R Basics

Nick Ulle Wesley Brooks

2025-12-12

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 R and RStudio. For sessions presented online, learners will also need the latest version of Zoom.

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

1.1 Why R?

Why should you use a programming language?

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

- Automate repetitive tasks within an analysis
- Recycle code from one analysis into another
- Package useful code for distribution to your colleagues or the general public

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

> **i** Note
>
> The term "R" can mean the R language (the code) or the R interpreter (the software which runs the code). Most of the time, the meaning is clear from the context, but we'll be explicit in cases where the distinction is important.

Compared to other programming languages, some of R's particular strengths are its:

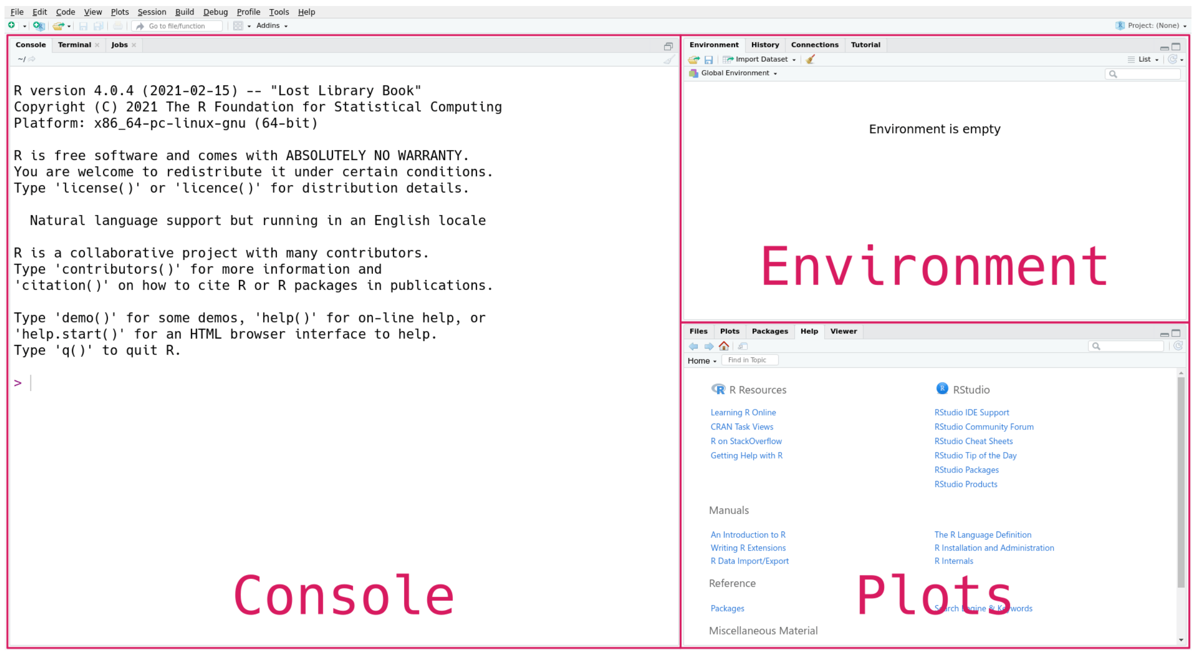
- Interactive interpreter and debugger
- Focus on statistical computing
  - Many statistical estimators, tests, and models are built-in
  - Statisticians tend to implement new methods in R first
- Over 23,000 community-developed **packages**: reusable bundles of code, often accompanied by documentation, examples, or data sets
- Functions and packages for creating high-quality data visualizations
- Flexible support for many different programming abstractions and paradigms

The main way you'll interact with R is by writing R code or **expressions**. We'll explain more soon, but first we need to install R and associated software.

> **!** Important
>
> To follow along with this and subsequent chapters, you'll need a recent version of R. Click here to download the R installer. When the download is complete, run the installer to install (or update) R on your computer. We recommend keeping the installer's default settings.
> In addition to R, you'll need a recent version of 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.
> Click here to download the free RStudio Desktop installer. When the download is complete, run the installer to install RStudio Deskop on your computer.

1.2 The RStudio 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.

>
```

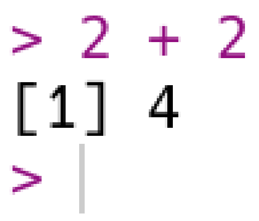
Environment is empty

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.

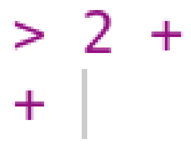
Let's start by using R to do some arithmetic. In the console, you'll see that the cursor is on a line that begins with `>`, called the **prompt**. You can make R compute the sum $2 + 2$ by typing the code `2 + 2` after the prompt and then pressing the `Enter` key. Your code and the result from R should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

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!

1.3 R Basics

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:

```
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.

Tip

Use spaces!
As with writing text, putting spaces in your code makes it easier for you and others to read, so it's good to make it a habit. Put a single space on each side of most operators, after commas, and after keywords.

1.3.1 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:

```
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:

```
pi * 3^2
```

```
[1] 28.27433
```

You can create a variable with the assignment operator `=` by writing a name on the left-hand side and a value or expression on the right-hand side. For example, to save the area of the circle in a variable called `area`, you can write:

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

Note

You can use the arrow operator `<-` instead of the assignment operator:

```r
area <- pi * 3^2
```

In most cases, the 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 `=` because it's the assignment operator in most programming languages and it's easier to type.

In R, variable names can contain any combination of letters and dots (`.`). Names can also include numbers and underscores (`_`), but can't start with a them. Spaces and other symbols are not allowed in variable names. So `geese`, `top50.dogs`, and `nine_lives` are valid variable names, but `goose teeth`, `_fishes`, and `9lives` are not.

The main reason to use variables is to temporarily save results from expressions so that you can use them in other expressions. For instance, now you can use the `area` variable anywhere you want the area of the circle.

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 clearer and 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. Instead, you can create a variable `r`, then rewrite the expression as `pi * r^2`. This makes the expression easier to understand, because the reader doesn't have to intuit that `3` is the radius in the formula. 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.

Tip

Try to choose descriptive variable names, so that you and your collaborators can understand the meaning and purpose of each variable when reading the code.

1.3.2 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.

1.3.3 Comparisons

Besides arithmetic, you can also use R to compare values. Programming tasks often involve comparing values. Use **comparison operators** to do so:

- `<` 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"

Notice that the "equal to" operator is two equal signs. This is to 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.

⚠ Warning

Some of R's equality operators return TRUE even when comparing 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.

1.3.4 Calling Functions

R can do a lot more than just arithmetic. 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, the function to round a number to a specified decimal place is named `round`. So you can round the number `8.153` to the nearest integer with this code:

```r
round(8.153)
```

```
[1] 8
```

Many functions accept more than one argument. For instance, the `round` function accepts at least two arguments: the number to round and the number of decimal places to keep. When you call a function with multiple arguments, separate the arguments with commas. So to round `8.153` to 1 decimal place:

```r
round(8.153, 1)
```

```
[1] 8.2
```

When you call a function, R assigns the arguments to the function's **parameter**. Parameters are special variables that represent the inputs to a function and only exist while that function runs. For example, the `round` function has parameters `x` and `digits`. The next section, Section 1.4, explains how to look up the parameters of a function.

Some parameters have **default arguments**. A parameter is automatically assigned its default argument whenever the parameter's argument is not specified explicitly. As a result, assigning arguments to these parameters is optional. For instance, the `digits` parameter of `round` has a default argument (round to the nearest integer), so it's okay to call `round` without setting `digits`, as in `round(8.153)`. In contrast, the `x` parameter doesn't have a default argument. Section 1.4 explains how to look up the default arguments for a function.

By default, R assigns arguments to parameters based on their position. The first argument is assigned to the function's first parameter, the second to the second, and so on. So in the code above, `8.153` is assigned to `x` and `1` is assigned to `digits`.

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
round(8.153, digits = 1)
```

```
[1] 8.2
```

```r
round(x = 8.153, digits = 1)
```

```
[1] 8.2
```

```r
round(digits = 1, x = 8.153)
```

```
[1] 8.2
```

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 = 4.755
round(digits = 2)
```

```
Error in round(digits = 2): 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
round(x, digits = 2)
```

```
[1] 4.76
```

Or, written more explicitly:

```r
round(x = x, digits = 2)
```

```
[1] 4.76
```

The point is that 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`.

1.3.5 Comments

In R and most other programming languages, you can mark parts of your code as **comments**: expressions to ignore rather than run. Use comments to plan, explain, and document your code. You can also temporarily "comment out" code to prevent it from running, which is often helpful for testing and debugging.

R comments begin with number sign `#` and extend to the end of the line:

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

R will ignore comments when you run your code.

1.4 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 `round` function is:

```
?round
```

For functions, help pages usually include a brief description, a list of parameters, a description of the return value, and some examples. The help page for `round` shows that there are two parameters `x` and `digits`. It also says that `digits = 0`, meaning the default argument for `digits` is `0`.

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.

1.4.1 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.

1.5 File Systems

This section is a review of how files on a computer work. You'll need to understand this in order to read a data set from a file, and it's also important for finding your saved notebooks and modules later.

Your computer's **file system** consists 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:

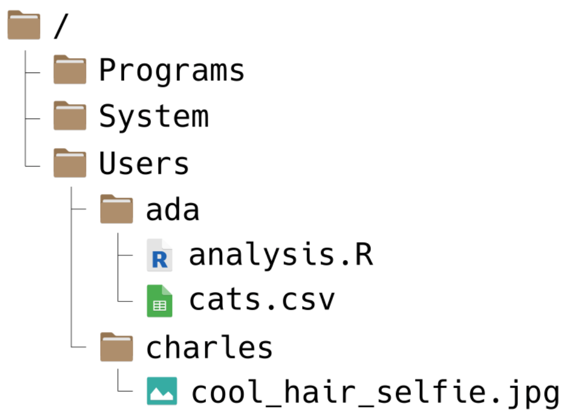


Figure 1.2: A typical file system. Don't worry if your file system looks a bit different from the picture.

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.

⚠ Warning

On Windows, the components of a path are conventionally separated with backslashes `\` instead of forward slashes `/`.
Regardless of the operating system, R uses forward slashes `/` to separate components of paths. In other words, you should use paths separated by with forward slashes in your R code, even on Windows. This is especially convenient when you want to share code with other people, because they might use a different operating system than you.

1.5.1 Absolute & Relative Paths

A path that starts from the root directory, like all of the ones we'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`, 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.

Tip

When use you paths in 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 `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.

1.5.2 The Working Directory

Section 1.5.1 explained that relative paths have a starting point that depends on the context where the path is used. 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()
```

Warning

In your R scripts and notebooks, avoid calls to `setwd`. They make your code more difficult to understand and to run on other computers. Use appropriate relative paths instead. In the R console, it's okay to occasionally use `setwd`. You might need to change the working directory before you run some code. R's default working directory is your home directory. In some cases, such as when you open a project, RStudio will automatically change the working directory. However, it doesn't always change the working directory, so `setwd` is sometimes still necessary.

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] "_build"          "_freeze"         "_quarto.yml"     "assessment"
 [5] "chapters"        "CONTRIBUTING.md" "data"            "images"
 [9] "index.html"      "index.qmd"       "LICENSE"         "notes"
[13] "pixi.lock"       "pixi.toml"       "R"               "README.md"
[17] "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.

1.6 Saving & Loading Code

Tip

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 store your R code in that directory. As you work, periodically save your code.

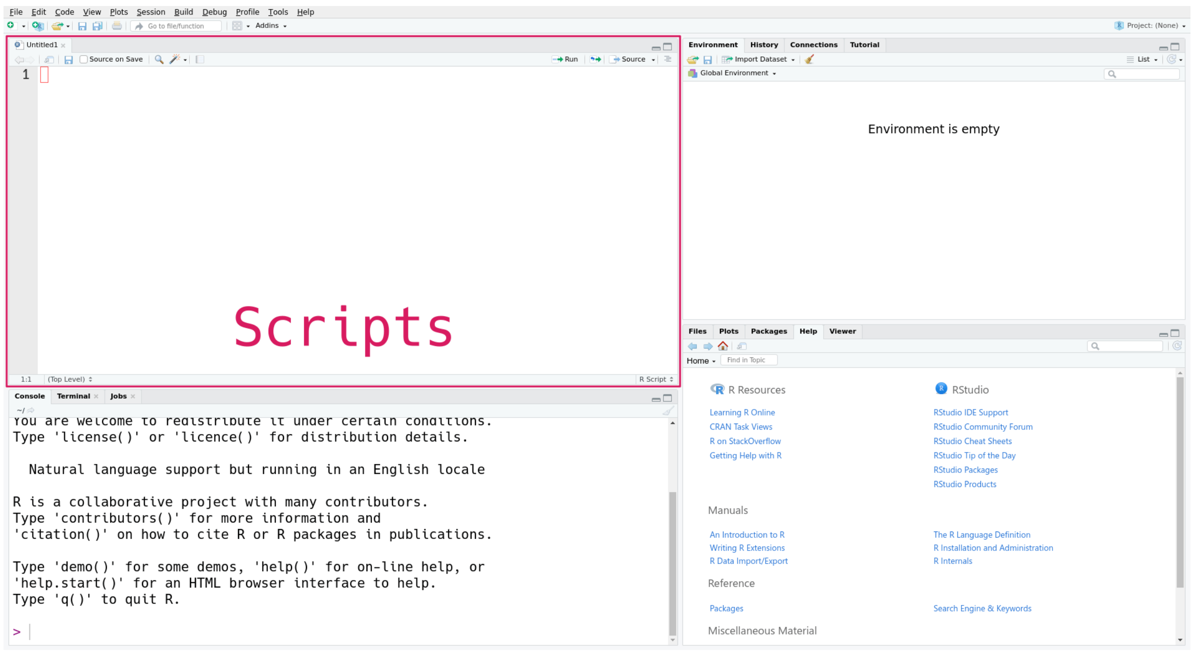
Most of the time, you won't just write code directly into the R console. Reproducibility and reusability are important benefits of R over point-and-click software, and in order to realize these, you have to save your code to your computer's hard drive.

The most common way to save R code is as an R **script** with the extension `.R` (see Section 1.7 for more about extensions). Editing a script is similar to editing any other text file. You can write, delete, copy, cut, and paste code.

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.

