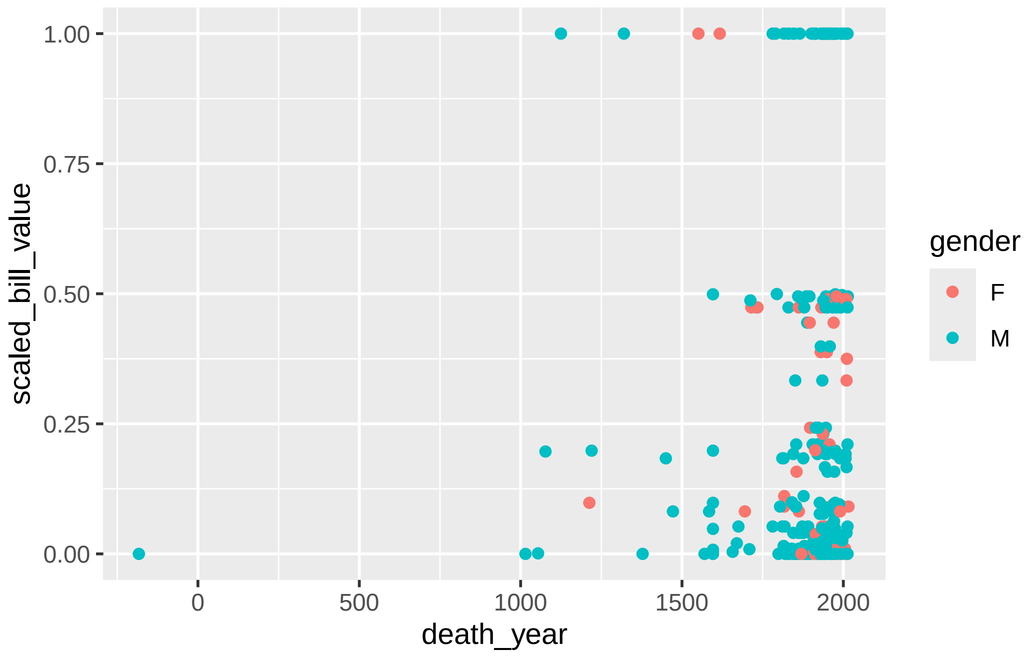
```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_point()`).
```



This is really only useful when you have multiple geometries. As an example, let's color-code the points by gender:

```r
ggplot(banknotes) +
  geom_point(aes(x = death_year, y = scaled_bill_value, color = gender))
```

```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_point()`).
```

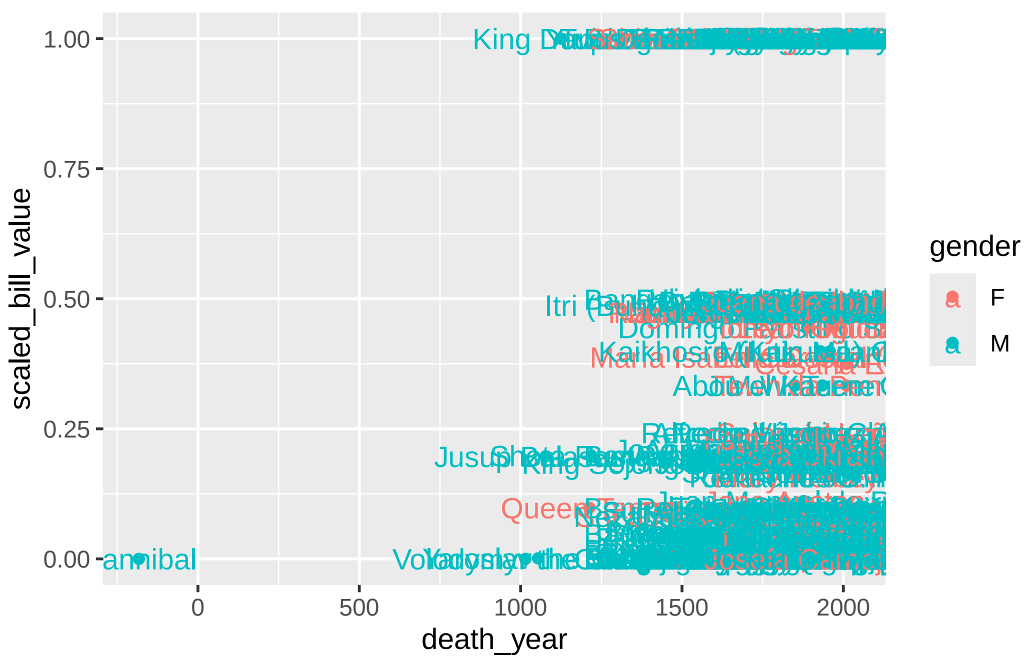


Now let's also add labels to each point. To do this, we need to add another geometry:

```r
ggplot(banknotes) +
  geom_point() +
  aes(x = death_year, y = scaled_bill_value, color = gender, label = name) +
  geom_text()
```

```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_point()`).
```

```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_text()`).
```

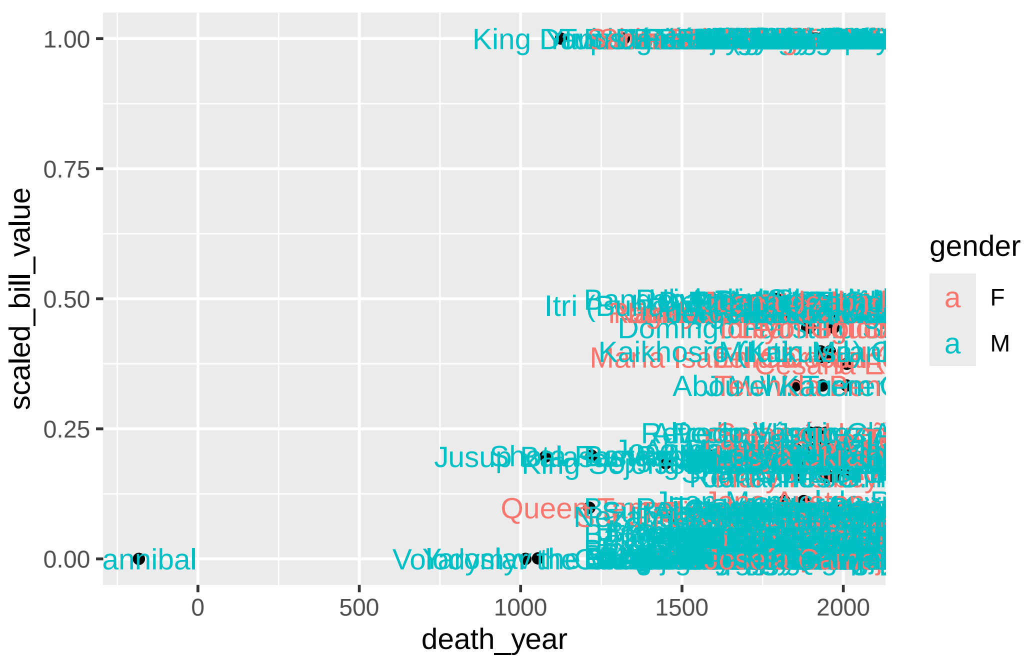


Where we put the aesthetics matters:

```r
ggplot(banknotes) +
  geom_point() +
  aes(x = death_year, y = scaled_bill_value, label = name) +
  geom_text(aes(color = gender))
```

```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_point()`).
```

```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_text()`).
```

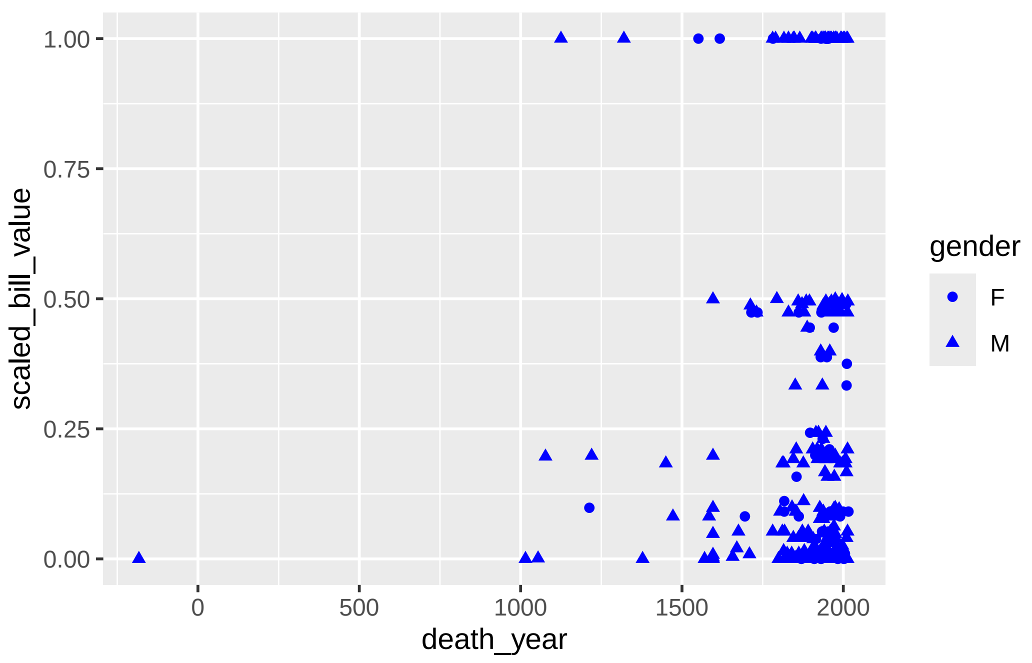


Constant Aesthetics

If you want to set an aesthetic to a constant value, rather than one that's data dependent, do so in the geometry layer rather than the aesthetic layer. For instance, suppose use point shape rather than color to indicate gender, and we want to make all of the points blue:

```
ggplot(banknotes) +
  geom_point(color = "blue") +
  aes(x = death_year, y = scaled_bill_value, shape = gender)
```

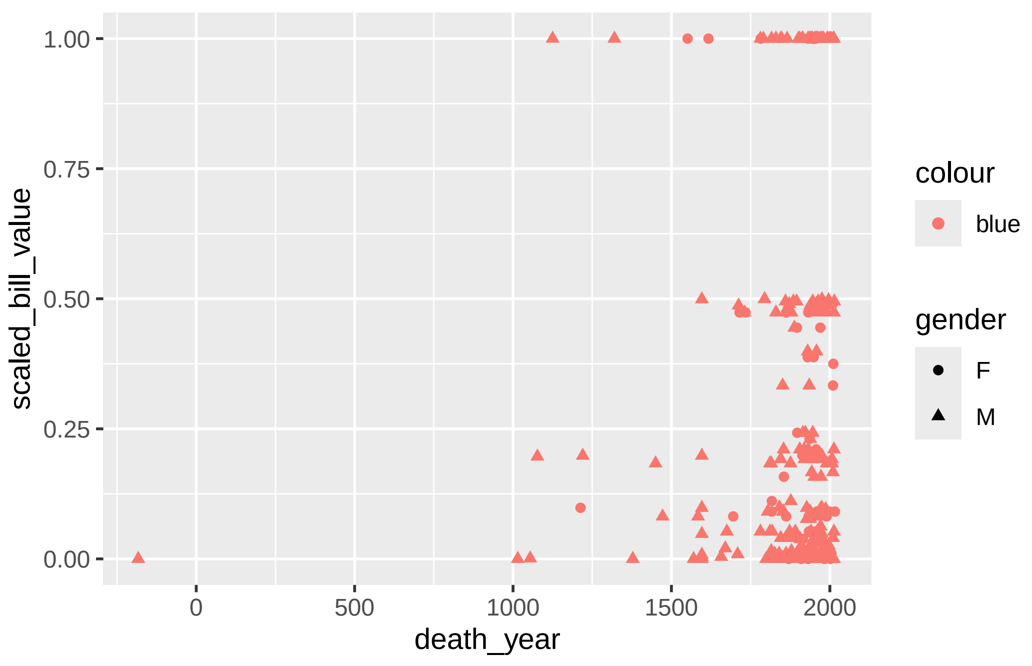
```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_point()`).
```



If you set an aesthetic to a constant value inside of the aesthetic layer, the results you get might not be what you expect:

```r
ggplot(banknotes) +
  geom_point() +
  aes(x = death_year, y = scaled_bill_value, shape = gender, color = "blue")
```

```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_point()`).
```



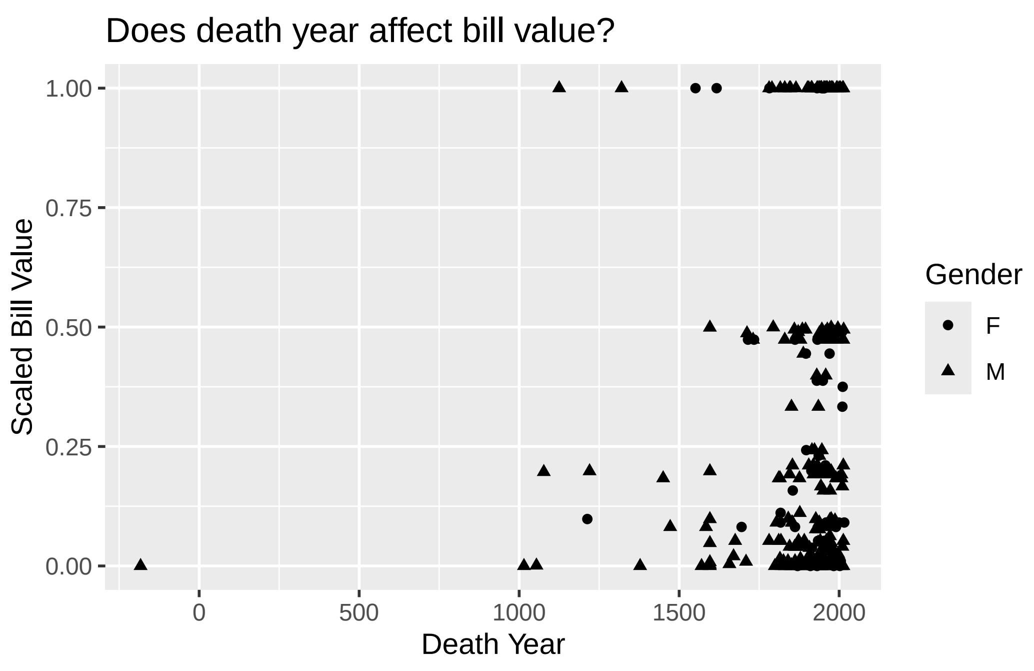
3.3.4 Layer 4: Scales

The scales layer controls the title, axis labels, and axis scales of the plot. Most of the functions in the scales layer are prefixed with `scale_`, but not all of them.

The `labs` function is especially important, because it's used to set the title and axis labels:

```
ggplot(banknotes) +
  geom_point() +
  aes(x = death_year, y = scaled_bill_value, shape = gender) +
  labs(x = "Death Year", y = "Scaled Bill Value",
    title = "Does death year affect bill value?", shape = "Gender")
```

```
Warning: Removed 9 rows containing missing values or values outside the scale range
(`geom_point()`).
```



3.3.5 Saving Plots

In ggplot2, use the `ggsave` function to save the most recent plot you created:

```
ggsave("banknote_scatter.png")
```

The file format is selected automatically based on the extension. Common formats are PNG and PDF.

The Plot Device

You can also save a plot with one of R's "plot device" functions. The steps are:

1. Call a plot device function: `png`, `jpeg`, `pdf`, `bmp`, `tiff`, or `svg`.
2. Run your code to make the plot.
3. Call `dev.off` to indicate that you're done plotting.

This strategy works with any of R's graphics systems (not just ggplot2).

Here's an example:

```r
# Run these lines in the console, not the notebook!
jpeg("banknote_scatter.jpeg")
ggplot(banknotes) +
  geom_point() +
  aes(x = death_year, y = scaled_bill_value, shape = gender) +
  labs(x = "Death Year", y = "Scaled Bill Value",
    title = "Does death year affect bill value?", shape = "Gender")
dev.off()
```

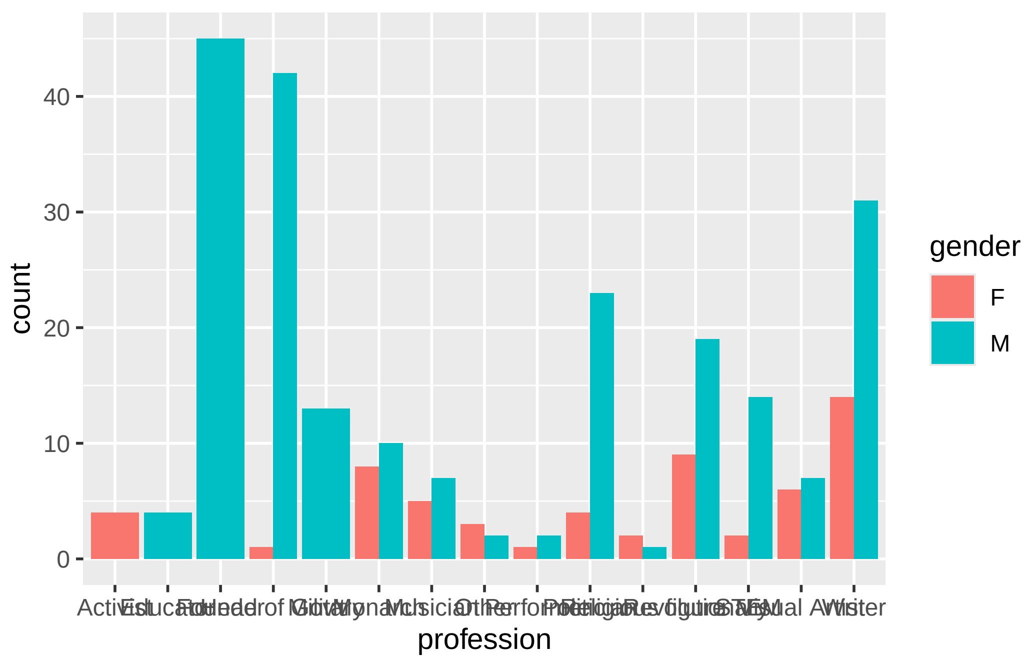
3.3.6 Example: Bar Plot

Now suppose you want to plot the number of banknotes with people from each profession in the banknotes data set. A bar plot is in an appropriate way to represent this visually.