Every line in an R script must be valid R code. Anything else you want to write in the script (notes, documentation, etc.) must be placed in a comment.

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.

Tip

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.

1.6.1 Running Scripts

You can **source** (that is, run) an 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 enter `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.

1.6.2 Notebooks: Quarto & R Markdown

In the context of data science, a **notebook** is an interactive file that can store a mix of code, formatted text, and images. With a notebook, you can write, run code, and view results all in one place. Viewing and editing a notebook requires a web browser or IDE. Some notebooks can also be converted to static documents, such as PDFs. Comments are a good way keep notes as you develop and run your code, but notebooks provide much more flexibility of expression. Notebooks are a kind of literate programming.

Notebooks excel when you want to do highly interactive work and/or want to communicate results. Use notebooks to prototype code, analyze data, refine plots, generate documents and presentations, and practice programming. Scripts excel when you want to reuse code (and perhaps share it as a package) or want to run code that doesn't require much user interaction (such as time-consuming computations you'll run on a server or high-performance computing cluster). The remainder of this reader assumes you're using an R script rather than the R console or a notebook, unless otherwise noted.

Quarto is a popular notebook format and system for R. It also supports Python, Julia, and other programming languages. Quarto files have the extension `.qmd`. Quarto is based on an older notebook format, R Markdown, which only supports R and can't be converted to as many kinds of documents. R Markdown is still widely used. R Markdown files have the extension `.Rmd`.

! Important

In order to use Quarto, you must first download and install it. It is not included with R or RStudio.

After installing Quarto, you can create a new Quarto file in RStudio with this menu option:

```
File -> New -> Quarto Document...
```

RStudio will prompt you to provide some details about the purpose of the file.

Notebooks are subdivided into **chunks** (or cells). You can create as many chunks as you like, but each chunk can contain only one kind of content. You can run a code chunk by clicking on the chunk and pressing `Ctrl-Enter`. The notebook will display the result.

Markdown is a simple language you can use to add formatting to (non-code) text in a notebook. For example, surrounding a word with asterisks, as in `Let *sleeping* dogs lie`, makes the surrounded word italic. You can find a short, interactive tutorial about Markdown here.

> See Also
>
> To learn more about Quarto, see the official documentation.

1.7 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

Name	Extension	Function or Package	Tabular?	Text?
R Data	`.rda`	`load`	Sometimes	No
Plaintext	`.txt`	`readLines`	Sometimes	Yes
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.

1.8 Dataset: CA Least Terns

The California least tern is a endangered subspecies of seabird that nests along the coast of California and Mexico. The California Department of Fish and Wildlife (CDFW) monitors least tern nesting sites across the state to estimate breeding pairs, fledglings, and predator activity in each annual breeding season.

Figure 1.4: A California least tern. Original photo by Mark Pavelka, U.S. Fish & Wildlife Service (CC BY 2.0).

The CDFW publishes most of the data it collects to the California Open Data portal. The examples in this and subsequent chapters use a cleaned 2000-2023 version of the California least tern data.

! Important

Click here to download the 2000-2023 California least tern data set.
If you haven't already, we recommend you create a directory for this workshop. In your workshop directory, create a `data/` subdirectory. Download and save the California least tern data set in the `data/` subdirectory.

Documentation for 2000-2023 California Least Tern Data Set

Each row in the data set contains measurements from one year-site combination.

Column	Description
`year`	Year of the breeding season
`site_name`	Site name
`site_name_2013_2018`	Site name from 2013-2018
`site_name_1988_2001`	Site name from 1988-2001
`site_abbr`	Abbreviated site name
`region_3`	Region of state: S.F. Bay, Central, or Southern (includes Ventura)
`region_4`	Region of state: S.F. Bay, Central, Ventura, or Southern
`event`	Climate events
`bp_min`	Reported minimum breeding pairs
`bp_max`	Reported maximum breeding pairs
`fl_min`	Reported minimum fledges
`fl_max`	Reported maximum fledges
`total_nests`	Total reported nests (maximum if a range was reported)
`nonpred_eggs`	Total non-predator-related mortalities of eggs
`nonpred_chicks`	Total non-predator-related mortalities of chicks
`nonpred_fl`	Total non-predator-related mortalities of fledges
`nonpred_ad`	Total non-predator-related mortalities of adults
`pred_control`	Site predator control (yes/no)
`pred_eggs`	Total predator-related mortalities of eggs
`pred_chicks`	Total predator-related mortalities of chicks
`pred_fl`	Total predator-related mortalities of fledges
`pred_ad`	Total predator-related mortalities of adults
`pred_pefa`	Predation by peregrine falcons (yes/no)
`pred_coy_fox`	Predation by coyotes or foxes (yes/no)
`pred_meso`	Predation by other mesocarnivores: dogs, cats, skunks, opossums, raccoons, weasels, etc. (yes/no)
`pred_owlspp`	Predation by owls (yes/no)
`pred_corvid`	Predation by corvids: ravens or crows (yes/no)
`pred_other_raptor`	Predation by raptors other than peregrine falcons and owls (yes/no)
`pred_other_avian`	Predation by birds other than raptors and corvids (yes/no)
`pred_misc`	Predation by other animals (yes/no)
`total_pefa`	Total mortalities due to peregrine falcons
`total_coy_fox`	Total mortalities due to coyotes and foxes
`total_meso`	Total mortalities due to other mesocarnivores
`total_owlspp`	Total mortalities due to owls
`total_corvid`	Total mortalities due to ravens and crows
`total_other_raptor`	Total mortalities due to other raptors

`total_other_avian`	Total mortalities due to other birds
`total_misc`	Total mortalities due to other animals
`first_observed`	Date CA least terns first observed at site
`last_observed`	Date CA least terns last observed at site
`first_nest`	Date first egg observed at site
`first_chick`	Date first chick observed at site
`first_fledge`	Date first fledge observed at site

The messy source data set (with more years and more columns) is available here.

Let's use R to read the California least tern data set. The data set is in a file called is `2000-2023_ca_least_tern.csv`, which suggests it's a CSV file. The function to read a CSV file is `read.csv`. The function's first and only required argument is the path to the CSV file.

In the following code, the path to the California least tern data set is `data/2000-2023_ca_least_tern.csv`, but it might be different for you, depending on R's working directory and where you saved the file. We'll save the result from the `read.csv` function in a variable called `terns`. We can use this variable to access the data in subsequent code.

```r
terns = read.csv("data/2000-2023_ca_least_tern.csv")
```

Note

The variable name `terns` 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 from Section 1.4. The most common mistake when reading a file is incorrectly specifying the path, so first check that you got the path right.

If the code ran without errors, it's a good idea to check that the data set looks like what the documentation describes. When working with a new data set, it usually isn't a good idea to print the whole thing (at least until you know how big it is). Large data sets can take a long time to print, and the output can be difficult to read.

Instead, use the `head` function to print only the beginning, or head, of the data set:

```r
head(terns)
```

year	site_name	site_name_2013_2018	site_name_1988_2001	site_abbr	region_3	region_4	event	bp_min	bp_max	fl_min	fl_max	total_nests	nonpred_eggs	nonpred_chicks	nonpred_fl	nonpred_ad	pred_control	pred_eggs	pred_chicks	pred_fl	pred_ad	pred_pefa	pred_coy_fox	pred_meso	pred_owlspp	pred_corvid	pred_other_raptor	pred_other_avian	pred_misc	total_pefa	total_coy_fox	total_meso	total_owlspp	total_corvid	total_other_raptor	total_other_avian	total_misc	first_observed	last_observed	first_nest	first_chick	first_fledge
2000	PITTSBURG POWER PLANT	Pittsburg Power Plant	NA_NO POLYGON	PITT_POWER	S.F._BAY	S.F._BAY	LA_NINA	15	15	16	18	15	3	0	0	0		4	2	0	0	N	N	N	N	Y	Y	N	N	0	0	0	0	4	2	0	0	2000-05-11	2000-08-05	2000-05-26	2000-06-18	2000-07-08
2000	ALBANY CENTRAL AVE	AL Central Avenue	NA_NO POLYGON	AL_CENTAVE	S.F._BAY	S.F._BAY	LA_NINA	6	12	1	1	20	NA	NA	NA	NA		NA	NA	NA	NA									NA	NA	NA	NA	NA	NA	NA	NA					
2000	ALAMEDA POINT	Alameda Point	NA_NO POLYGON	AL_PT	S.F._BAY	S.F._BAY	LA_NINA	282	301	200	230	312	124	81	2	1		17	0	0	0	N	N	N	N	N	Y	Y	N	0	0	0	0	0	6	11	0	2000-05-01	2000-08-19	2000-05-16	2000-06-07	2000-06-30
2000	KETTLEMAN CITY	Kettleman	NA_2013_2018 POLYGON	KET_CITY	KINGS	KINGS	LA_NINA	[illegible]	[illegible]	[illegible]	2	3	NA	3	1	6		NA	NA	NA	NA									NA	NA	NA	NA	NA	NA	NA	NA	2000-06-10	2000-09-24	2000-06-17	2000-07-22	2000-08-06
2000	OCEANO DUNES STATE VEHICULAR RECREATION AREA	Oceano Dunes State Vehicular Recreation Area	NA_2013_2018 POLYGON	OCEANO_DUNES	CENTRAL	[illegible]	LA_NINA	[illegible]	[illegible]	[illegible]	[illegible]	5	2	0	0	0		0	4	0	0	N	N	N	N	N	N	Y	N	0	0	0	0	0	0	4	0	2000-05-04	2000-08-30	2000-05-28	2000-06-20	2000-07-13
2000	RANCHO GUADALUPE DUNES PRESERVE	Rancho Guadalupe Dunes Preserve	NA_2013_2018 POLYGON	RANCH_GUAD	CENTRAL	[illegible]	LA_NINA	[illegible]	[illegible]	[illegible]	17	9	0	1	0	0		NA	NA	NA	NA									NA	NA	NA	NA	NA	NA	NA	NA	2000-05-07	2000-08-13	2000-05-31	2000-06-22	2000-07-20

If you run this code and see a similar table, then congratulations, you've read your first data set into R!

The California least terns 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.

Typically, each row in a data frame corresponds to a single subject and is called an **observation**. Each column corresponds to a measurement of the subject and is called a **feature** or **covariate**.

> **Note**
>
> Sometimes people also refer to columns as "variables," but we'll try to avoid this, because in programming contexts a variable is a name for a value (which might not be a column).

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(terns)
```

```
[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.

1.9 Inspecting a Data Frame

Similar to how the `head` function shows the first six rows of a data frame, the `tail` function shows the last six:

```r
tail(terns)
```

```
    year                                site_name         site_name_2013_2018
786 2023                         NAS NORTH ISLAND         Naval Base Coronado
787 2023             NAVAL AMPHIBIOUS BASE CORONADO         Naval Base Coronado
788 2023          DSTREET FILL SWEETWATER MARSH NWR                D Street Fill
789 2023              CHULA VISTA WILDLIFE RESERVE Chula Vista Wildlife Refuge
790 2023 SOUTH SAN DIEGO BAY UNIT SDNWR SALTWORKS                    Saltworks
791 2023                        TIJUANA ESTUARY NERR             Tijuana Estuary
     site_name_1988_2001 site_abbr region_3 region_4   event bp_min bp_max
786 NA_2013_2018 POLYGON     NASNI SOUTHERN SOUTHERN LA_NINA      0      0
787 NA_2013_2018 POLYGON       NAB SOUTHERN SOUTHERN LA_NINA    596    644
```

```
788 NA_2013_2018 POLYGON      D_ST SOUTHERN SOUTHERN LA_NINA     29     38
789 NA_2013_2018 POLYGON        CV SOUTHERN SOUTHERN LA_NINA     47     54
790 NA_2013_2018 POLYGON      SALT SOUTHERN SOUTHERN LA_NINA     38     41
791 NA_2013_2018 POLYGON    TJ_RIV SOUTHERN SOUTHERN LA_NINA    144    165
    fl_min fl_max total_nests nonpred_eggs nonpred_chicks nonpred_fl nonpred_ad
786      0      0           0            0              0          0          0
787     90    128         717          329            185          6          6
788      4      4          44           25              2          0          0
789      5      6          59           32              1          0          0
790      7      7          48           11              2          0          0
791     35     35         171           65             44          1          1
    pred_control pred_eggs pred_chicks pred_fl pred_ad pred_pefa pred_coy_fox
786            Y        NA          NA      NA      NA         N            N
787            Y        NA          NA      NA      NA         N            N
788            Y        NA          NA      NA      NA         Y            N
789            Y        NA          NA      NA      NA         Y            N
790            Y        NA          NA      NA      NA         Y            Y
791            Y        NA          NA      NA      NA         N            N
    pred_meso pred_owlspp pred_corvid pred_other_raptor pred_other_avian
786         N           N           N                 N                N
787         N           N           Y                 N                Y
788         N           N           N                 Y                Y
789         N           N           N                 N                N
790         N           N           N                 Y                N
791         N           N           N                 N                Y
    pred_misc total_pefa total_coy_fox total_meso total_owlspp total_corvid
786         N         NA            NA         NA           NA           NA
787         Y         NA            NA         NA           NA           NA
788         Y         NA            NA         NA           NA           NA
789         Y         NA            NA         NA           NA           NA
790         Y         NA            NA         NA           NA           NA
791         Y         NA            NA         NA           NA           NA
    total_other_raptor total_other_avian total_misc first_observed
786                 NA                NA         NA
787                 NA                NA         NA     2023-04-22
788                 NA                NA         NA     2023-04-20
789                 NA                NA         NA     2023-04-20
790                 NA                NA         NA     2023-04-24
791                 NA                NA         NA     2023-04-26
    last_observed first_nest first_chick first_fledge
786
787    2023-09-09 2023-05-07  2023-05-31
788    2023-08-24 2023-05-12  2023-06-05
```

```
789     2023-09-22 2023-05-14  2023-06-05
790     2023-09-22 2023-05-19  2023-06-09
791     2023-08-28 2023-05-12  2023-06-10
```

If there are lots of columns or the columns are wide, as is the case here, R wraps the output across lines.