The geometry for a bar plot is `geom_bar`. Since bar plots are mainly used to display frequencies, the `geom_bar` function automatically computes frequencies when given mapped to a categorical feature.

We can also use a fill color to further breakdown the bars by gender. Here's the code to make the bar plot:

```r
ggplot(banknotes) +
  aes(x = profession, fill = gender) +
  geom_bar(position = "dodge")
```



The setting `position = "dodge"` instructs `geom_bar` to put the bars side-by-side rather than stacking them.

In some cases, you may want to make a bar plot with frequencies you've already computed. To prevent `geom_bar` from computing frequencies automatically, set `stat = "identity"`.

3.3.7 Visualization Design

Designing high-quality visualizations goes beyond just mastering which R functions to call. You also need to think carefully about what kind of data you have and what message you want to convey. This section provides a few guidelines.

The first step in data visualization is choosing an appropriate kind of plot. Here are some suggestions (not rules):

Feature 1	Feature 2	Plot
categorical		bar, dot
categorical	categorical	bar, dot, mosaic
numerical		box, density, histogram
numerical	categorical	box, density, ridge
numerical	numerical	line, scatter, smooth scatter

If you want to add a:

- 3rd numerical feature, use it to change point/line sizes.
- 3rd categorical feature, use it to change point/line styles.
- 4th categorical feature, use side-by-side plots.

Once you've selected a plot, here are some rules you should almost always follow:

- Always add a title and axis labels. These should be in plain English, not variable names!
- Specify units after the axis label if the axis has units. For instance, "Height (ft)".
- Don't forget that many people are colorblind! Also, plots are often printed in black and white. Use point and line styles to distinguish groups; color is optional.
- Add a legend whenever you've used more than one point or line style.
- Always write a few sentences explaining what the plot reveals. Don't describe the plot, because the reader can just look at it. Instead, explain what they can learn from the plot and point out important details that are easily overlooked.
- Sometimes points get plotted on top of each other. This is called *overplotting*. Plots with a lot of overplotting can be hard to read and can even misrepresent the data by hiding how many points are present. Use a two-dimensional density plot or jitter the points to deal with overplotting.
- For side-by-side plots, use the same axis scales for both plots so that comparing them is not deceptive.

Visualization design is a deep topic, and whole books have been written about it. One resource where you can learn more is DataLab's Principles of Data Visualization Workshop Reader.

3.4 Apply Functions

Section 2.1.3 introduced vectorization, a convenient and efficient way to compute multiple results. That section also mentioned that some of R's functions—the ones that summarize or aggregate data—are not vectorized.

The `class` function is an example of a function that's not vectorized. If we call the `class` function on the banknotes data set, we get just one result for the data set as a whole:

```r
class(banknotes)
```

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

What if we want to get the class of each column? We can get the class for a single column by selecting the column with `$`, the dollar sign operator:

```
class(banknotes$currency_code)
```

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

But what if we want the classes for all the columns? We could write a call to `class` for each column, but that would be tedious. When you're working with a programming language, you should try to avoid tedium; there's usually a better, more automated way.

Section 2.2.1 pointed out that data frames are technically lists, where each column is one element. With that in mind, what we need here is a line of code that calls `class` on each element of the data frame. The idea is similar to vectorization, but since we have a list and a non-vectorized function, we have to do a bit more than just call `class(banknotes)`.

The `lapply` function calls, or *applies*, a function on each element of a list or vector. The syntax is:

```
lapply(X, FUN, ...)
```

The function `FUN` is called once for each element of `X`, with the element as the first argument. The `...` is for additional arguments to `FUN`, which are held constant across all the elements.

Let's try this out with the banknotes data and the `class` function:

```
lapply(banknotes, class)
```

```
$currency_code
[1] "character"

$country
[1] "character"

$currency_name
[1] "character"

$name
[1] "character"

$gender
[1] "character"
```

```
$bill_count
[1] "numeric"

$profession
[1] "character"

$known_for_being_first
[1] "character"

$current_bill_value
[1] "integer"

$prop_total_bills
[1] "numeric"

$first_appearance_year
[1] "integer"

$death_year
[1] "numeric"

$comments
[1] "character"

$hover_text
[1] "character"

$has_portrait
[1] "character"

$id
[1] "character"

$scaled_bill_value
[1] "numeric"
```

The result is similar to if the `class` function was vectorized. In fact, if we use a vector and a vectorized function with `lapply`, the result is nearly identical to the result from vectorization:

```r
x = c(1, 2, pi)

sin(x)
```

```
[1] 8.414710e-01 9.092974e-01 1.224647e-16
```

```r
lapply(x, sin)
```

```
[[1]]
[1] 0.841471

[[2]]
[1] 0.9092974

[[3]]
[1] 1.224647e-16
```

The only difference is that the result from `lapply` is a list. In fact, the `lapply` function always returns a list with one element for each element of the input data. The "l" in `lapply` stands for "list".

The `lapply` function is one member of a family of functions called *apply functions*. All of the apply functions provide ways to apply a function repeatedly to different parts of a data structure. We'll meet a few more apply functions soon.

When you have a choice between using vectorization or an apply function, you should always choose vectorization. Vectorization is clearer—compare the two lines of code above—and it's also significantly more efficient. In fact, vectorization is the most efficient way to call a function repeatedly in R.

As we saw with the `class` function, there are some situations where vectorization is not possible. That's when you should think about using an apply function.

3.4.1 The `sapply` Function

The related `sapply` function calls a function on each element of a list or vector, and simplifies the result. That last part is the crucial difference compared to `lapply`. When results from the calls all have the same type and length, `sapply` returns a vector or matrix instead of a list. When the results have different types or lengths, the result is the same as for `lapply`. The "s" in `sapply` stands for "simplify".

For instance, if we use `sapply` to find the classes of the columns in the banknotes data, we get a character vector:

```r
sapply(banknotes, class)
```

```
      currency_code               country         currency_name 
        "character"           "character"           "character" 
               name                gender            bill_count 
        "character"           "character"             "numeric" 
         profession known_for_being_first    current_bill_value 
        "character"           "character"             "integer" 
   prop_total_bills first_appearance_year            death_year 
          "numeric"             "integer"             "numeric" 
           comments            hover_text          has_portrait 
        "character"           "character"           "character" 
                 id     scaled_bill_value 
        "character"             "numeric" 
```

Likewise, if we use `sapply` to compute the `sin` values, we get a numeric vector, the same as from vectorization:

```r
sapply(x, sin)
```

```
[1] 8.414710e-01 9.092974e-01 1.224647e-16
```

In spite of that, vectorization is still more efficient than `sapply`, so use vectorization instead when possible.

Apply functions are incredibly useful for summarizing data. For example, suppose we want to compute the frequencies for all of the columns in the banknotes data set that aren't numeric.

First, we need to identify the columns. One way to do this is with the `is.numeric` function. Despite the name, this function actually tests whether its argument is a real number, not whether it its argument is a numeric vector. In other words, it also returns true for integer values. We can use `sapply` to apply this function to all of the columns in the banknotes data set:

```r
is_not_number = !sapply(banknotes, is.numeric)
is_not_number
```

```
      currency_code               country         currency_name 
               TRUE                  TRUE                  TRUE 
               name                gender            bill_count 
               TRUE                  TRUE                 FALSE 
```

```
          profession known_for_being_first     current_bill_value
                TRUE                  TRUE                  FALSE
    prop_total_bills first_appearance_year             death_year
               FALSE                 FALSE                  FALSE
            comments            hover_text           has_portrait
                TRUE                  TRUE                   TRUE
                  id     scaled_bill_value
                TRUE                 FALSE
```

Is it worth using R code to identify the non-numeric columns? Since there are only 17 columns in the banknotes data set, maybe not. But if the data set was larger, with say 100 columns, it definitely would be.

In general, it's a good habit to use R to do things rather than do them manually. You'll get more practice programming, and your code will be more flexible if you want to adapt it to other data sets.