Tip

Both `head` and `tail` accept an optional second argument that specifies the number of rows to print:

```r
head(terns, 1)
```

```
  year             site_name   site_name_2013_2018  site_name_1988_2001
1 2000 PITTSBURG POWER PLANT Pittsburg Power Plant NA_2013_2018 POLYGON
   site_abbr region_3 region_4   event bp_min bp_max fl_min fl_max total_nests
1 PITT_POWER S.F._BAY S.F._BAY LA_NINA     15     15     16     18          15
  nonpred_eggs nonpred_chicks nonpred_fl nonpred_ad pred_control pred_eggs
1            3              0          0          0                      4
  pred_chicks pred_fl pred_ad pred_pefa pred_coy_fox pred_meso pred_owlspp
1           2       0       0         N            N         N           N
  pred_corvid pred_other_raptor pred_other_avian pred_misc total_pefa
1           Y                 Y                N         N          0
  total_coy_fox total_meso total_owlspp total_corvid total_other_raptor
1             0          0            0            4                  2
  total_other_avian total_misc first_observed last_observed first_nest
1                 0          0     2000-05-11    2000-08-05 2000-05-26
  first_chick first_fledge
1  2000-06-18   2000-07-08
```

One way to get a quick idea of what your data looks like without having to skim through all the columns and rows is by inspecting its **dimensions**. This is the number of rows and columns in a data frame, and you can access this information with the `dim` function:

```r
dim(terns)
```

```
[1] 791  43
```

So this data set has 791 rows and 43 columns. As an alternative to the `dim` function, you can use the `nrow` and `ncol` functions to get just the number of rows and number of columns, respectively.

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(terns)
```

```
 [1] "year"              "site_name"         "site_name_2013_2018"
 [4] "site_name_1988_2001" "site_abbr"       "region_3"
 [7] "region_4"          "event"             "bp_min"
[10] "bp_max"            "fl_min"            "fl_max"
[13] "total_nests"       "nonpred_eggs"      "nonpred_chicks"
[16] "nonpred_fl"        "nonpred_ad"        "pred_control"
[19] "pred_eggs"         "pred_chicks"       "pred_fl"
[22] "pred_ad"           "pred_pefa"         "pred_coy_fox"
[25] "pred_meso"         "pred_owlspp"       "pred_corvid"
[28] "pred_other_raptor" "pred_other_avian"  "pred_misc"
[31] "total_pefa"        "total_coy_fox"     "total_meso"
[34] "total_owlspp"      "total_corvid"      "total_other_raptor"
[37] "total_other_avian" "total_misc"        "first_observed"
[40] "last_observed"     "first_nest"        "first_chick"
[43] "first_fledge"
```

```r
colnames(terns)
```

```
 [1] "year"              "site_name"         "site_name_2013_2018"
 [4] "site_name_1988_2001" "site_abbr"       "region_3"
 [7] "region_4"          "event"             "bp_min"
[10] "bp_max"            "fl_min"            "fl_max"
[13] "total_nests"       "nonpred_eggs"      "nonpred_chicks"
[16] "nonpred_fl"        "nonpred_ad"        "pred_control"
[19] "pred_eggs"         "pred_chicks"       "pred_fl"
[22] "pred_ad"           "pred_pefa"         "pred_coy_fox"
[25] "pred_meso"         "pred_owlspp"       "pred_corvid"
[28] "pred_other_raptor" "pred_other_avian"  "pred_misc"
[31] "total_pefa"        "total_coy_fox"     "total_meso"
[34] "total_owlspp"      "total_corvid"      "total_other_raptor"
[37] "total_other_avian" "total_misc"        "first_observed"
[40] "last_observed"     "first_nest"        "first_chick"
[43] "first_fledge"
```

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

1.9.1 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 least terns data set:

```r
str(terns)
```

```
'data.frame':	791 obs. of  43 variables:
 $ year               : int  2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 ...
 $ site_name          : chr  "PITTSBURG POWER PLANT" "ALBANY CENTRAL AVE" "ALAMEDA POINT" "KE
 $ site_name_2013_2018: chr  "Pittsburg Power Plant" "NA_NO POLYGON" "Alameda Point" "Kettlem
 $ site_name_1988_2001: chr  "NA_2013_2018 POLYGON" "Albany Central Avenue" "NA_2013_2018 POL
 $ site_abbr          : chr  "PITT_POWER" "AL_CENTAVE" "ALAM_PT" "KET_CTY" ...
 $ region_3           : chr  "S.F._BAY" "S.F._BAY" "S.F._BAY" "KINGS" ...
 $ region_4           : chr  "S.F._BAY" "S.F._BAY" "S.F._BAY" "KINGS" ...
 $ event              : chr  "LA_NINA" "LA_NINA" "LA_NINA" "LA_NINA" ...
 $ bp_min             : num  15 6 282 2 4 9 30 21 73 166 ...
 $ bp_max             : num  15 12 301 3 5 9 32 21 73 167 ...
 $ fl_min             : int  16 1 200 1 4 17 11 9 60 64 ...
 $ fl_max             : int  18 1 230 2 4 17 11 9 65 64 ...
 $ total_nests        : int  15 20 312 3 5 9 32 22 73 252 ...
 $ nonpred_eggs       : int  3 NA 124 NA 2 0 NA 4 2 NA ...
 $ nonpred_chicks     : int  0 NA 81 3 0 1 27 3 0 NA ...
 $ nonpred_fl         : int  0 NA 2 1 0 0 0 NA 0 NA ...
 $ nonpred_ad         : int  0 NA 1 6 0 0 0 NA 0 NA ...
 $ pred_control       : chr  "" "" "" "" ...
 $ pred_eggs          : int  4 NA 17 NA 0 NA 0 NA NA NA ...
 $ pred_chicks        : int  2 NA 0 NA 4 NA 3 NA NA NA ...
 $ pred_fl            : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ pred_ad            : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ pred_pefa          : chr  "N" "" "N" "" ...
 $ pred_coy_fox       : chr  "N" "" "N" "" ...
 $ pred_meso          : chr  "N" "" "N" "" ...
 $ pred_owlspp        : chr  "N" "" "N" "" ...
 $ pred_corvid        : chr  "Y" "" "N" "" ...
 $ pred_other_raptor  : chr  "Y" "" "Y" "" ...
 $ pred_other_avian   : chr  "N" "" "Y" "" ...
 $ pred_misc          : chr  "N" "" "N" "" ...
 $ total_pefa         : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ total_coy_fox      : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ total_meso         : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ total_owlspp       : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ total_corvid       : int  4 NA 0 NA 0 NA 0 NA NA NA ...
 $ total_other_raptor : int  2 NA 6 NA 0 NA 3 NA NA NA ...
 $ total_other_avian  : int  0 NA 11 NA 4 NA 0 NA NA NA ...
 $ total_misc         : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ first_observed     : chr  "2000-05-11" "" "2000-05-01" "2000-06-10" ...
 $ last_observed      : chr  "2000-08-05" "" "2000-08-19" "2000-09-24" ...
 $ first_nest         : chr  "2000-05-26" "" "2000-05-16" "2000-06-17" ...
 $ first_chick        : chr  "2000-06-18" "" "2000-06-07" "2000-07-22" ...
 $ first_fledge       : chr  "2000-07-08" "" "2000-06-30" "2000-08-06" ...
```

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 least terns data set:

```
summary(terns)
```

```
      year       site_name          site_name_2013_2018 site_name_1988_2001
 Min.   :2000   Length:791         Length:791          Length:791
 1st Qu.:2008   Class :character   Class :character    Class :character
 Median :2013   Mode  :character   Mode  :character    Mode  :character
 Mean   :2013
 3rd Qu.:2018
 Max.   :2023

  site_abbr           region_3           region_4            event
 Length:791         Length:791         Length:791         Length:791
 Class :character   Class :character   Class :character   Class :character
 Mode  :character   Mode  :character   Mode  :character   Mode  :character
```

```
    bp_min           bp_max           fl_min            fl_max
Min.   :   0.0   Min.   :   0.0   Min.   :   0.00   Min.   :   0.00
1st Qu.:   3.0   1st Qu.:   5.0   1st Qu.:   0.00   1st Qu.:   0.00
Median :  30.0   Median :  38.0   Median :   7.00   Median :   9.00
Mean   : 129.3   Mean   : 151.0   Mean   :  40.82   Mean   :  50.35
3rd Qu.: 127.5   3rd Qu.: 148.5   3rd Qu.:  38.00   3rd Qu.:  47.50
Max.   :1691.0   Max.   :1691.0   Max.   :1025.00   Max.   :1145.00
NA's   :8        NA's   :8        NA's   :12        NA's   :12
 total_nests      nonpred_eggs    nonpred_chicks      nonpred_fl
Min.   :   0.0   Min.   :  0.00   Min.   :   0.00   Min.   :  0.000
1st Qu.:   5.0   1st Qu.:  2.00   1st Qu.:   0.00   1st Qu.:  0.000
Median :  42.0   Median : 12.00   Median :   3.00   Median :  0.000
Mean   : 162.8   Mean   : 60.29   Mean   :  44.37   Mean   :  4.181
3rd Qu.: 164.0   3rd Qu.: 69.00   3rd Qu.:  22.00   3rd Qu.:  2.000
Max.   :1741.0   Max.   :748.00   Max.   :1063.00   Max.   :207.000
NA's   :8        NA's   :164      NA's   :198       NA's   :240
  nonpred_ad     pred_control          pred_eggs        pred_chicks
Min.   : 0.000   Length:791          Min.   :  0.00   Min.   :  0.000
1st Qu.: 0.000   Class :character    1st Qu.:  2.00   1st Qu.:  0.000
Median : 0.000   Mode  :character    Median :  6.50   Median :  2.000
Mean   : 0.851                       Mean   : 41.57   Mean   :  8.519
3rd Qu.: 1.000                       3rd Qu.: 25.50   3rd Qu.:  7.500
Max.   :22.000                       Max.   :417.00   Max.   :149.000
NA's   :234                          NA's   :737      NA's   :737
   pred_fl           pred_ad      pred_pefa          pred_coy_fox
Min.   : 0.000   Min.   : 0.00   Length:791         Length:791
1st Qu.: 0.000   1st Qu.: 0.00   Class :character   Class :character
Median : 0.000   Median : 0.50   Mode  :character   Mode  :character
Mean   : 2.365   Mean   : 2.69
3rd Qu.: 2.000   3rd Qu.: 2.00
Max.   :23.000   Max.   :41.00
NA's   :739      NA's   :733
 pred_meso           pred_owlspp         pred_corvid         pred_other_raptor
Length:791          Length:791          Length:791          Length:791
Class :character    Class :character    Class :character    Class :character
Mode  :character    Mode  :character    Mode  :character    Mode  :character



pred_other_avian     pred_misc            total_pefa      total_coy_fox
Length:791          Length:791          Min.   : 0.000   Min.   :  0.000
```

```
 Class :character   Class :character   1st Qu.: 0.000   1st Qu.:  0.000
 Mode  :character   Mode  :character   Median : 0.000   Median :  0.000
                                       Mean   : 1.741   Mean   :  9.464
                                       3rd Qu.: 0.000   3rd Qu.:  0.000
                                       Max.   :34.000   Max.   :348.000
                                       NA's   :737      NA's   :735
   total_meso      total_owlspp     total_corvid     total_other_raptor
 Min.   :  0.000   Min.   : 0.000   Min.   :  0.000   Min.   : 0.000
 1st Qu.:  0.000   1st Qu.: 0.000   1st Qu.:  0.000   1st Qu.: 0.000
 Median :  0.000   Median : 0.000   Median :  0.000   Median : 0.000
 Mean   :  5.556   Mean   : 1.455   Mean   :  7.962   Mean   : 1.712
 3rd Qu.:  0.000   3rd Qu.: 0.500   3rd Qu.:  2.000   3rd Qu.: 1.000
 Max.   :244.000   Max.   :41.000   Max.   :177.000   Max.   :43.000
 NA's   :737       NA's   :736      NA's   :739       NA's   :739
 total_other_avian   total_misc      first_observed     last_observed
 Min.   :  0.000   Min.   :  0.000   Length:791         Length:791
 1st Qu.:  0.000   1st Qu.:  0.000   Class :character   Class :character
 Median :  0.000   Median :  0.000   Mode  :character   Mode  :character
 Mean   :  8.898   Mean   :  6.566
 3rd Qu.:  2.000   3rd Qu.:  0.000
 Max.   :140.000   Max.   :168.000
 NA's   :742       NA's   :738
  first_nest         first_chick         first_fledge
 Length:791         Length:791         Length:791
 Class :character   Class :character   Class :character
 Mode  :character   Mode  :character   Mode  :character
```

1.9.2 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 example, for the least terns data set, `terns$year` selects the `year` column, which is the year of observation:

```r
terns$year
```

```
  [1] 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000
 [16] 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2004
 [31] 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004
 [46] 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004 2004
 [61] 2004 2004 2004 2005 2005 2005 2005 2005 2005 2005 2005 2005 2005 2005 2005
 [76] 2005 2005 2005 2005 2005 2005 2005 2005 2005 2005 2005 2005 2005 2005 2005
 [91] 2005 2005 2005 2005 2006 2006 2006 2006 2006 2006 2006 2006 2006 2006 2006
[106] 2006 2006 2006 2006 2006 2006 2006 2006 2006 2006 2006 2006 2006 2006 2006
[121] 2006 2006 2006 2006 2006 2006 2006 2007 2007 2007 2007 2007 2007 2007 2007
[136] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007
[151] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 2008
[166] 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008
[181] 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008 2008
[196] 2008 2008 2008 2008 2008 2008 2008 2009 2009 2009 2009 2009 2009 2009 2009
[211] 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009
[226] 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009 2009
[241] 2009 2009 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010
[256] 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010
[271] 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 2011 2011 2011
[286] 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011
[301] 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011
[316] 2011 2011 2011 2011 2011 2011 2011 2011 2012 2012 2012 2012 2012 2012 2012
[331] 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012
[346] 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 2012
[361] 2012 2012 2012 2012 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013
[376] 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013
[391] 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013
[406] 2013 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014
[421] 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014
[436] 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 2015 2015
[451] 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015
[466] 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015
[481] 2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 2016 2016 2016 2016 2016
[496] 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016
[511] 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016 2016
[526] 2016 2016 2016 2016 2016 2016 2016 2016 2017 2017 2017 2017 2017 2017 2017
[541] 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017
[556] 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 2017
[571] 2017 2017 2017 2017 2017 2017 2017 2017 2018 2018 2018 2018 2018 2018 2018
[586] 2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 2018
```

```
[601] 2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 2019 2019 2019
[616] 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019
[631] 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019 2019
[646] 2019 2019 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020
[661] 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020
[676] 2020 2020 2020 2020 2020 2020 2020 2021 2021 2021 2021 2021 2021 2021 2021
[691] 2021 2021 2021 2021 2021 2021 2021 2021 2021 2021 2021 2021 2021 2021 2021
[706] 2021 2021 2021 2021 2021 2021 2021 2021 2021 2021 2021 2021 2021 2022 2022
[721] 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022
[736] 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022 2022
[751] 2022 2022 2022 2022 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023
[766] 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023
[781] 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023
```

R provides a variety of functions to compute on columns (and other vectors of data). For instance, what if you want to know the time period the data set covers? You can use the `range` function to compute the minimum and maximum of a column:

```r
range(terns$year)
```

```
[1] 2000 2023
```

So the oldest observations are from 2000 and the newest are from 2023, although this function and output doesn't tell us whether there are observations for the years in between.

You can count the observations for each year with the `table` function:

```r
table(terns$year)
```

```

2000 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 
  29   34   31   33   37   38   40   40   41   41   42   42   42   43   45   34 
2019 2020 2021 2022 2023 
  35   35   36   36   37 
```

The `table` function is great for summarizing columns of categories, where numerical statistics like means and standard deviations aren't defined.

On the other hand, numerical statistics work well for summarizing columns of numbers. You can use the `mean` function to compute the mean of a column. For instance, let's compute the mean of the `total_nests` column, which is the total number of nests seen at a site:

```r
mean(terns$total_nests)
```

```
[1] NA
```

The result is `NA` because column is missing some values (we'll explain this in detail in Section 2.3.1). To compute the mean with only the values that are present, set `na.rm = TRUE` in the call to `mean`:

```r
mean(terns$total_nests, na.rm = TRUE)
```

```
[1] 162.8455
```

You can also use the dollar sign operator to assign values to columns. For instance, to assign `2000` to the entire `year` column:

```r
terns$year = 2000
```

Be careful when you do this, as there is no undo. Fortunately, you haven't saved this change to the least terns data set to your computer's hard drive yet, so you can reload the data set to reset it:

```r
terns = read.csv("data/2000-2023_ca_least_tern.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.

1.10 Exercises

1.10.1 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?

1.10.2 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.

1.10.3 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 least terns data set from Section 1.8.

2 Data Structures

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 `site_name` column in the California least terns data set (Section 1.8) 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:

```r
head(terns$site_name, 10)
```

```
 [1] "PITTSBURG POWER PLANT"
 [2] "ALBANY CENTRAL AVE"
 [3] "ALAMEDA POINT"
 [4] "KETTLEMAN CITY"
 [5] "OCEANO DUNES STATE VEHICULAR RECREATION AREA"
 [6] "RANCHO GUADALUPE DUNES PRESERVE"
 [7] "VANDENBERG SFB"
 [8] "SANTA CLARA RIVER MCGRATH STATE BEACH"
 [9] "ORMOND BEACH"
[10] "NBVC POINT MUGU"
```

Like all vectors, this vector is **ordered**, which just means the values, or **elements**, have specific positions. The value of the 1st element is PITTSBURG POWER PLANT, the 2nd is ALBANY CENTRAL AVE, the 5th is OCEANO DUNES STATE VEHICULAR RECREATION AREA, and so on.

Notice that the elements of this vector are all strings. 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 any of several other types of data, but not a mix these all at once.

The other columns in the least terns data frame are also vectors. For instance, the `year` column is a vector of integers:

```r
head(terns$year)
```

```
[1] 2000 2000 2000 2000 2000 2000
```

Vectors can contain any number of elements, including 0 or 1 element. Unlike mathematics, R does not distinguish between vectors and scalars (solitary values). 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(terns$year)
```

```
[1] 791
```

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

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:

```
1:3
```

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

```
-3:5
```

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

```
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:

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

```
[1] 8
```

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

```
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:

```
x - y
```

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

```
x * y
```

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

```
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:

```
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 1.9, 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 `terns` data frame:

```r
class(terns)
```

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

```r
typeof(terns)
```

```
[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:

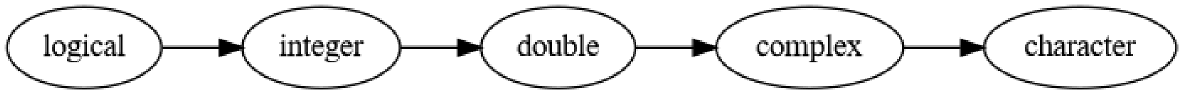


Figure 2.1: R's hierarchy of types.

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"
```

```r
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 it reads 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 you want to know how many songs are in the `rock` genre, you 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 least terns data set. To refresh your memory of what's in the data set, take a look at the structural summary:

```r
str(terns)
```

```
'data.frame':	791 obs. of  43 variables:
 $ year               : int  2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 ...
 $ site_name          : chr  "PITTSBURG POWER PLANT" "ALBANY CENTRAL AVE" "ALAMEDA POINT" "K
 $ site_name_2013_2018: chr  "Pittsburg Power Plant" "NA_NO POLYGON" "Alameda Point" "Kettle
 $ site_name_1988_2001: chr  "NA_2013_2018 POLYGON" "Albany Central Avenue" "NA_2013_2018 PO
 $ site_abbr          : chr  "PITT_POWER" "AL_CENTAVE" "ALAM_PT" "KET_CTY" ...
 $ region_3           : chr  "S.F._BAY" "S.F._BAY" "S.F._BAY" "KINGS" ...
 $ region_4           : chr  "S.F._BAY" "S.F._BAY" "S.F._BAY" "KINGS" ...
 $ event              : chr  "LA_NINA" "LA_NINA" "LA_NINA" "LA_NINA" ...
 $ bp_min             : num  15 6 282 2 4 9 30 21 73 166 ...
 $ bp_max             : num  15 12 301 3 5 9 32 21 73 167 ...
 $ fl_min             : int  16 1 200 1 4 17 11 9 60 64 ...
 $ fl_max             : int  18 1 230 2 4 17 11 9 65 64 ...
 $ total_nests        : int  15 20 312 3 5 9 32 22 73 252 ...
 $ nonpred_eggs       : int  3 NA 124 NA 2 0 NA 4 2 NA ...
 $ nonpred_chicks     : int  0 NA 81 3 0 1 27 3 0 NA ...
 $ nonpred_fl         : int  0 NA 2 1 0 0 0 NA 0 NA ...
 $ nonpred_ad         : int  0 NA 1 6 0 0 0 NA 0 NA ...
 $ pred_control       : chr  "" "" "" "" ...
 $ pred_eggs          : int  4 NA 17 NA 0 NA 0 NA NA NA ...
 $ pred_chicks        : int  2 NA 0 NA 4 NA 3 NA NA NA ...
 $ pred_fl            : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ pred_ad            : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ pred_pefa          : chr  "N" "" "N" "" ...
 $ pred_coy_fox       : chr  "N" "" "N" "" ...
 $ pred_meso          : chr  "N" "" "N" "" ...
 $ pred_owlspp        : chr  "N" "" "N" "" ...
 $ pred_corvid        : chr  "Y" "" "N" "" ...
 $ pred_other_raptor  : chr  "Y" "" "Y" "" ...
 $ pred_other_avian   : chr  "N" "" "Y" "" ...
 $ pred_misc          : chr  "N" "" "N" "" ...
 $ total_pefa         : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ total_coy_fox      : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ total_meso         : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ total_owlspp       : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ total_corvid       : int  4 NA 0 NA 0 NA 0 NA NA NA ...
 $ total_other_raptor : int  2 NA 6 NA 0 NA 3 NA NA NA ...
 $ total_other_avian  : int  0 NA 11 NA 4 NA 0 NA NA NA ...
 $ total_misc         : int  0 NA 0 NA 0 NA 0 NA NA NA ...
 $ first_observed     : chr  "2000-05-11" "" "2000-05-01" "2000-06-10" ...
 $ last_observed      : chr  "2000-08-05" "" "2000-08-19" "2000-09-24" ...
 $ first_nest         : chr  "2000-05-26" "" "2000-05-16" "2000-06-17" ...
 $ first_chick        : chr  "2000-06-18" "" "2000-06-07" "2000-07-22" ...
 $ first_fledge       : chr  "2000-07-08" "" "2000-06-30" "2000-08-06" ...
```

The `site_name`, `site_abbr`, and `event` columns are all examples of categorical data. The `region_` columns and some of the `pred_` columns also contain categorical data.

One way to check whether a feature is useful for grouping (and thus effectively categorical) is to count the number of times each value appears. You can do this with the `table` function. For instance, to count the number of times each category of `event` appears:

```
table(terns$event)
```

```
EL_NINO LA_NINA NEUTRAL 
    120     258     413 
```

Features with only a few unique values, repeated many times, are ideal for grouping. Numerical features, like `total_nests`, usually aren't good for grouping, both because of what they measure and because they tend to have many unique values, which leads to very small groups.

The `year` column can be treated as categorical or quantitative data. It's easy to imagine grouping observations by year, but years are also numerical: they have an order and we might want to do math on them. The most appropriate type for `year` depends on how we want to use it for analysis.

You can convert a column to the `factor` class with the `factor` function. Try this for the `event` column:

```
event = factor(terns$event)
event
```

```
  [1] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
 [10] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
 [19] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
 [28] LA_NINA LA_NINA NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
 [37] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
 [46] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
 [55] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
 [64] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
 [73] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
 [82] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
 [91] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[100] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[109] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[118] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[127] NEUTRAL EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[136] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[145] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[154] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[163] EL_NINO EL_NINO LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[172] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[181] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[190] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[199] LA_NINA LA_NINA LA_NINA LA_NINA NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[208] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[217] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[226] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[235] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL EL_NINO
[244] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[253] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[262] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[271] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[280] EL_NINO EL_NINO EL_NINO LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[289] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[298] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[307] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[316] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[325] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[334] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[343] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[352] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[361] LA_NINA LA_NINA LA_NINA LA_NINA NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[370] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[379] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[388] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[397] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[406] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[415] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[424] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[433] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[442] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[451] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[460] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
```

```
[469] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[478] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[487] NEUTRAL NEUTRAL NEUTRAL NEUTRAL EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[496] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[505] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[514] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[523] EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO EL_NINO
[532] EL_NINO EL_NINO NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[541] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[550] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[559] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[568] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[577] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[586] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[595] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[604] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[613] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[622] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[631] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[640] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[649] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[658] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[667] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL
[676] NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL NEUTRAL LA_NINA LA_NINA
[685] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[694] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[703] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[712] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[721] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[730] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[739] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[748] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[757] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[766] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[775] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
[784] LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA LA_NINA
Levels: EL_NINO LA_NINA NEUTRAL
```

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:

```r
levels(event)
```

A factor remembers all possible levels even if you take a subset where some of the levels aren't present:

```r
event[1:3]
```

```
[1] LA_NINA LA_NINA LA_NINA
Levels: EL_NINO LA_NINA NEUTRAL
```

```r
levels(event[1:3])
```

```
[1] "EL_NINO" "LA_NINA" "NEUTRAL"
```

This is an important way factors are different from strings (or `character` vectors). It ensures that if you plot a factor, the missing levels will still be represented on the plot.

Tip

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

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

```
[1] LA_NINA LA_NINA LA_NINA
Levels: LA_NINA
```

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:

```
0 / 0
```

```
[1] NaN
```

```
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`:

```
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:

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

```
NULL
```

```
class(NULL)
```

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

```
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`:

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

```
[1] FALSE
```

```
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 1.9, 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:

```r
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:

```r
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:

```r
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:

```r
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:

```
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:

```
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:

```
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:

```
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:

```r
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`:

```r
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:

```r
x[is_small]
```

```
 a  b 
10 20 
```

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

```
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`:

```
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`:

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

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

```
!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

1. Create a new data frame from the least terns data with the following characteristics:

    - Each entry's year is between 2010 and 2019 (inclusive).
    - Each entry reports at least 100 breeding pairs.
    - The columns are `year`, `site_name`, `bp_min`, `bp_max`, `total_nests`.

    Use this data frame for the remaining questions.

2. Count the number of entries for each site. How many sites have at least 100 breeding pairs across all 10 years?

3. Which site-year combination has the highest number of nests?

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 least terns data:

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

```
  year             site_name   site_name_2013_2018   site_name_1988_2001
1 2000 PITTSBURG POWER PLANT Pittsburg Power Plant  NA_2013_2018 POLYGON
2 2000     ALBANY CENTRAL AVE         NA_NO POLYGON Albany Central Avenue
3 2000          ALAMEDA POINT         Alameda Point  NA_2013_2018 POLYGON
   site_abbr region_3 region_4   event bp_min bp_max fl_min fl_max total_nests
1 PITT_POWER S.F._BAY S.F._BAY LA_NINA     15     15     16     18          15
2 AL_CENTAVE S.F._BAY S.F._BAY LA_NINA      6     12      1      1          20
3    ALAM_PT S.F._BAY S.F._BAY LA_NINA    282    301    200    230         312
  nonpred_eggs nonpred_chicks nonpred_fl nonpred_ad pred_control pred_eggs
1            3              0          0          0                      4
2           NA             NA         NA         NA                     NA
3          124             81          2          1                     17
  pred_chicks pred_fl pred_ad pred_pefa pred_coy_fox pred_meso pred_owlspp
1           2       0       0         N            N         N           N
2          NA      NA      NA
3           0       0       0         N            N         N           N
  pred_corvid pred_other_raptor pred_other_avian pred_misc total_pefa
1           Y                 Y                N         N          0
2                                                                  NA
3           N                 Y                Y         N          0
  total_coy_fox total_meso total_owlspp total_corvid total_other_raptor
1             0          0            0            4                  2
2            NA         NA           NA           NA                 NA
3             0          0            0            0                  6
  total_other_avian total_misc first_observed last_observed first_nest
1                 0          0     2000-05-11    2000-08-05 2000-05-26
2                NA         NA
3                11          0     2000-05-01    2000-08-19 2000-05-16
  first_chick first_fledge
1  2000-06-18   2000-07-08
2
3  2000-06-07   2000-06-30
```

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
terns[c(3, 5), c(2, 4)]
```

```
                                       site_name  site_name_1988_2001
3                                  ALAMEDA POINT NA_2013_2018 POLYGON
5 OCEANO DUNES STATE VEHICULAR RECREATION AREA NA_2013_2018 POLYGON
```