Now that we know which columns are non-numeric, we can use the `table` function to compute frequencies. We only want to compute frequencies for those columns, so we need to subset the data:

```r
lapply(banknotes[, is_not_number], table)
```

```
$currency_code

ARS AUD BDT BOB CAD CLP COP CRC CVE CZK DOP GBP GEL IDR ILS ISK JMD JPY KGS KRW
  5   9   9  15   9   5  11   6   5   6  11   8   9  12   4   5   5   3   7   4
MWK MXN NGN NZD PEN PGK PHP RMB RSD SEK STD TND TRY UAH USD UYU VES VES ZAR
  7  11   9   5   7   1   9   6   9   6   8   4  12  10   7   7   2   6   5

$country

           Argentina             Australia             Bangladesh
                   5                     9                      9
             Bolivia                Canada             Cape Verde
                  15                     9                      5
               Chile                 China               Colombia
                   5                     6                     11
          Costa Rica        Czech Republic     Dominican Republic
                   6                     6                     11
             England               Georgia                Iceland
                   8                     9                      5
           Indonesia                Israel                Jamaica
```

```
                  12                     4                     5
               Japan            Kyrgyzstan                Malawi
                   3                     7                     7
              Mexico           New Zealand               Nigeria
                  11                     5                     9
    Papua New Guinea                  Peru           Philippines
                   1                     7                     9
São Tomé and Príncipe               Serbia          South Africa
                   8                     9                     5
         South Korea                Sweden               Tunisia
                   4                     6                     4
              Turkey               Ukraine         United States
                  12                    10                     7
             Uruguay             Venezuela
                   7                     8

$currency_name

  Argentinian Peso  Australian Dollar          Boliviano    Canadian Dollar
                 5                  9                 15                  9
      Chilean Peso     Colombian Peso              Colón       Czech koruna
                 5                 11                  6                  6
             dobra            Escudo              hryvna    Jamaican dollar
                 8                  5                 10                  5
              kina              Króna             Kwacha               lari
                 1                  5                  7                  9
              Lira              naira New Zealand dollar     Peso dominicano
                12                  9                  5                 11
     Peso mexicano      Peso uruguayo               piso              pound
                11                  7                  9                  8
              rand          Renminbi              rupiah      Serbian dinar
                 5                  6                 12                  9
            shekel                Sol                Som       Swedish krona
                 4                  7                  7                  6
              Taka     Tunisian dinar          US dollar Venezuelan bolivar
                 9                  4                  7                  8
               Won                Yen
                 4                  3

$name

               Abou el Kacem Chebbi                     Abraham Lincoln
                                  1                                   1
```

Abraham Valdelomar Pinto	Agnes Macphail
1	1
Akaki Tsereteli	Alan Turing
1	1
Alejo Calatayud	Alexander Hamilton
1	1
Alfonso López Michelsen	Alfredo González Flores
1	1
Alfredo Vásquez Acevedo	Alhaji Aliyu Mai-Bornu
1	1
Alhaji Sir Abubarkar Tafawa Balewa	Alhaji Sir Ahmadu Bello
1	1
Alykul Osmonov	Andrés Bello
1	1
Andrew Jackson	Antonio José de Sucre
1	2
Apiaguaiki Tüpa	Apirana Ngata
1	1
Aristides Maria Pereira	Arturo Prat Chacón
1	1
Astrid Lindgren	Aydın Sayılı
1	1
Bangabandhu Sheikh Mujibur Rahman	Banjo Paterson
9	1
Bartolina Sisa	Benigno S. Aquino, Jr.
1	1
Benito Juárez	Benjamin Franklin
2	1
Birgit Nilsson	Bogdan Khmelnitsky
1	1
Božena Němcová	Braulio Carrillo Colina
1	1
Bruno Racua	Brynjólfur Sveinsson
1	1
Cahit Arf	Carlos Lleras Restrepo
1	1
Carmen Serdán	Cesária Évora
1	1
Charles IV	Chief Obafemi Awolowo
1	1
Codé Di Dona	Corazon C. Aquino
1	1
Dag Hammarskiöld	Dámaso Antonio Larrañaga

1
David Unaipon
1
Diego Rivera
1
Djuanda Kartawidjaja
1
Dorde Vajfert
1
Dr. Clement Isong
1
Dr. Nnamdi Azikiwe
1
Edmund Hillary
1
Ema Destinnová
1
Eustaquio Méndez
1
Evert Taube
1
Farhat Hached
1
Francisco de Miranda
1
Francisco del Rosario Sánchez
1
Francisco José Tenreiro
1
František Palacký
1
Gabriel García Márquez
1
General Murtala Mohammed
1
George Washington
1
Greta Garbo
1
Gusti Ngurah Rai
1
Henrique Teixeira de Sousa
1

1
Debora Arango
1
Diosdado P. Macapagal
1
Domingo Faustino Sarmiento
1
Dr. Alvan Ikoku
1
Dr. Hastings Kamuzu Banda
1
Edith Cowan
1
Eduardo Fabini
1
Emilio Prud'Homme
1
Eva Perón
1
Ezequiel Zamora
1
Fatma Aliye Topuz
1
Francisco de Paula Santander
1
Francisco I. Madero
1
Frans Kaisiepo
1
Frida Kahlo
1
Gabriela Mistral
1
Genoveva Ríos
1
Gregorio Luperon
1
GSSJ Ratulangi
1
Hannibal
1
Hermila Galindo
1

Hideyo Noguchi	Hryhoriy Skovoroda
1	1
Hugh Lawson Shearer	Ichiyo Higuchi
1	1
Idham Chalid	Ignacio Carrera Pinto
1	1
Ilia Chavchavadze	Ingmar Bergman
1	1
Inkosi ya Makhosi M'mbelwa II	Inkosi Ya Mokhosi Gomani II
1	1
Itri (Buhurizade Mustafa Efendi)	Ivan Franko
1	1
Ivan Mazepa	Ivane Javakhishvili
1	1
J.M.W. Turner	James Federick Sangala
1	1
James Gladstone (Akay-na-muka)	Jane Austen
1	1
Jóhannes S. Kjarval	John A. Macdonald
1	1
John Amos Comenius	Jón Sigurdsson
1	1
Jónas Hallgrímsson	Jorge Barbosa
1	1
Jorge Basadre Grohmann	Jorge Eliécer Gaitán
1	1
Jorge Isaacs	José Abad Santos
1	1
José Abelardo Quiñones Gonzales	José Asunción Silva
1	1
José Félix Ribas	José Figueres
1	1
José Manuel Baca "Cañoto"	José María Morelos
1	1
José Pedro Varela	José Rufino Reyes y Siancas
1	1
José Santos Vargas	Josefa Camejo
1	1
Josefa Llanes Escoda	Jovan Cvijic
1	1
Juan Manuel de Rosas	Juan Pablo Duarte
1	1
Juan Zorrilla de San Martín	Juana Azurduy de Padilla

1	1
Juana de Ibarbourou	Julio Argentino Roca
1	1
Julio Garavito Armero	Jusup Balasagyn
1	1
Kaikhosro (Kakutsa) Cholokashvili	Kate Sheppard
1	1
Kemal Atatürk	Kemaleddin
6	1
King David IV	King Sejong the Great
1	1
Kurmanjan Datka	Ladi Kwali
1	1
Leah Goldberg	Lesya Ukrainka
1	1
Manuel A. Roxas	Manuel Belgrano
1	1
Manuel L. Quezon	Manuel Rodríguez Erdoyza
1	1
Mao Zedong	María Isabel Carvajal (Carmen Lyra)
6	1
María Isabel Granda y Larco	María Teresa Mirabal
1	1
Mary Gilmore	Mary Reibey
1	1
Matías Ramón Mella	Mauro Fernández Acuña
1	1
Michael Norman Manley	Michael Somare
1	1
Miguel Hidalgo	Milutin Milanković
1	1
Minerva Mirabal	Mohammad Hatta
1	1
Mohammad Husni Thamrin	Mykhailo Hrusheskyi
1	1
Nadežda Petrović	Nanny of the Maroons
1	1
Nathan Alterman	Nellie Melba
1	1
Nelson Mandela	Nezahualcóyotl
5	1
Niko Pirosmanashvili	Nikola Tesla
1	1

Oto Iskander Di Nata	1	Pablo Zárate Willka	1
Patria Mirabal	1	Pedro Figari	1
Pedro Henriquez Ureña	1	Pedro Ignacio Muiba	1
Pedro Paulet	1	Petar Petrovic Njegoš	1
Policarpa Salavarrieta	1	Prince Antasari	1
Queen Elizabeth II	7	Queen Tamar	1
Rachel Bluwstein	1	Rafael Urdaneta	1
Ragnheiður Jónsdóttir	1	Raúl Porras Barrenechea	1
Rei Amador	7	Reverend John Chilembwe	2
Reverend John Flynn	1	Ricardo Jiménez Oreamuno	1
Rose Lomathinda Chibambo	1	Rutherford of Nelson	1
Salomé Ureña	1	Samuel Sharpe	1
Santa Rosa de Lima	1	Sayakbay Karalaev	1
Sergio Osmeña	1	Shaul Tchernichovsky	1
Shin Saimdang	1	Shota Rustaveli	1
Simón Bolívar	2	Simón Rodrigues	1
Sir Donald Sangster	1	Sir George-Étienne Cartier	1
Sir John Monash	1	Sir Robert Borden	1
Sir Wilfrid Laurier	1	Slobodan Jovanovic	1
Soekarno	1	Sor Juana Inés de la Cruz	1
Stevan Stevanovic Mokranjac	1	Sultan Mahmud Badaruddin II	1
Suymonkul Chokmorov		Taras Shevchenko	

```
                    1                                   1
     Tewhida Ben Sheikh                   Thomas Jefferson
                    1                                   1
            Tjut Meutia                      Togolok Moldo
                    1                                   1
    Toktogul Satylganov            Tomáš Garrigue Masaryk
                    1                                   1
           Tomás Katari    Tupac Katari (Julian Apasa Nina)
                    1                                   1
       Ulysses S. Grant                         Vicente Lim
                    1                                   1
          Viola Desmond                 Virginia Gutiérrez
                    1                                   1
    Volodymyr the Great               Volodymyr Vernadskyi
                    1                                   1
Vuk Stefanovic Karadžic         William Lyon Mackenzie King
                    1                                   1
      Winston Churchill                   Yaroslav the Wise
                    1                                   1
               Yi Hwang                                Yi I
                    1                                   1
       Yukichi Fukuzawa                          Yunus Emre
                    1                                   1
    Zakaria Paliashvili
                    1

$gender

  F   M
 59 220

$profession

        Activist         Educator          Founder    Head of Gov't
               4                4               45               43
        Military          Monarch         Musician            Other
              13               18               12                5
       Performer       Politician Religious figure    Revolutionary
               3               27                3               28
            STEM    Visual Artist           Writer
              16               13               45

$known_for_being_first
```

```
 No Yes
185  94

$comments
```

```
canadian prime minister of canada
```

```
president
```

1s

Alleged first person to
propellant rocket engine and modern rocket propulsion system

entitled Datka (meaning "Righteou

day Colombia)

First

first Unite

Founder of Bangladesh wh

Founder of the country's first Montessori school. She was a co-founder of the Communist Part

Given

He also wrote the first-ever programming manual, and his programming system was used in t

he is c

known for car

organized meetings for sla

Sh

Shares with another p

Shares with another person. In 1927 when his book of Aboriginal legends, Hurgarrda was

Shares with another person. She became the first person to be app

Shares with another person. She is

Shares with another person. The fi

story writer

was acting President and
in-chief for a short time, and Chief Justice of the Supreme Court

Wrote the first Costa Ri

$hover_text

Alleged first person to build a liquid-
propellant rocket engine and modern rocket propulsion system

Composed

First Australian to ac

I

First a

Firs

Fi

First

Canadian Prime Minister of Canada

President

story writer

First person appointed Dame Comman

First preside
day Colombia)

First to promote and write

First US Pos

First Visayan person to be President of

Founded Natio

Founded Roy

Given the honorary title

J

```
                                                                                                           Wrote firs
                                                                Wrote the first Costa Rican General Code wh
                                                                                                                   Wr
Wrote the first-ever programming manual, and his programming system was used in the Ferranti


$has_portrait

false  true
  155   124

$id

   ARS_Argentino     ARS_Belgrano      ARS_Domingo        ARS_Evita
               1                1                1                1
       ARS_Rosas        AUD_Banjo        AUD_Edith    AUD_Elizabeth
               1                1                1                7
       AUD_Flynn      AUD_Gilmore       AUD_Monash       AUD_Nellie
               1                1                1                1
      AUD_Reibey      AUD_Unaipon      BDT_Rahman        BOB_Alejo
               1                1                9                1
     BOB_Azurduy    BOB_Bartolina      BOB_Bolivar        BOB_Bruno
               1                1                2                1
      BOB_Canoto   BOB_Eustaquino      BOB_Ignacio       BOB_Katari
               1                1                1                1
        BOB_Rios       BOB_Santos        BOB_Sucre         BOB_Tupa
               1                1                2                1
       BOB_Tupac       BOB_Zarate        CAD_Agnes       CAD_Borden
               1                1                1                1
     CAD_Cartier    CAD_Gladstone      CAD_Laurier         CAD_Lyon
               1                1                1                1
   CAD_Macdonald        CAD_Viola       CLP_Andres       CLP_Chacon
               1                1                1                1
     CLP_Erdoyza      CLP_Ignacio      CLP_Mistral      COP_Alfonso
               1                1                1                1
    COP_Asuncion       COP_Debora      COP_Gabriel       COP_Gaitan
```

```
              1                 1                 1                 1
  COP_Garavito        COP_Isaacs        COP_Lleras    COP_Policarpa
              1                 1                 1                 1
 COP_Santander      COP_Virginia         CRC_Acuna     CRC_Carrillo
              1                 1                 1                 1
  CRC_Carvajal      CRC_Figueres      CRC_Gonzalez      CRC_Ricardo
              1                 1                 1                 1
   CVE_Barbosa       CVE_Cesaria          CVE_Code     CVE_Henrique
              1                 1                 1                 1
   CVE_Pereira          CZK_Amos        CZK_Bozena      CZK_Charles
              1                 1                 1                 1
       CZK_Ema     CZK_Frantisek      CZK_Garrigue       DOP_Duarte
              1                 1                 1                 1
    DOP_Emilio       DOP_Luperon         DOP_Maria       DOP_Matias
              1                 1                 1                 1
   DOP_Minerva        DOP_Patria       DOP_Rosario       DOP_Rufino
              1                 1                 1                 1
    DOP_Salome         DOP_Urena        GBP_Austen    GBP_Churchill
              1                 1                 1                 1
    GBP_Turing        GBP_Turner         GEL_Akaki        GEL_David
              1                 1                 1                 1
      GEL_Ilia         GEL_Ivane       GEL_Kakutsa         GEL_Niko
              1                 1                 1                 1
     GEL_Shota         GEL_Tamar       GEL_Zakaria      IDR_Djuanda
              1                 1                 1                 1
     IDR_Frans          IDR_GSSJ         IDR_Gusti        IDR_Hatta
              1                 1                 1                 1
     IDR_Husni         IDR_Idham           IDR_Oto       IDR_Prince
              1                 1                 1                 1
  IDR_Soekarno        IDR_Sultan          IDR_Tjut     ILS_Goldberg
              1                 1                 1                 1
    ILS_Nathan        ILS_Rachel         ILS_Shaul ISK_Hallgrimsson
              1                 1                 1                 1
   ISK_Kjarval    ISK_Ragnheiour    ISK_Sigurdsson    ISK_Sveinsson
              1                 1                 1                 1
      JMD_Hugh         JMD_Nanny        JMD_Norman     JMD_Sangster
              1                 1                 1                 1
    JMD_Sharpe        JPY_Hideyo        JPY_Ichiyo      JPY_Yukichi
              1                 1                 1                 1
 KGS_Balasagyn         KGS_Datka       KGS_Osmonov     KGS_Sayakbay
              1                 1                 1                 1
 KGS_Suymonkul       KGS_Togolok      KGS_Toktogul        KRW_Hwang
              1                 1                 1                 1
```

```
            KRW_I       KRW_Sejong         KRW_Shin    MWK_Chilembwe
                1                1                1                2
       MWK_Gomani     MWK_Hastings      MWK_Mmbelwa         MWK_Rose
                1                1                1                1
      MWK_Sangala       MXN_Benito        MXN_Frida      MXN_Hermila
                1                2                1                1
      MXN_Hidalgo        MXN_Juana       MXN_Madero      MXN_Morelos
                1                1                1                1
         MXN_Neza       MXN_Rivera       MXN_Serdan        NGN_Aliyu
                1                1                1                1
        NGN_Alvan       NGN_Balewa        NGN_Bello      NGN_Clement
                1                1                1                1
        NGN_Kwali      NGN_Murtala       NGN_Nnamdi      NGN_Obafemi
                1                1                1                1
       NZD_Edmund        NZD_Ngata   NZD_Rutherford     NZD_Sheppard
                1                1                1                1
     PEN_Abelardo      PEN_Basadre       PEN_Granda       PEN_Paulet
                1                1                1                1
        PEN_Pinto         PEN_Raul        PEN_Santa       PGK_Somare
                1                1                1                1
         PHP_Abad      PHP_Benigno      PHP_Corazon     PHP_Diosdado
                1                1                1                1
       PHP_Llanes       PHP_Osmena       PHP_Quezon        PHP_Roxas
                1                1                1                1
      PHP_Vicente          RMB_Mao       RSD_Cvijic        RSD_Dorde
                1                6                1                1
      RSD_Milutin       RSD_Njegos     RSD_Petrovic     RSD_Slobodan
                1                1                1                1
   RSD_Stevanovic        RSD_Tesla          RSD_Vuk          SEK_Dag
                1                1                1                1
        SEK_Evert        SEK_Greta       SEK_Ingmar     SEK_Lindgren
                1                1                1                1
      SEK_Nilsson          STD_Rei     STD_Tenreiro       TND_Chebbi
                1                7                1                1
       TND_Farhat     TND_Hannibal      TND_Tewhida          TRY_Arf
                1                1                1                1
      TRY_Ataturk        TRY_Fatma         TRY_Itri   TRY_Kemaleddin
                6                1                1                1
       TRY_Sayili        TRY_Yunus       UAH_Bogdon       UAH_Franko
                1                1                1                1
        UAH_Great       UAH_Mazepa     UAH_Mykhailo    UAH_Skovoroda
                1                1                1                1
        UAH_Taras     UAH_Ukrainka   UAH_Vernadskyi     UAH_Yaroslav
```

```
             1                1                1                1
  USD_Franklin     USD_Hamilton      USD_Jackson    USD_Jefferson
             1                1                1                1
   USD_Lincoln      USD_Ulysses   USD_Washington      UYU_Eduardo
             1                1                1                1
    UYU_Figari   UYU_Ibarbourou    UYU_Larranaga       UYU_Varela
             1                1                1                1
   UYU_Vasquez     UYU_Zorrilla       VES_Camejo     VES_Ezequiel
             1                1                1                1
   VES_Miranda       VES_Rajael        VES_Ribas    VES_Rodrigues
             1                1                1                1
   ZAR_Mandela
             5
```

We use `lapply` rather than `sapply` for this step because the table for each column will have a different length (but try `sapply` and see what happens!).

3.4.2 The Split-Apply Pattern

In a data set with categorical features, it's often useful to compute something for each category. The `lapply` and `sapply` functions can compute something for each element of a data structure, but categories are not necessarily elements.