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

```
terns[c(3, 5), c("year", "site_name")]
```

```
  year                                    site_name
3 2000                                ALAMEDA POINT
5 2000 OCEANO DUNES STATE VEHICULAR RECREATION AREA
```

For data frames, it's especially common to index the rows by condition and the columns by name. For instance, let's get the `site_name` and `bp_min` columns for all year 2000 observations in the least terns data set:

```
result = terns[terns$year == 2000, c("site_name", "bp_min")]
head(result)
```

```
                                       site_name bp_min
1                           PITTSBURG POWER PLANT     15
2                              ALBANY CENTRAL AVE      6
3                                   ALAMEDA POINT    282
4                                  KETTLEMAN CITY      2
5 OCEANO DUNES STATE VEHICULAR RECREATION AREA      4
6                  RANCHO GUADALUPE DUNES PRESERVE      9
```

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"]
```

```
assertthat       base  base64enc       boot      bslib     cachem cellranger 
   "0.2.1"    "4.5.1"    "0.1-3"   "1.3-32"    "0.9.0"    "1.1.0"    "1.1.0" 
     class        cli    cluster 
  "7.3-23"    "3.6.5"  "2.1.8.1" 
```

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)

Let's visualize the California least terns data set from Section 1.8 to see how the grammar of graphics works in practice. But 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 the number of breeding pairs and the total number of nests at each site, and whether this relationship is affected by climate events. One way to show the relationship between two numerical features like these is to make a scatter plot.

3.3.1 Loading ggplot2

Before we can make the plot, 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")
```

3.3.2 Layer 1: Data

The **data layer** determines the data set(s) used to make the plot.

ggplot2 and most other Tidyverse packages are designed to work with **tidy** data, which means:

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

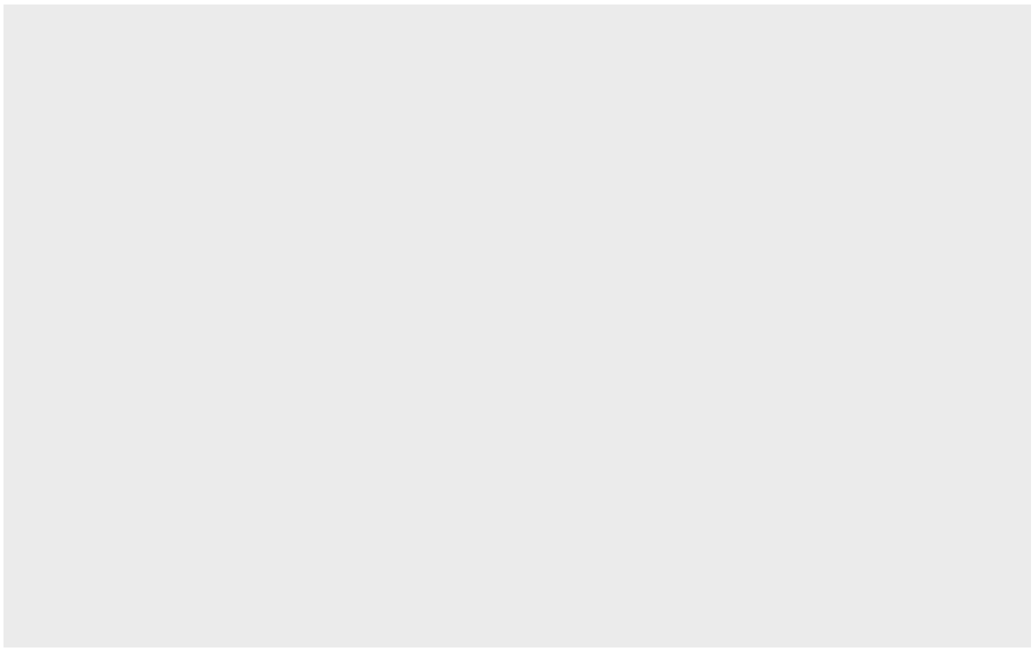
These rules ensure data are easy to read visually and access with indexing. The least terns data set satisfies all of these rules.

See also

All of the data sets we use in this reader are tidy. To learn how to tidy an untidy data set, see the Untidy & Relational Data chapter of DataLab's Intermediate R workshop reader.

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

```
ggplot(terns)
```



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

3.3.3 Layer 2: Geometry

The **geometry 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. We'll use `geom_point`, which makes a scatter plot (a plot with points):

```r
ggplot(terns) + 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.

3.3.4 Layer 3: Aesthetics

The **aesthetic 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:

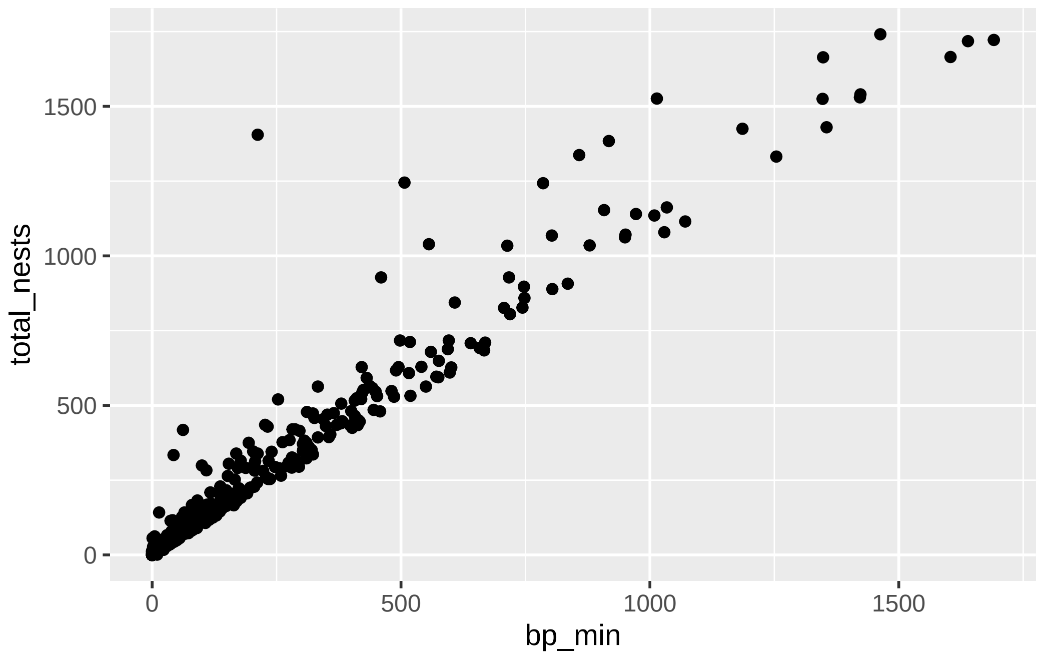
```r
aes(AESTHETIC = FEATURE, ...)
```

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 the scatter plot of breeding pairs against total nests, we'll put `bp_min` on the x-axis and `total_nests` on the y-axis. Below, we set both of these aesthetics. We also enclose all of the code for the plot in parentheses `()` so that we can put the code for each layer on a separate line, which makes the layers easier to distinguish:

```r
ggplot(terns) +
  aes(x = bp_min, y = total_nests) +
  geom_point()
```

```
Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_point()`).
```



! Important

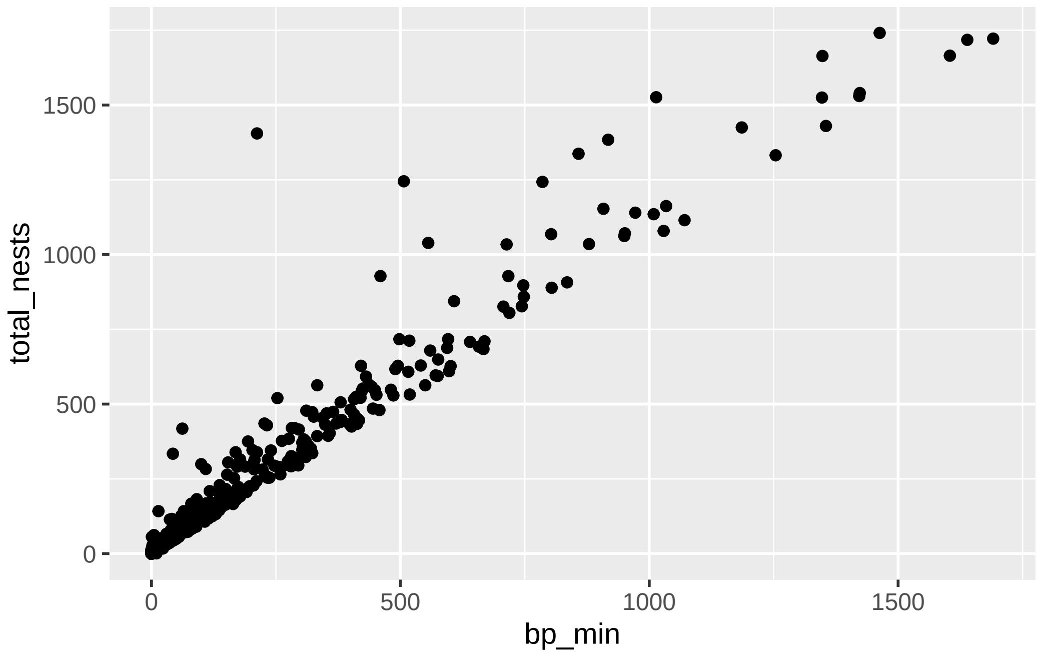
In the `aes` function, column names are never quoted.

i Note: The Old Aesthetic Layer Syntax

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:

```r
ggplot(terns, aes(x = bp_min, y = total_nests)) +
  geom_point()
```

```
Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_point()`).
```

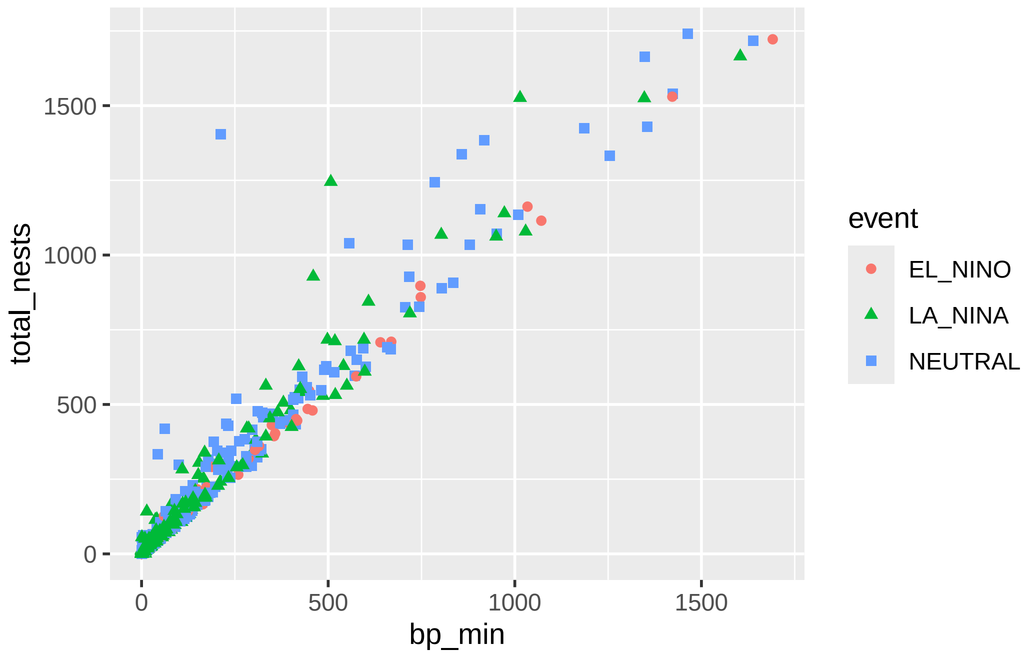


At this point, we've supplied all three layers necessary to make a plot: data, geometry, and aesthetics. The plot shows what looks like a linear relationship between number of breeding pairs and total nests. To refine the plot, you can add more layers and/or set parameters on the layers you have.

Let's add another aesthetic to the plot: we'll make the color and shape of each point correspond to `event`, the climate event for each observation:

```
ggplot(terns) +
  aes(x = bp_min, y = total_nests, color = event, shape = event) +
  geom_point()
```

```
Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Using color and shape for the same feature is redundant, but ensures that the plot is accessible to colorblind people.

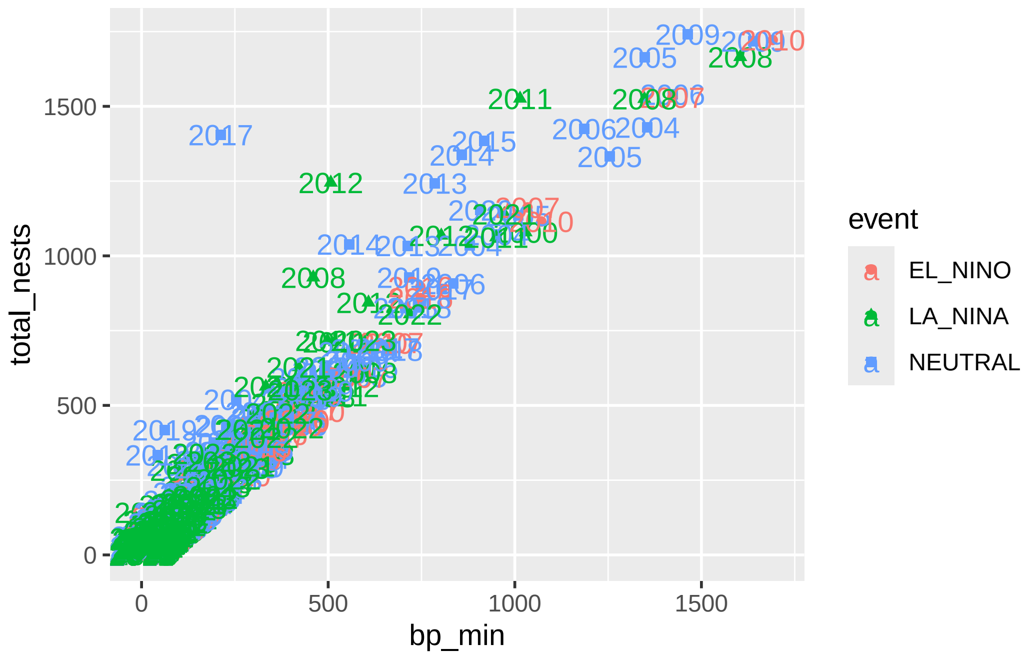
Additional Geometries

Each observation in the least terns data corresponds to a specific year and site. What if we label the points with their years? You can add text labels to a plot with `geom_text`. The required aesthetic for this geometry is `label`:

```
ggplot(terns) +
  aes(
    x = bp_min, y = total_nests,
    color = event, shape = event,
    label = year
  ) +
  geom_point() +
  geom_text()
```

```
Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_point()`).
```

```
Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_text()`).
```



The labels make the plot more difficult to read and probably would even if we made them smaller, because there are so many points on the plot. Making a high-quality visualization is typically a process of drafting and revising, similar to writing a high-quality essay. In this example, adding year labels to the plot doesn't work well, so we'll backtrack and leave them off of the plot. If accounting for year was critical to our research question, we could do it in other ways, such as by making separate plots for each year.

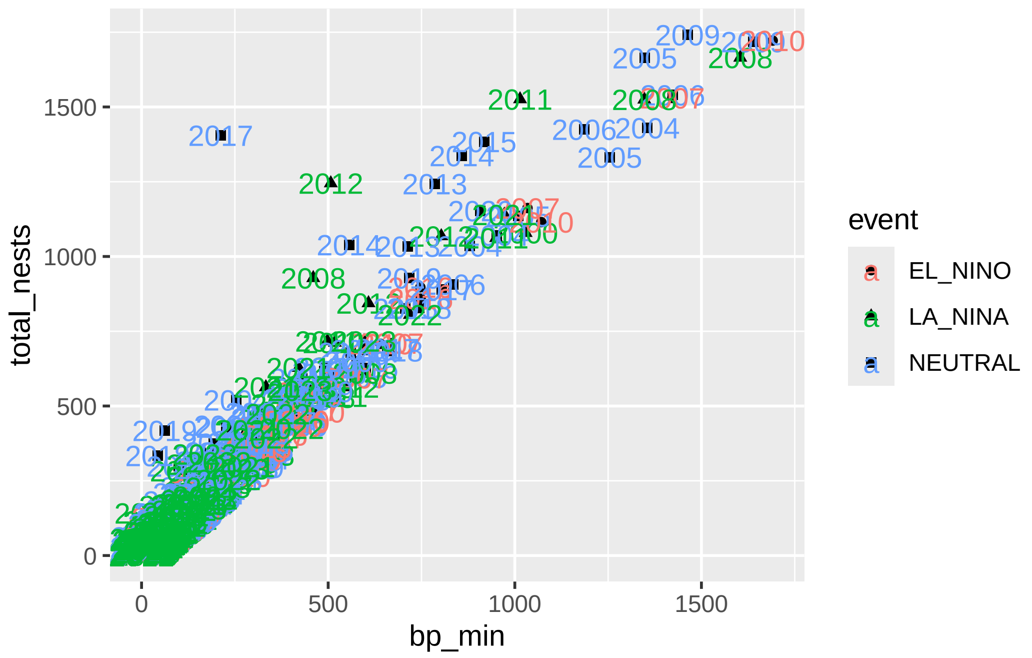
Per-geometry Aesthetics

Before we remove the labels, let's use them to demonstrate an important point about using multiple geometry and aesthetic layers: when you add an aesthetic layer to a plot, it applies to the entire plot. You can also set an aesthetic layer for an individual geometry by passing the layer as the first argument in the `geom_` function. Here's the same plot as above, but with the `color` aesthetic only set for the labels:

```
ggplot(terns) +
  aes(
    x = bp_min, y = total_nests,
    shape = event,
    label = year
  ) +
  geom_point() +
  geom_text(aes(color = event))
```

```
Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_point()`).
```

```
Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_text()`).
```



Notice that the points are no longer color-coded. Where you put aesthetic layers matters.

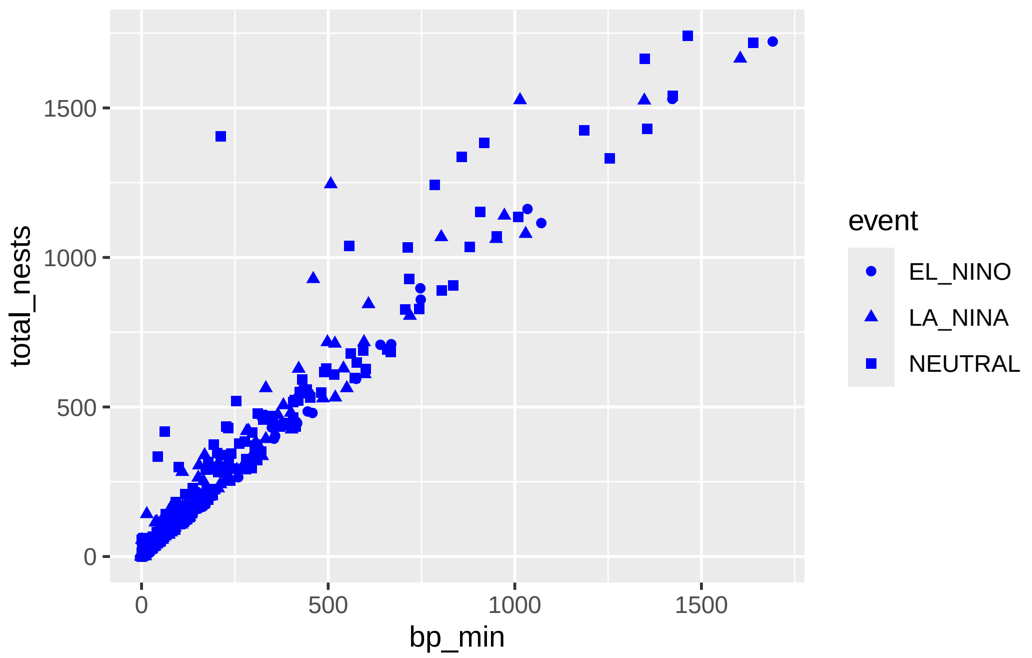
Constant Aesthetics

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 we want to make all of the points blue and use only point shape to indicate climate events:

```r
ggplot(terns) +
  aes(
    x = bp_min, y = total_nests,
    shape = event
  ) +
  geom_point(color = "blue")
```

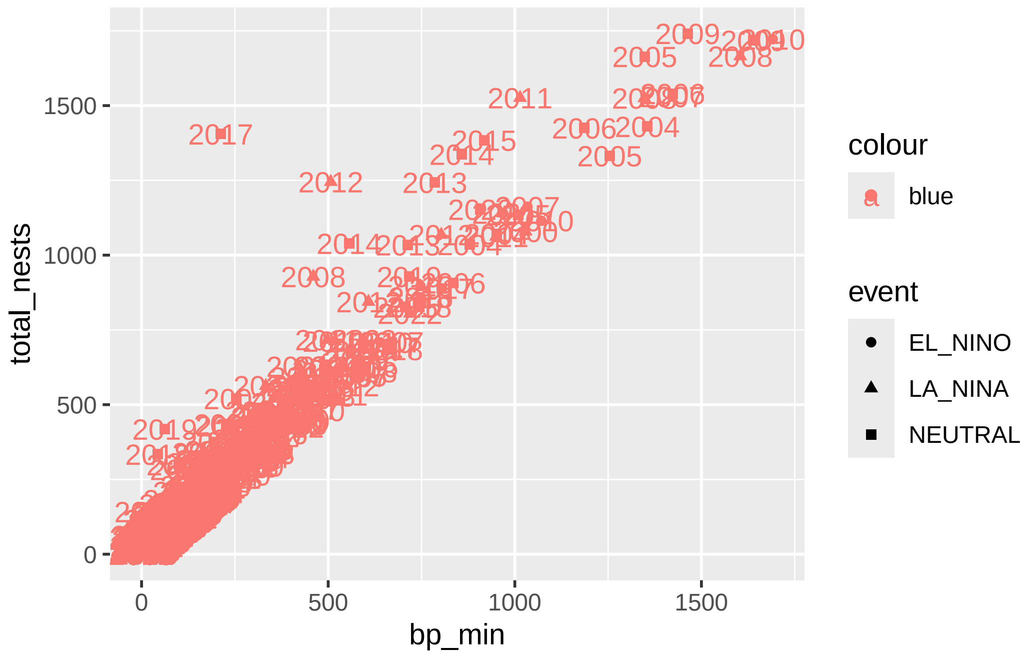
```
Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_point()`).
```



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(terns) +
  aes(
    x = bp_min, y = total_nests,
    color = "blue", shape = event,
    label = year
  ) +
  geom_point() +
  geom_text()
```

```
Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_point()`).

Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_text()`).
```



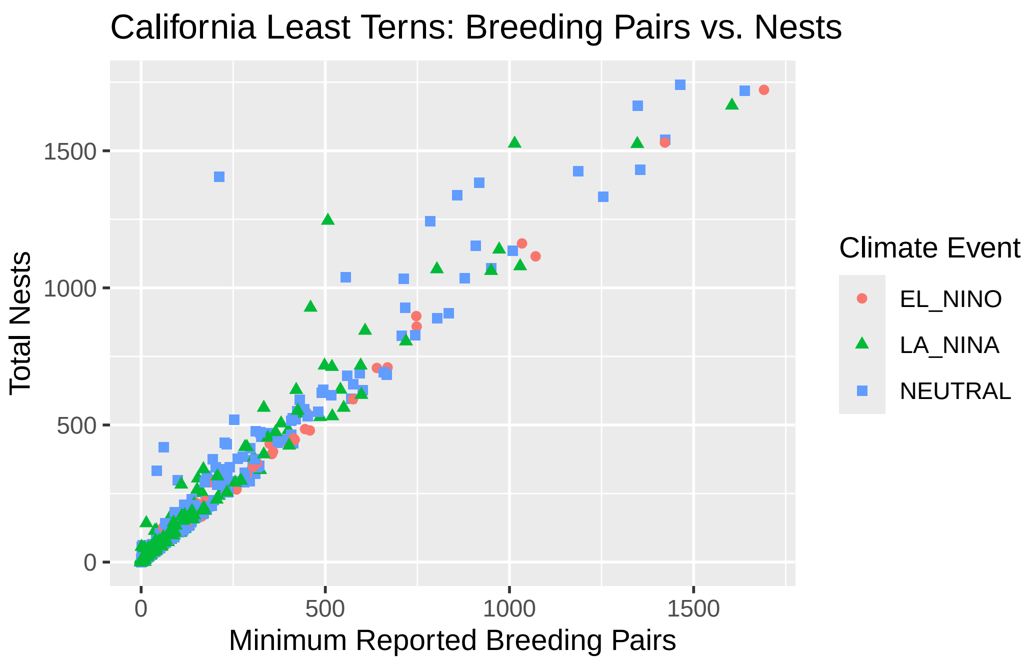
3.3.5 Layer 4: Scales

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. Visualizations should generally have a title and axis labels, to aid the viewer:

```r
ggplot(terns) +
  aes(
    x = bp_min, y = total_nests,
    color = event, shape = event
  ) +
  geom_point() +
  labs(
    x = "Minimum Reported Breeding Pairs",
    y = "Total Nests",
    color = "Climate Event", shape = "Climate Event",
    title = "California Least Terns: Breeding Pairs vs. Nests"
  )
```

```
Warning: Removed 8 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Notice that to set the title for a legend with `labs`, you can set the parameters of the same names as the corresponding aesthetics. While our plot is still far from perfect—some of the points are hard to see because of how many there are—it's now good enough to provide some insight into the relationship between number of breeding pairs and nests.

3.3.6 Saving Plots

You can use the `ggsave` function to save a plot you've assigned to a variable or the most recent plot you created (with no argument to `ggsave`):

```
ggsave("myplot.png")
```

The file format is selected automatically based on the extension. Common formats include PNG, TIFF, SVG, and PDF.

PNG and SVG are good choices for sharing visualizations online, while TIFF and PDF are good choices for print. Many journals require that visualizations be in TIFF format.

Note: R Plot Devices

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:

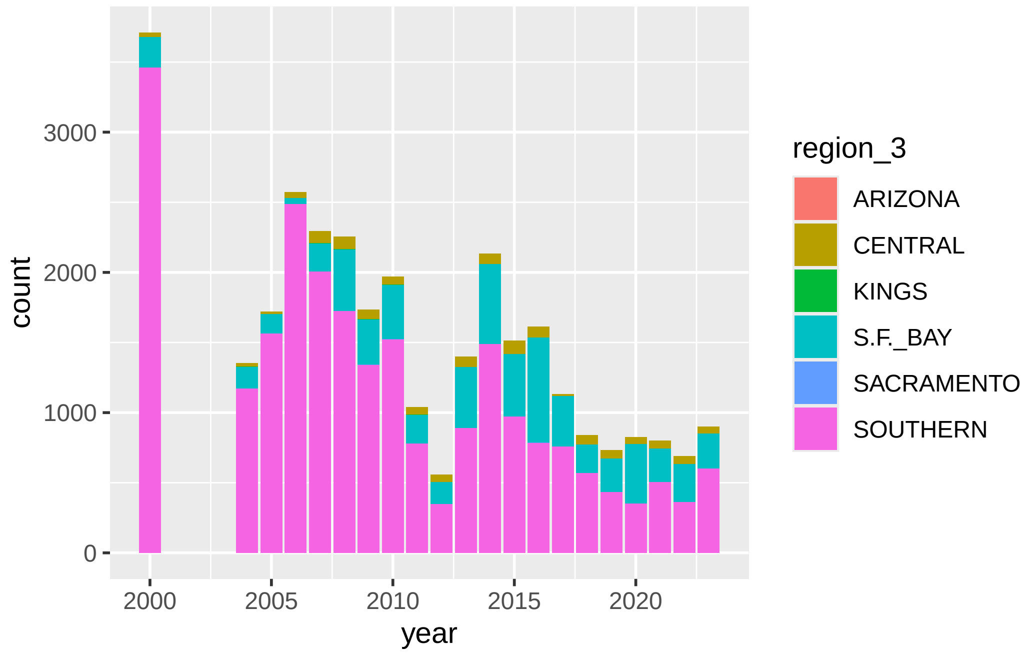
```
# Run these lines in the console, not the notebook!
jpeg("myplot.jpeg")
ggplot(terns) +
  aes(
    x = bp_min, y = total_nests,
    color = event, shape = event
  ) +
  geom_point() +
  labs(
    x = "Minimum Reported Breeding Pairs",
    y = "Total Nests",
    color = "Climate Event", shape = "Climate Event",
    title = "California Least Terns: Breeding Pairs vs. Nests"
  )
dev.off()
```

3.3.7 Example: Bar Plot

Suppose we want to visualize how many fledglings there are each year, further broken down by region. A bar plot is one appropriate way to represent this visually.