For example, the banknotes data set has 38 different categories in the `country` column. If we want all of the rows for one country, one way to get them is by indexing:

```r
usa = banknotes[banknotes$country == "United States", ]
head(usa)
```

```
    currency_code       country currency_name              name gender
253           USD United States     US dollar  Ulysses S. Grant      M
254           USD United States     US dollar  Thomas Jefferson      M
255           USD United States     US dollar   Abraham Lincoln      M
256           USD United States     US dollar George Washington      M
257           USD United States     US dollar    Andrew Jackson      M
258           USD United States     US dollar Benjamin Franklin      M
    bill_count    profession known_for_being_first current_bill_value
253          1 Head of Gov't                    No                 50
254          1       Founder                   Yes                  2
255          1 Head of Gov't                    No                  5
256          1       Founder                   Yes                  1
257          1 Head of Gov't                    No                 20
```

```
258          1       Founder                   Yes                 100
    prop_total_bills first_appearance_year death_year
253             0.05                  1913       1885
254             0.03                  1869       1826
255             0.06                  1914       1865
256             0.26                  1869       1799
257             0.23                  1928       1845
258             0.33                  1914       1790
                                                                                       comments
253
254                                                                  1st US secretary of state
255
256                                                                       1st president of USA
257
258 first United States Postmaster General, first United States Ambassador to France, etc...
                                                                      hover_text
253
254                                                   First US Secretary of State
255
256                                                            First US President
257
258 First US Postmaster General, First United States Ambassador to France
    has_portrait             id scaled_bill_value
253         true    USD_Ulysses        0.49494949
254         true  USD_Jefferson        0.01010101
255         true    USD_Lincoln        0.04040404
256         true USD_Washington        0.00000000
257        false    USD_Jackson        0.19191919
258         true   USD_Franklin        1.00000000
```

To get all 38 countries separately, we'd have to do this 38 times. If we want to compute something for each country, say the mean of the `first_appearance_year` column, we also have to repeat that computation 38 times. Here's what it would look like for just the United States:

```r
mean(usa$first_appearance_year)
```

```
[1] 1905.143
```

If the categories were elements, we could avoid writing code to index each category, and just use the `sapply` (or `lapply`) function to apply the `mean` function to each.

The `split` function splits a vector or data frame into groups based on a vector of categories. The first argument to `split` is the data, and the second argument is a congruent vector of categories.

We can use `split` to elegantly compute means of `first_appearance_year` broken down by country. First, we split the data by country. Since we only want to compute on the `first_appearance_year` column, we only split that column:

```r
by_country = split(banknotes$first_appearance_year, banknotes$country)
class(by_country)
```

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

```r
names(by_country)
```

```
 [1] "Argentina"             "Australia"             "Bangladesh"
 [4] "Bolivia"               "Canada"                "Cape Verde"
 [7] "Chile"                 "China"                 "Colombia"
[10] "Costa Rica"            "Czech Republic"        "Dominican Republic"
[13] "England"               "Georgia"               "Iceland"
[16] "Indonesia"             "Israel"                "Jamaica"
[19] "Japan"                 "Kyrgyzstan"            "Malawi"
[22] "Mexico"                "New Zealand"           "Nigeria"
[25] "Papua New Guinea"      "Peru"                  "Philippines"
[28] "São Tomé and Príncipe" "Serbia"                "South Africa"
[31] "South Korea"           "Sweden"                "Tunisia"
[34] "Turkey"                "Ukraine"               "United States"
[37] "Uruguay"               "Venezuela"
```

The result from `split` is a list with one element for each category. The individual elements contain pieces of the original `first_appearance_year` column:

```r
head(by_country$Mexico)
```

```
[1] 2010 2010 1979 2020 2020 1973
```

Since the categories are elements in the split data, now we can use `sapply` the same way we did in previous examples:

```r
sapply(by_country, mean)
```

```
           Argentina            Australia           Bangladesh
            1992.200             1990.778             1972.000
             Bolivia               Canada           Cape Verde
            2010.667             1983.778             2014.000
               Chile                China             Colombia
            1988.600             1980.000             2001.727
          Costa Rica       Czech Republic   Dominican Republic
            2002.667             1993.500             1991.545
             England              Georgia              Iceland
            1994.000             1997.222             1991.800
           Indonesia               Israel              Jamaica
            2004.750             2015.750             1989.000
               Japan           Kyrgyzstan               Malawi
            1997.333             1998.714             2001.857
              Mexico          New Zealand              Nigeria
            1988.545             1987.000             1994.667
    Papua New Guinea                 Peru          Philippines
            1989.000             2000.143             1982.111
São Tomé and Príncipe               Serbia         South Africa
            1981.500             2005.111             2012.000
         South Korea               Sweden              Tunisia
            1982.250             2015.167             2015.750
              Turkey              Ukraine        United States
            1968.000             1997.000             1905.143
             Uruguay            Venezuela
            1998.429             1991.500
```

This two-step process is an R idiom called the *split-apply pattern*. First you use `split` to convert categories into list elements, then you use an apply function to compute something on each category. Any time you want to compute results by category, you should think of this pattern.

The split-apply pattern is so useful that R provides the `tapply` function as a shortcut. The `tapply` function is equivalent to calling `split` and then `sapply`. Like `split`, the first argument is the data and the second argument is a congruent vector of categories. The third argument is a function to apply, like the function argument in `sapply`.

We can use `tapply` to compute the `first_appearance_year` means by `country` for the banknotes data set:

```r
tapply(banknotes$first_appearance_year, banknotes$country, mean)
```

```
            Argentina             Australia            Bangladesh 
             1992.200              1990.778              1972.000 
              Bolivia                Canada            Cape Verde 
             2010.667              1983.778              2014.000 
                Chile                 China              Colombia 
             1988.600              1980.000              2001.727 
           Costa Rica        Czech Republic    Dominican Republic 
             2002.667              1993.500              1991.545 
              England               Georgia               Iceland 
             1994.000              1997.222              1991.800 
            Indonesia                Israel               Jamaica 
             2004.750              2015.750              1989.000 
                Japan            Kyrgyzstan                Malawi 
             1997.333              1998.714              2001.857 
               Mexico           New Zealand               Nigeria 
             1988.545              1987.000              1994.667 
     Papua New Guinea                  Peru           Philippines 
             1989.000              2000.143              1982.111 
São Tomé and Príncipe                Serbia          South Africa 
             1981.500              2005.111              2012.000 
          South Korea                Sweden               Tunisia 
             1982.250              2015.167              2015.750 
               Turkey               Ukraine         United States 
             1968.000              1997.000              1905.143 
              Uruguay             Venezuela 
             1998.429              1991.500 
```

Notice that the result is identical to the one we computed before.

The "t" in `tapply` stands for "table", because the `tapply` function is a generalization of the `table` function. If you use `length` as the third argument to `tapply`, you get the same results as you would from using the `table` function on the category vector.

The `aggregate` function is closely related to `tapply`. It computes the same results, but organizes them into a data frame with one row for each category. In some cases, this format is more convenient. The arguments are the same, except that the second argument must be a list or data frame rather than a vector.

As an example, here's the result of using `aggregate` to compute the `first_appearance_year` means:

```r
aggregate(banknotes$first_appearance_year, list(banknotes$country), mean)
```

```
                 Group.1        x
1              Argentina 1992.200
2              Australia 1990.778
3             Bangladesh 1972.000
4                Bolivia 2010.667
5                 Canada 1983.778
6             Cape Verde 2014.000
7                  Chile 1988.600
8                  China 1980.000
9               Colombia 2001.727
10            Costa Rica 2002.667
11        Czech Republic 1993.500
12    Dominican Republic 1991.545
13               England 1994.000
14               Georgia 1997.222
15               Iceland 1991.800
16             Indonesia 2004.750
17                Israel 2015.750
18               Jamaica 1989.000
19                 Japan 1997.333
20            Kyrgyzstan 1998.714
21                Malawi 2001.857
22                Mexico 1988.545
23           New Zealand 1987.000
24               Nigeria 1994.667
25      Papua New Guinea 1989.000
26                  Peru 2000.143
27           Philippines 1982.111
28 São Tomé and Príncipe 1981.500
29                Serbia 2005.111
30          South Africa 2012.000
31           South Korea 1982.250
32                Sweden 2015.167
33               Tunisia 2015.750
34                Turkey 1968.000
35               Ukraine 1997.000
36         United States 1905.143
37               Uruguay 1998.429
38             Venezuela 1991.500
```

The `lapply`, `sapply`, and `tapply` functions are the three most important functions in the family of apply functions, but there are many more. You can learn more about all of R's apply functions by reading this StackOverflow post.

3.5 Exercises

3.5.1 Exercise

Compute the number of banknotes that feature a person who died before 1900.

3.5.2 Exercise

Compute the range of `first_appearance_year` for each country.

3.5.3 Exercise

1. Compute the set of banknotes with people who died in this century.

   Hint: be careful of missing values in the `death_year` column!

```r
is21 = !is.na(banknotes$death_year) & banknotes$death_year >= 2000
people21 = banknotes[is21, ]
```

2. Use ggplot2's `geom_segment` function to create a plot which shows the timespan between death year and first appearance as a horizontal segment, for each banknote in the result from step 1. Put the name of each person on the y-axis. Color code the segments by gender.