The geometry for a bar plot is `geom_bar`. Since bar plots are mainly used to display frequencies, by default the `geom_bar` function counts the number of observations in each category on the x-axis and displays these counts on the y-axis. You can make `geom_bar` display values from a column on the y-axis by setting the `weight` aesthetic:

```
ggplot(terns) +
  aes(x = year, weight = fl_min, fill = region_3) +
  geom_bar()
```

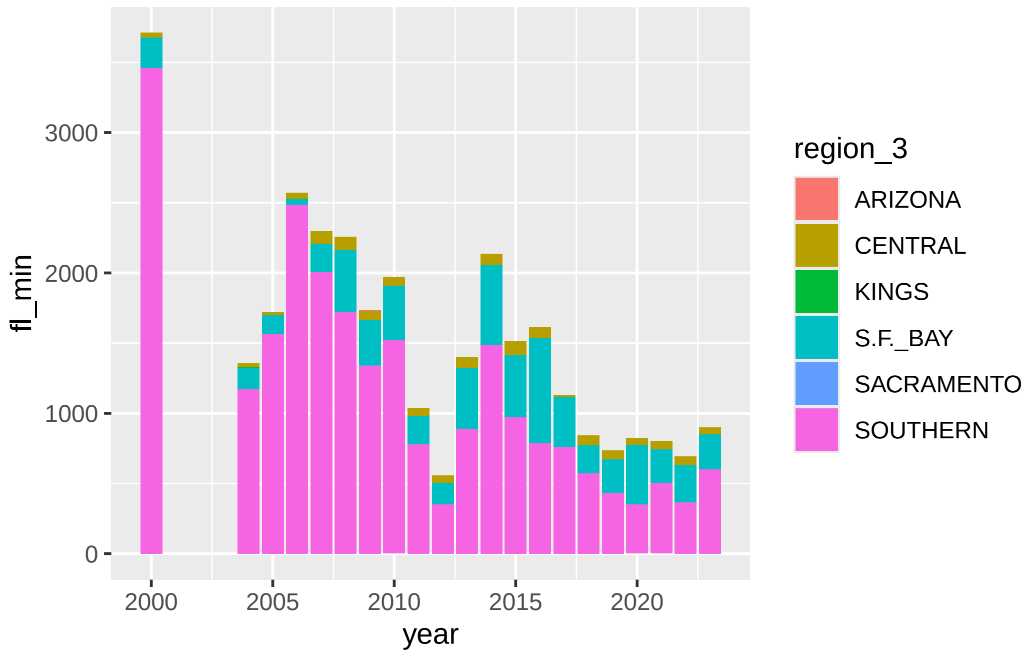


Note: Setting the Statistics Layer

Every geometry layer has a corresponding statistics layer, which transforms feature values into quantities to plot. For many geometries, the default statistics layer is the only one that makes sense.
Bar plots are an exception. The default statistics layer is `stat_count`, which counts observations. If you already have counts (or just want to display some quantities as bars), you need `stat_identity` (or the `weight` aesthetic described above). Here's one way to change the statistics layer:

```
ggplot(terns) +
  aes(x = year, y = fl_min, fill = region_3) +
  geom_bar(stat = "identity")
```

```
Warning: Removed 12 rows containing missing values or values outside the scale range
(`geom_bar()`).
```

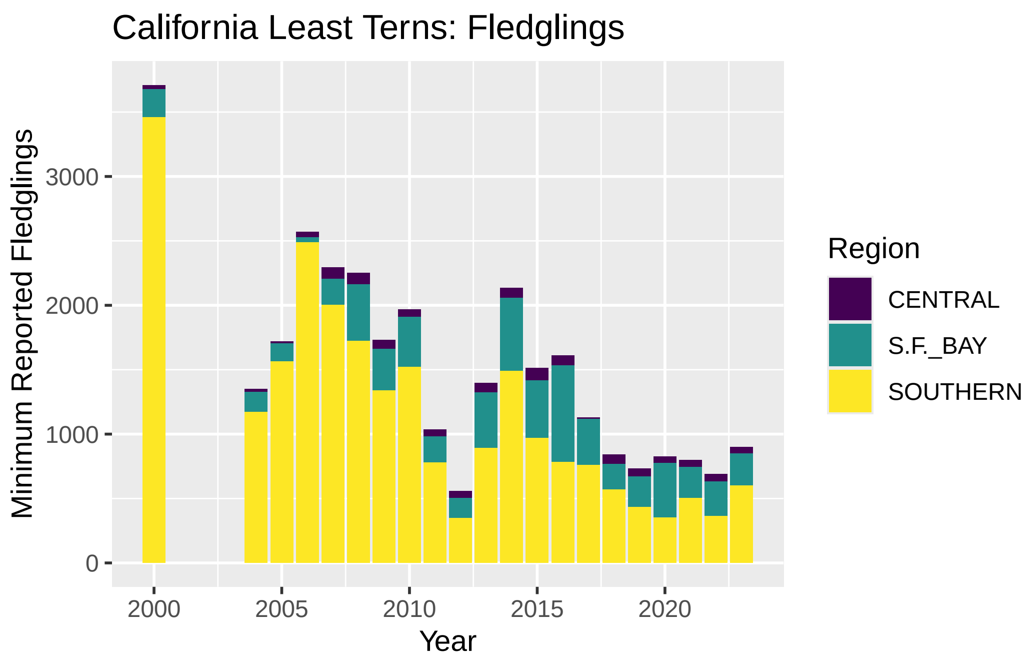


This produces the same plot as setting `weight` and using the default statistics layer `stat_count`.

The plot reveals that there are a few extraneous categories in the `region_3` column: `ARIZONA`, `KINGS`, and `SACRAMENTO`. These might or might not be erroneous—and it would be good to investigate—but they don't add anything to this plot, so let's exclude them.

Let's also change the **color map**, the palette of colors used for the categories. These are both properties of the scale layer for the `fill` aesthetic, so we'll use a `scale_fill_` function. In particular, we'll use the "viridis" color map, and since the fill color corresponds to categorical (discrete) data, we'll use `scale_fill_viridis_d`. We'll also add labels:

```
terms_to_keep = c("S.F._BAY", "CENTRAL", "SOUTHERN")
terns_filtered = terns[terns$region_3 %in% terms_to_keep, ]

ggplot(terns_filtered) +
  aes(x = year, weight = fl_min, fill = region_3) +
  geom_bar() +
  scale_fill_viridis_d() +
  labs(
    title = "California Least Terns: Fledglings",
    x = "Year",
    y = "Minimum Reported Fledglings",
    fill = "Region"
  )
```



You can read more about the viridis color map in ggplot2's documentation for this function. The plot reveals that the data set is missing 2001-2003 and that overall, fledgling counts seem to be declining in recent years.

Tip

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

3.3.8 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.

See also

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 least terns data set, we get just one result for the data set as a whole:

```r
class(terns)
```

```
[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:

```r
class(terns$year)
```

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

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(terns)`.

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 least terns data and the `class` function:

```r
lapply(terns, class)
```

```
$year
[1] "integer"

$site_name
[1] "character"

$site_name_2013_2018
[1] "character"

$site_name_1988_2001
[1] "character"

$site_abbr
[1] "character"

$region_3
[1] "character"

$region_4
[1] "character"

$event
[1] "character"

$bp_min
[1] "numeric"

$bp_max
[1] "numeric"

$fl_min
[1] "integer"

$fl_max
[1] "integer"

$total_nests
[1] "integer"

$nonpred_eggs
[1] "integer"

$nonpred_chicks
```

```
[1] "integer"

$nonpred_fl
[1] "integer"

$nonpred_ad
[1] "integer"

$pred_control
[1] "character"

$pred_eggs
[1] "integer"

$pred_chicks
[1] "integer"

$pred_fl
[1] "integer"

$pred_ad
[1] "integer"

$pred_pefa
[1] "character"

$pred_coy_fox
[1] "character"

$pred_meso
[1] "character"

$pred_owlspp
[1] "character"

$pred_corvid
[1] "character"

$pred_other_raptor
[1] "character"

$pred_other_avian
[1] "character"
```

```
$pred_misc
[1] "character"

$total_pefa
[1] "integer"

$total_coy_fox
[1] "integer"

$total_meso
[1] "integer"

$total_owlspp
[1] "integer"

$total_corvid
[1] "integer"

$total_other_raptor
[1] "integer"

$total_other_avian
[1] "integer"

$total_misc
[1] "integer"

$first_observed
[1] "character"

$last_observed
[1] "character"

$first_nest
[1] "character"

$first_chick
[1] "character"

$first_fledge
[1] "character"
```

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 least terns data, we get a character vector:

```r
sapply(terns, class)
```

```
               year           site_name site_name_2013_2018 site_name_1988_2001
          "integer"         "character"         "character"         "character"
          site_abbr            region_3            region_4               event
        "character"         "character"         "character"         "character"
             bp_min              bp_max              fl_min              fl_max
          "numeric"           "numeric"           "integer"           "integer"
        total_nests        nonpred_eggs      nonpred_chicks          nonpred_fl
          "integer"           "integer"           "integer"           "integer"
         nonpred_ad        pred_control           pred_eggs         pred_chicks
          "integer"         "character"           "integer"           "integer"
            pred_fl             pred_ad           pred_pefa        pred_coy_fox
          "integer"           "integer"         "character"         "character"
          pred_meso         pred_owlspp         pred_corvid   pred_other_raptor
        "character"         "character"         "character"         "character"
   pred_other_avian           pred_misc          total_pefa       total_coy_fox
        "character"         "character"           "integer"           "integer"
         total_meso        total_owlspp        total_corvid  total_other_raptor
          "integer"           "integer"           "integer"           "integer"
  total_other_avian          total_misc      first_observed       last_observed
          "integer"           "integer"         "character"         "character"
         first_nest         first_chick        first_fledge
        "character"         "character"         "character"
```

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 least terns 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 least terns data set:

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

```
               year           site_name site_name_2013_2018 site_name_1988_2001
              FALSE                TRUE                TRUE                TRUE
          site_abbr            region_3            region_4               event
               TRUE                TRUE                TRUE                TRUE
             bp_min              bp_max              fl_min              fl_max
              FALSE               FALSE               FALSE               FALSE
        total_nests        nonpred_eggs      nonpred_chicks          nonpred_fl
              FALSE               FALSE               FALSE               FALSE
         nonpred_ad        pred_control           pred_eggs         pred_chicks
              FALSE                TRUE               FALSE               FALSE
            pred_fl             pred_ad           pred_pefa        pred_coy_fox
              FALSE               FALSE                TRUE                TRUE
          pred_meso         pred_owlspp         pred_corvid   pred_other_raptor
               TRUE                TRUE                TRUE                TRUE
   pred_other_avian           pred_misc          total_pefa       total_coy_fox
               TRUE                TRUE               FALSE               FALSE
         total_meso        total_owlspp        total_corvid  total_other_raptor
              FALSE               FALSE               FALSE               FALSE
  total_other_avian          total_misc      first_observed       last_observed
              FALSE               FALSE                TRUE                TRUE
         first_nest         first_chick        first_fledge
               TRUE                TRUE                TRUE
```

Is it worth using R code to identify the non-numeric columns? Since there are only 43 columns in the least terns 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(terns[, is_not_number], table)
```

```
$site_name

                    ALAMEDA POINT
                               21
               ALBANY CENTRAL AVE
                                2
                     ANAHEIM LAKE
                                3
                 ARIZONA GLENDALE
                                1
BATIQUITOS LAGOON ECOLOGICAL RESERVE
                               21
     BOLSA CHICA ECOLOGICAL RESERVE
                               21
                    BURRIS ISLAND
                               18
      CHULA VISTA WILDLIFE RESERVE
                               21
           COAL OIL POINT RESERVE
                               14
  DSTREET FILL SWEETWATER MARSH NWR
                               21
    EDEN LANDING ECOLOGICAL RESERVE
                               17
                   FAIRBANKS RANCH
                                9
        GUADALUPE NIPOMO DUNES NWR
                                1
        HAYWARD REGIONAL SHORELINE
                               19
                   HOLLYWOOD BEACH
                               18
            HUNTINGTON STATE BEACH
                               21
                    KETTLEMAN CITY
                               15
```

```
                                    LA HARBOR
                                           21
                                MALIBU LAGOON
                                            7
                           MCB CAMP PENDLETON
                                           21
                       MISSION BAY FAA ISLAND
                                           21
                   MISSION BAY MARINERS POINT
                                           21
              MISSION BAY NORTH FIESTA ISLAND
                                           21
            MISSION BAY SAN DIEGO RIVER MOUTH
                                           14
                      MISSION BAY STONY POINT
                                           18
                           MONTEZUMA WETLANDS
                                           18
              NAPA SONOMA MARSH WILDLIFE AREA
                                           17
                             NAS NORTH ISLAND
                                           21
               NAVAL AMPHIBIOUS BASE CORONADO
                                           21
                              NBVC POINT MUGU
                                           21
OCEANO DUNES STATE VEHICULAR RECREATION AREA
                                           21
                                 ORMOND BEACH
                                           21
                        PITTSBURG POWER PLANT
                                           15
              RANCHO GUADALUPE DUNES PRESERVE
                                           20
                       SACRAMENTO BUFFERLANDS
                                           10
                                   SALTON SEA
                                            7
       SAN DIEGUITO LAGOON ECOLOGICAL RESERVE
                                           10
          SAN ELIJO LAGOON ECOLOGICAL RESERVE
                                           21
                            SAN PABLO BAY NWR
```

```
                                         3
      SANTA CLARA RIVER MCGRATH STATE BEACH
                                        21
 SATICOY UNITED WATER CONSERVATION DISTRICT
                                         9
                        SDIA LINDBERGH FIELD
                                        21
                   SEAL BEACH NWR ANAHEIM BAY
                                        21
                    SILVER STRAND STATE BEACH
                                         1
   SOUTH SAN DIEGO BAY UNIT SDNWR SALTWORKS
                                        21
                        TIJUANA ESTUARY NERR
                                        21
        UPPER NEWPORT BAY ECOLOGICAL RESERVE
                                        21
                              VANDENBERG SFB
                                        21
                                VENICE BEACH
                                        21

$site_name_2013_2018


                                                     3
                                         Alameda Point
                                                    21
                                          Anaheim Lake
                                                     3
                                     Batiquitos Lagoon
                                                    21
                                           Bolsa Chica
                                                    21
                                           Bufferlands
                                                    10
                                          Burris Basin
                                                    18
                                         Camp Pendleton
                                                    21
                           Chula Vista Wildlife Refuge
                                                    21
                                 Coal Oil Point Reserve
```