   Hint: You can make the plot more visually appealing if you first sort the data by death year. You can use the `order` function to get indexes that will sort the rows of a data frame according to some column.

4 Organizing Code

Learning Goals

After completing this chapter, learners should be able to:

- Create code that only runs when a condition is satisfied
- Create custom functions in order to organize and reuse code

By now, you've learned all of the basic skills necessary to explore a data set in R. The focus of this chapter is how to organize your code so that it's concise, clear, and easy to automate. This will help you and your collaborators avoid tedious, redundant work, reproduce results efficiently, and run code in specialized environments for scientific computing, such as high-performance computing clusters.

4.1 Conditional Expressions

Sometimes you'll need code to do different things, depending on a condition. **If-expressions** provide a way to write conditional code.

For example, suppose we want to greet one person differently from the others:

```r
name = "Nick"
if (name == "Nick") {
   # If name is Nick:
   message("We went down the TRUE branch")
   msg = "Hi Nick, nice to see you again!"
} else {
   # Anything else:
   msg = "Nice to meet you!"
}
```

```
We went down the TRUE branch
```

Indent code inside of the if-expression by 2 or 4 spaces. Indentation makes your code easier to read.

The condition in an if-expression has to be a scalar:

```r
name = c("Nick", "Susan")
if (name == "Nick") {
   msg = "Hi Nick!"
} else {
   msg = "Nice to meet you!"
}
```

```
Error in if (name == "Nick") {: the condition has length > 1
```

You can chain together if-expressions:

```r
name = "Susan"
if (name == "Nick") {
   msg = "Hi Nick, nice to see you again!"
} else if (name == "Peter") {
   msg = "Go away Peter, I'm busy!"
} else {
   msg = "Nice to meet you!"
}
msg
```

```
[1] "Nice to meet you!"
```

If-expressions return the value of the last expression in the evaluated block:

```r
name = "Tom"
msg = if (name == "Nick") {
   "Hi Nick, nice to see you again!"
} else {
   "Nice to meet you!"
}
msg
```

```
[1] "Nice to meet you!"
```

Curly braces `{ }` are optional for single-line expressions:

```
name = "Nick"
msg = if (name == "Nick") "Hi Nick, nice to see you again!" else
   "Nice to meet you!"
msg
```

```
[1] "Hi Nick, nice to see you again!"
```

But you have to be careful if you don't use them:

```
# NO GOOD:
msg = if (name == "Nick")
   "Hi Nick, nice to see you again!"
else
   "Nice to meet you!"
```

```
Error: <text>:4:1: unexpected 'else'
3:    "Hi Nick, nice to see you again!"
4: else
   ^
```

The `else` block is optional:

```
msg = "Hi"
name = "Tom"
if (name == "Nick")
   msg = "Hi Nick, nice to see you again!"
msg
```

```
[1] "Hi"
```

When there's no `else` block, the value of the `else` block is `NULL`:

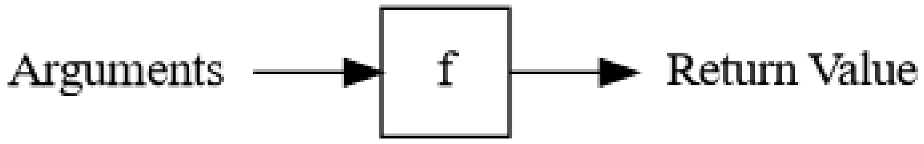
```
name = "Tom"
msg = if (name == "Nick")
   "Hi Nick, nice to see you again!"
msg
```

```
NULL
```

4.2 Functions

The main way to interact with R is by calling functions, which was first explained way back in Section 1.2.4. Since then, you've learned how to use many of R's built-in functions. This section explains how you can write your own functions.

To start, let's briefly review what functions are, and some of the jargon associated with them. It's useful to think of functions as factories: raw materials (inputs) go in, products (outputs) come out. We can also represent this visually:



Programmers use several specific terms to describe the parts and usage of functions:

- **Parameters** are placeholder variables for inputs.
    - **Arguments** are the actual values assigned to the parameters in a call.
- The **return value** is the output.
- The **body** is the code inside.
- **Calling** a function means using a function to compute something.

Almost every command in R is a function, even the arithmetic operators and the parentheses! You can view the body of a function by typing its name without trailing parentheses (in contrast to how you call functions). The body of a function is usually surrounded by curly braces `{}`, although they're optional if the body only contains one line of code. Indenting code inside of curly braces by 2-4 spaces also helps make it visually distinct from other code.

For example, let's look at the body of the `append` function, which appends a value to the end of a list or vector:

```r
append
```

```
function (x, values, after = length(x))
{
    lengx <- length(x)
    if (!after)
        c(values, x)
    else if (after >= lengx)
        c(x, values)
    else c(x[1L:after], values, x[(after + 1L):lengx])
}
```

```
<bytecode: 0x63d987cde588>
<environment: namespace:base>
```

Don't worry if you can't understand everything the `append` function's code does yet. It will make more sense later on, after you've written a few functions of your own.

Many of R's built-in functions are not entirely written in R code. You can spot these by calls to the special `.Primitive` or `.Internal` functions in their code.

For instance, the `sum` function is not written in R code:

```r
sum
```

```
function (..., na.rm = FALSE)  .Primitive("sum")
```

The `function` keyword creates a new function. Here's the syntax:

```
function(parameter1, parameter2, ...) {
  # Your code goes here

  # The result goes here
}
```

A function can have any number of parameters, and will automatically return the value of the last line of its body.

A function is a value, and like any other value, if you want to reuse it, you need to assign it to variable. Choosing descriptive variable names is a good habit. For functions, that means choosing a name that describes what the function does. It often makes sense to use verbs in function names.

Let's write a function that gets the largest values in a vector. The inputs or arguments to the function will be the vector in question and also the number of values to get. Let's call these `vec` and `n`, respectively. The result will be a vector of the `n` largest elements. Here's one way to write the function:

```r
get_largest = function(vec, n) {
  sorted = sort(vec, decreasing = TRUE)
  head(sorted, n)
}
```

The name of the function, `get_largest`, describes what the function does and includes a verb. If this function will be used frequently, a shorter name, such as `largest`, might be preferable (compare to the `head` function).

Any time you write a function, the first thing you should do afterwards is test that it actually works. Let's try the `get_largest` function on a few test cases:

```r
x = c(1, 10, 20, -3)
get_largest(x, 2)
```

```
[1] 20 10
```

```r
get_largest(x, 3)
```

```
[1] 20 10  1
```

```r
y = c(-1, -2, -3)
get_largest(y, 2)
```

```
[1] -1 -2
```

```r
z = c("d", "a", "t", "a", "l", "a", "b")
get_largest(z, 3)
```

```
[1] "t" "l" "d"
```

Notice that the parameters `vec` and `n` inside the function do not exist as variables outside of the function:

```r
vec
```

```
Error in eval(expr, envir, enclos): object 'vec' not found
```

In general, R keeps parameters and variables you define inside of a function separate from variables you define outside of a function. You can read more about the specific rules for how R searches for variables in DataLab's Intermediate R reader.

As a function for quickly summarizing data, `get_largest` would be more convenient if the parameter `n` for the number of values to return was optional (again, compare to the `head` function). You can make the parameter `n` optional by setting a **default argument**: an argument assigned to the parameter if no argument is assigned in the call to the function. You can use `=` to assign default arguments to parameters when you define a function with the `function` keyword. Here's a new definition of the function with the default `n = 5`:

```
get_largest = function(vec, n = 5) {
  sorted = sort(vec, decreasing = TRUE)
  head(sorted, n)
}
```

After making this change, it's a good idea to test the function again:

```
get_largest(x)
```

```
[1] 20 10  1 -3
```

```
get_largest(y)
```

```
[1] -1 -2 -3
```

```
get_largest(z)
```

```
[1] "t" "l" "d" "b" "a"
```

4.2.1 Returning Values

We've already seen that a function will automatically return the value of its last line.

The `return` keyword causes a function to return a result immediately, without running any subsequent code in its body. It only makes sense to use `return` from inside of an if-expression. If your function doesn't have any if-expressions, you don't need to use `return`.

For example, suppose you want the `get_largest` function to immediately return `NULL` if the argument for `vec` is a list. Here's the code, along with some test cases:

```r
get_largest = function(vec, n = 5) {
  if (is.list(vec))
    return(NULL)

  sorted = sort(vec, decreasing = TRUE)
  head(sorted, n)
}

get_largest(x)
```

```
[1] 20 10  1 -3
```

```r
get_largest(z)
```

```
[1] "t" "l" "d" "b" "a"
```

```r
get_largest(list(1, 2))
```

```
NULL
```

Alternatively, you could make the function raise an error by calling the `stop` function. Whether it makes more sense to return `NULL` or print an error depends on how you plan to use the `get_largest` function.

Notice that the last line of the `get_largest` function still doesn't use the `return` keyword. It's idiomatic to only use `return` when strictly necessary.

A function returns one R object, but sometimes computations have multiple results. In that case, return the results in a vector, list, or other data structure.

For example, let's make a function that computes the mean and median for a vector. We'll return the results in a named list, although we could also use a named vector:

```r
compute_mean_med = function(x) {
  m1 = mean(x)
  m2 = median(x)
  list(mean = m1, median = m2)
}
compute_mean_med(c(1, 2, 3, 1))
```

```
$mean
[1] 1.75

$median
[1] 1.5
```

The names make the result easier to understand for the caller of the function, although they certainly aren't required here.

4.2.2 Planning Your Functions

Before you write a function, it's useful to go through several steps:

1. Write down what you want to do, in detail. It can also help to draw a picture of what needs to happen.
2. Check whether there's already a built-in function. Search online and in the R documentation.
3. Write the code to handle a simple case first. For data science problems, use a small dataset at this step.

Let's apply this in one final example: a function that detects leap years. A year is a leap year if either of these conditions is true:

- It is divisible by 4 and not 100
- It is divisible by 400

That means the years 2004 and 2000 are leap years, but the year 2200 is not. Here's the code and a few test cases:

```r
# If year is divisible by 4 and not 100 -> leap
# If year is divisible by 400 -> leap
year = 2004
is_leap = function(year) {
  if (year %% 4 == 0 & year %% 100 != 0) {
    leap = TRUE
  } else if (year %% 400 == 0) {
    leap = TRUE
  } else {
    leap = FALSE
  }
  leap
}
is_leap(400)
```

```
[1] TRUE
```

```
is_leap(1997)
```

```
[1] FALSE
```

Functions are the building blocks for solving larger problems. Take a divide-and-conquer approach, breaking large problems into smaller steps. Use a short function for each step. This approach makes it easier to:

- Test that each step works correctly.
- Modify, reuse, or repurpose a step.

4.3 Exercises

These exercises are meant to challenge you, so they're quite difficult compared to the previous ones. Don't get disheartened, and if you're able to complete them, excellent work!

4.3.1 Exercise

Create a function `compute_day` which uses the Doomsday algorithm to compute the day of week for any given date in the 1900s. The function's parameters should be `year`, `month`, and `day`. The function's return value should be a day of week, as a string (for example, `"Saturday"`).

Hint: the modulo operator is `%%` in R.

5 Appendix

5.1 More About Comparisons

5.1.1 Equality

The == operator is the primary way to test whether two values are equal, as explained in Section 1.2.3. Nonetheless, equality can be defined in many different ways, especially when dealing with computers. As a result, R also provides several different functions to test for different kinds of equality. This describes tests of equality in more detail, and also describes some other important details of comparisons.

5.1.1.1 The == Operator

The == operator tests whether its two arguments have the exact same representation as a **binary number** in your computer's memory. Before testing the arguments, the operator applies R's rules for vectorization (Section 2.1.3), recycling (Section 2.1.4), and implicit coercion (Section 2.2.2). Until you've fully internalized these three rules, some results from the equality operator may seem surprising. For example:

```r
# Recycling:
c(1, 2) == c(1, 2, 1, 2)
```

```
[1] TRUE TRUE TRUE TRUE
```

```r
# Implicit coercion:
TRUE == 1
```

```
[1] TRUE
```

```r
TRUE == "TRUE"
```

```
[1] TRUE
```

```
1 == "TRUE"
```

```
[1] FALSE
```

The length of the result from the equality operator is usually the same as its longest argument (with some exceptions).

5.1.1.2 The `all.equal` Function

The `all.equal` function tests whether its two arguments are equal up to some acceptable difference called a **tolerance**. Computer representations for decimal numbers are inherently imprecise, so it's necessary to allow for very small differences between computed numbers. For example:

```
x = 0.5 - 0.3
y = 0.3 - 0.1

# FALSE on most machines:
x == y
```

```
[1] FALSE
```

```
# TRUE:
all.equal(x, y)
```

```
[1] TRUE
```

The `all.equal` function does not apply R's rules for vectorization, recycling, or implicit coercion. The function returns `TRUE` when the arguments are equal, and returns a string summarizing the differences when they are not. For instance:

```
all.equal(1, c(1, 2, 1))
```

```
[1] "Numeric: lengths (1, 3) differ"
```

The `all.equal` function is often used together with the `isTRUE` function, which tests whether the result is `TRUE`:

```r
all.equal(3, 4)
```

```
[1] "Mean relative difference: 0.3333333"
```

```r
isTRUE(all.equal(3, 4))
```

```
[1] FALSE
```

You should generally use the `all.equal` function when you want to compare decimal numbers.

5.1.1.3 The `identical` Function

The `identical` function checks whether its arguments are completely identical, including their metadata (names, dimensions, and so on). For instance:

```r
x = list(a = 1)
y = list(a = 1)
z = list(1)

identical(x, y)
```

```
[1] TRUE
```

```r
identical(x, z)
```

```
[1] FALSE
```

The `identical` function does not apply R’s rules for vectorization, recycling, or implicit coercion. The result is always a single logical value.

You’ll generally use the `identical` function to compare non-vector objects such as lists or data frames. The function also works for vectors, but most of the time the equality operator `==` is sufficient.

5.1.2 The %in% Operator

Another common comparison is to check whether elements of one vector are contained in another vector at any position. For instance, suppose you want to check whether 1 or 2 appear anywhere in a longer vector x. Here's how to do it:

```r
x = c(3, 4, 2, 7, 3, 7)
c(1, 2) %in% x
```

```
[1] FALSE  TRUE
```

R returns FALSE for the 1 because there's no 1 in x, and returns TRUE for the 2 because there is a 2 in x.

Notice that this is different from comparing with the equality operator ==. If you use use the equality operator, the shorter vector is recycled until its length matches the longer one, and then compared element-by-element. For the example, this means only the elements at odd-numbered positions are compared to 1, and only the elements at even-numbered positions are compared to 2:

```r
c(1, 2) == x
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

5.1.3 Summarizing Comparisons

The comparison operators are vectorized, so they compare their arguments element-by-element:

```r
c(1, 2, 3) < c(1, 3, -3)
```

```
[1] FALSE  TRUE FALSE
```

```r
c("he", "saw", "her") == c("she", "saw", "him")
```

```
[1] FALSE  TRUE FALSE
```

What if you want to summarize whether all the elements in a vector are equal (or unequal)? You can use the all function on any logical vector to get a summary. The all function takes a vector of logical values and returns TRUE if all of them are TRUE, and returns FALSE otherwise:

```r
all(c(1, 2, 3) < c(1, 3, -3))
```

```
[1] FALSE
```

The related `any` function returns `TRUE` if any one element is `TRUE`, and returns `FALSE` otherwise:

```r
any(c("hi", "hello") == c("hi", "bye"))
```

```
[1] TRUE
```

5.1.4 Other Pitfalls

New programmers sometimes incorrectly think they need to append `== TRUE` to their comparisons. This is redundant, makes your code harder to understand, and wastes computational time. Comparisons already return logical values. If the result of the comparison is `TRUE`, then `TRUE == TRUE` is again just `TRUE`. If the result is `FALSE`, then `FALSE == TRUE` is again just `FALSE`. Likewise, if you want to invert a condition, choose an appropriate operator rather than appending `== FALSE`.

5.2 The `drop` Parameter

If you use two-dimensional indexing with `[` to select exactly one column, you get a vector:

```r
result = banknotes[1:3, 2]
class(result)
```

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

The container is dropped, even though the indexing operator `[` usually keeps containers. This also occurs for matrices. You can control this behavior with the `drop` parameter:

```r
result = banknotes[1:3, 2, drop = FALSE]
class(result)
```

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

The default is `drop = TRUE`.

Where to Learn More

This reader provides an introduction to the basics of R, but there's lots more to learn!

DataLab's Intermediate R workshops are designed specifically for learners who have taken this workshop and want to learn more about R.

DataLab also teaches many other workshops about data science. If you're not sure where to start or how these workshops all fit together, take a look at our Reproducibility Principles and Practices reader.

If you want to learn more about how to design clear and effective data visualizations, take a look at our Principles of Data Visualization reader.

Many R and data science learning resources are available for free online or through the library. Here are a few books created by others that we've found useful:

- R for Data Science by Wickham & Grolemund. An introduction to using R for data science, but with a very heavy focus on Tidyverse packages.
- The Art of R Programming by Matloff. A general reference on R programming.
- Advanced R by Wickham. A description of how R works at a deeper level, with many examples of R features that are important for package/software development.
- The R Inferno by Burns. A discussion of the most difficult and confusing parts of R.

Finally, here are some websites popular in the R community:

- R Graph Gallery. Examples of graphs you can make in R, with code.
- RStudio Cheat Sheets. Cheat sheets for a variety of R tools and packages.
- R Weekly. Weekly updates about what's happening in the R community.
- R-bloggers. An aggregator for blog posts about R.

Acknowledgements

Sincere thanks to the rest of the DataLab team, for support and many thoughtful discussions about data science!

Nick's Acknowledgements

Sincere thanks to:

Duncan Temple Lang, whose guidance and encouragement has had outsize influence on the way I think about and teach data science.

Deb Nolan, for being a role model as a data scientist, educator, and academic. Special thanks for the opportunity to collaborate on STAT 33 (on which this reader and workshop is loosely based) at Cal.

Last but not least, every former student that asked questions or offered feedback when I taught.

Assessment

If you are taking this workshop to complete a GradPathways Pathway, you can download the assessment instructions here.