14
D Street Fill
21
Eden Landing Ecological Reserve
17
FAA Island
21
Fairbanks Ranch
9
Hayward Regional Shoreline
19
Hollywood Beach
18
Huntington Beach State Park
21
Kettleman
15
Lindbergh Field/Former Naval Training Center
21
Malibu Lagoon
7
Mariner's Point
21
Montezuma
18
NA_NO POLYGON
5
Napa Sonoma Marsh Wildlife Area Huichica Unit (Pond 7/7A)
17
Naval Base Coronado
42
NBVC Point Mugu
21
North Fiesta Island
21
Oceano Dunes State Vehicular Recreation Area
21
Ormond Beach
21
Pittsburg Power Plant
15
Port of LA
21

```
                 Rancho Guadalupe Dunes Preserve
                                              20
                                      Salton Sea
                                               7
                                       Saltworks
                                              21
                            San Diego River Mouth
                                              14
                              San Diequito Lagoon
                                              10
                                 San Elijo Lagoon
                                              21
                                Santa Clara River
                                              21
      Saticoy United Water Conservation District
                                               9
             Seal Beach National Wildlife Refuge
                                              21
                                     Stony Point
                                              18
                                  Tijuana Estuary
                                              21
                                Upper Newport Bay
                                              21
                                   Vandenberg AFB
                                              21
                                    Venice Beach
                                              21

$site_name_1988_2001

                Albany Central Avenue  NA_2013_2018 POLYGON
                    3                     2                  783
        NA_NO POLYGON
                    3

$site_abbr

  AL_CENTAVE      ALAM_PT       ANA_LK     ARZ_GLEN         BCER         BLER
           2           21            3            1           21           21
     BUR_ISL     COAL_OIL           CV         D_ST         ELER     FAIR_RAN
          18           14           21           21           17            9
     GND_NWR HAY_REG_SHOR           HB          HSB      KET_CTY      LA_HARB
```

```
          1           19           18           21           15           21
    MAL_LAG       MB_FAA    MB_MAR_PT       MB_NFI   MB_SDRIV_S     MB_STONY
          7           21           21           21           14           18
      MCBCP         MONT          NAB        NASNI        NSMWA OCEANO_DUNES
         21           18           21           21           17           21
     ORMOND   PITT_POWER      PT_MUGU         RGDP   S_CLAR_MCG      SAC_BUF
         21           15           21           20           21           10
       SALT     SALT_SEA    SAN_ELIJO SANDIEGU_LAG      SAT_WCD      SDIA_LF
         21            7           21           10            9           21
SEAL_BCH_NWR     SLVRSTRD       SPBNWR       TJ_RIV        UNBER      VAN_SFB
         21            1            3           21           21           21
    VEN_BCH
         21

$region_3

   ARIZONA    CENTRAL      KINGS   S.F._BAY SACRAMENTO   SOUTHERN
         1         77         15        112         10        576

$region_4

   ARIZONA    CENTRAL      KINGS   S.F._BAY SACRAMENTO   SOUTHERN    VENTURA
         1         77         15        112         10        486         90

$event

EL_NINO LA_NINA NEUTRAL
    120     258     413

$pred_control

      N   Y
342 134 315

$pred_pefa

      N   Y
143 423 225

$pred_coy_fox

      N   Y
142 535 114
```

```
$pred_meso

      N   Y 
142 552  97 

$pred_owlspp

      N   Y 
142 531 118 

$pred_corvid

      N   Y 
142 423 226 

$pred_other_raptor

      N  NN   Y 
145 437   1 208 

$pred_other_avian

      N   Y 
143 395 253 

$pred_misc

      N   Y 
159 415 217 

$first_observed

           2000-04-16 2000-04-19 2000-04-21 2000-04-22 2000-04-23 2000-04-26 
       141          1          1          1          1          1          1 
2000-04-28 2000-05-01 2000-05-04 2000-05-06 2000-05-07 2000-05-09 2000-05-11 
         5          1          3          1          2          1          1 
2000-05-21 2000-06-06 2000-06-10 2004-04-08 2004-04-11 2004-04-12 2004-04-21 
         1          1          1          1          1          2          1 
2004-04-22 2004-04-23 2004-04-27 2004-04-30 2004-05-01 2004-05-03 2004-05-07 
         2          1          1          1          1          2          1 
2004-05-09 2004-05-10 2004-05-14 2004-05-17 2004-05-31 2004-06-03 2005-04-14 
         1          1          1          1          1          1          1 
```

```
2005-04-15 2005-04-18 2005-04-19 2005-04-20 2005-04-21 2005-04-22 2005-04-23
         1          2          1          2          2          1          2
2005-04-24 2005-04-28 2005-04-29 2005-04-30 2005-05-01 2005-05-05 2005-05-07
         1          2          1          1          1          1          1
2005-05-08 2005-05-10 2005-05-13 2005-05-27 2005-05-28 2006-04-08 2006-04-10
         1          1          1          1          1          2          1
2006-04-13 2006-04-15 2006-04-16 2006-04-20 2006-04-22 2006-04-27 2006-04-28
         1          1          1          1          3          1          1
2006-05-01 2006-05-03 2006-05-08 2006-05-10 2006-05-14 2006-05-15 2006-05-20
         1          2          1          1          2          1          1
2006-06-23 2006-06-24 2007-04-16 2007-04-18 2007-04-22 2007-04-23 2007-04-24
         1          1          1          1          1          1          2
2007-04-25 2007-04-26 2007-04-28 2007-05-01 2007-05-02 2007-05-03 2007-05-11
         4          2          1          3          3          1          1
2007-05-13 2007-05-14 2007-05-18 2007-05-24 2007-06-02 2007-06-06 2007-06-08
         2          1          1          1          1          1          1
2008-04-11 2008-04-16 2008-04-23 2008-04-24 2008-04-25 2008-04-27 2008-04-28
         1          2          1          4          1          3          3
2008-05-01 2008-05-04 2008-05-05 2008-05-07 2008-05-08 2008-05-09 2008-05-12
         1          1          1          1          1          2          1
2008-05-14 2008-05-15 2008-05-21 2008-05-22 2008-05-24 2008-05-28 2008-05-31
         1          1          1          1          1          1          1
2009-04-16 2009-04-19 2009-04-22 2009-04-23 2009-04-24 2009-04-25 2009-04-26
         2          1          1          4          2          1          3
2009-04-29 2009-04-30 2009-05-02 2009-05-03 2009-05-04 2009-05-05 2009-05-06
         1          1          1          2          1          1          1
2009-05-08 2009-05-11 2009-05-13 2009-05-28 2010-04-14 2010-04-16 2010-04-17
         3          1          2          1          1          2          1
2010-04-18 2010-04-19 2010-04-21 2010-04-22 2010-04-25 2010-04-27 2010-04-28
         2          1          1          2          2          1          1
2010-04-29 2010-05-01 2010-05-02 2010-05-04 2010-05-06 2010-05-07 2010-05-13
         1          1          1          1          1          1          2
2010-05-14 2010-05-15 2010-05-16 2010-05-26 2010-05-29 2010-05-30 2010-06-09
         2          4          1          1          1          1          1
2011-04-09 2011-04-15 2011-04-18 2011-04-20 2011-04-21 2011-04-22 2011-04-24
         1          1          1          1          5          2          1
2011-04-25 2011-04-27 2011-04-28 2011-04-29 2011-05-01 2011-05-04 2011-05-05
         1          1          1          1          1          1          2
2011-05-07 2011-05-08 2011-05-11 2011-05-12 2011-05-14 2011-05-28 2011-06-11
         1          1          1          1          3          1          1
2011-06-14 2011-06-25 2012-04-14 2012-04-15 2012-04-17 2012-04-18 2012-04-19
         1          1          2          1          2          3          5
2012-04-20 2012-04-21 2012-04-22 2012-04-25 2012-04-29 2012-05-01 2012-05-03
```

```
         1          2          2          1          2          1          1 
2012-05-06 2012-05-08 2012-05-10 2012-05-11 2012-05-13 2012-05-14 2012-05-23 
         1          1          3          1          1          1          1 
2012-05-30 2012-06-09 2013-04-14 2013-04-15 2013-04-18 2013-04-21 2013-04-24 
         1          1          1          1          1          1          2 
2013-04-25 2013-04-29 2013-04-30 2013-05-01 2013-05-02 2013-05-03 2013-05-07 
         1          1          1          2          2          3          1 
2013-05-10 2013-05-11 2013-05-12 2013-05-13 2013-05-15 2013-05-17 2013-05-19 
         1          1          1          1          1          1          1 
2013-05-27 2013-06-01 2013-06-09 2013-06-30 2013-07-19 2014-04-15 2014-04-17 
         1          2          1          1          1          2          5 
2014-04-19 2014-04-21 2014-04-24 2014-04-25 2014-04-26 2014-04-30 2014-05-01 
         1          1          1          2          1          1          1 
2014-05-02 2014-05-04 2014-05-08 2014-05-09 2014-05-10 2014-05-11 2014-05-16 
         1          2          1          2          2          1          1 
2014-05-22 2014-05-24 2014-05-25 2014-05-31 2015-04-06 2015-04-15 2015-04-17 
         1          2          1          1          1          2          2 
2015-04-18 2015-04-19 2015-04-22 2015-04-23 2015-04-27 2015-04-29 2015-04-30 
         1          3          3          1          1          2          3 
2015-05-01 2015-05-07 2015-05-08 2015-05-09 2015-05-14 2015-05-17 2015-05-20 
         3          1          2          1          2          1          1 
2015-05-30 2015-06-05 2015-07-05 2015-07-07 2015-07-22 2016-04-10 2016-04-11 
         1          1          1          1          1          2          1 
2016-04-12 2016-04-13 2016-04-14 2016-04-15 2016-04-18 2016-04-20 2016-04-21 
         1          1          2          1          2          1          1 
2016-04-23 2016-04-26 2016-04-30 2016-05-01 2016-05-02 2016-05-03 2016-05-04 
         1          1          2          1          1          2          1 
2016-05-06 2016-05-07 2016-05-11 2016-05-12 2016-05-14 2016-05-18 2016-05-25 
         2          2          2          1          1          2          1 
2016-05-31 2016-06-12 2016-06-16 2016-06-18 2016-07-15 2017-04-13 2017-04-14 
         1          1          1          2          1          2          1 
2017-04-15 2017-04-16 2017-04-17 2017-04-18 2017-04-19 2017-04-20 2017-04-21 
         2          1          2          2          1          1          1 
2017-04-22 2017-04-24 2017-04-26 2017-04-29 2017-05-04 2017-05-06 2017-05-07 
         1          1          1          1          2          3          2 
2017-05-10 2017-05-11 2017-05-12 2017-05-19 2017-05-25 2017-06-03 2017-07-02 
         3          2          1          1          2          1          1 
2017-07-04 2018-03-25 2018-03-30 2018-04-13 2018-04-18 2018-04-19 2018-04-20 
         1          1          1          2          1          2          3 
2018-04-21 2018-04-25 2018-04-26 2018-04-27 2018-04-28 2018-04-29 2018-05-04 
         2          2          1          1          4          2          1 
2018-05-05 2018-05-10 2018-05-11 2018-05-13 2018-05-16 2018-05-20 2018-05-21 
         1          2          1          1          1          1          1 
```

```
2019-04-06 2019-04-12 2019-04-14 2019-04-17 2019-04-18 2019-04-19 2019-04-20
         1          3          1          1          1          3          2
2019-04-22 2019-04-24 2019-04-26 2019-04-27 2019-05-01 2019-05-02 2019-05-03
         1          1          1          4          1          1          2
2019-05-06 2019-05-08 2019-05-09 2019-05-10 2019-05-11 2019-05-12 2020-04-06
         2          1          2          1          3          1          1
2020-04-10 2020-04-11 2020-04-17 2020-04-18 2020-04-20 2020-04-21 2020-04-23
         1          1          2          1          2          1          1
2020-04-24 2020-04-25 2020-04-26 2020-04-29 2020-04-30 2020-05-01 2020-05-02
         2          2          2          2          2          1          1
2020-05-06 2020-05-08 2020-05-09 2020-05-10 2020-05-14 2020-05-17 2020-06-24
         2          3          3          1          1          1          1
2021-04-10 2021-04-12 2021-04-13 2021-04-17 2021-04-18 2021-04-19 2021-04-21
         1          1          1          1          1          2          4
2021-04-22 2021-04-23 2021-04-24 2021-04-25 2021-04-27 2021-04-28 2021-04-29
         3          1          2          1          1          2          2
2021-05-01 2021-05-06 2021-05-11 2021-05-13 2021-05-16 2021-05-21 2021-05-26
         1          2          1          2          1          1          1
2021-05-27 2021-06-04 2021-06-16 2022-04-10 2022-04-13 2022-04-15 2022-04-17
         1          1          1          1          1          1          2
2022-04-18 2022-04-21 2022-04-22 2022-04-23 2022-04-24 2022-04-26 2022-04-28
         2          2          2          1          1          2          2
2022-04-30 2022-05-05 2022-05-06 2022-05-08 2022-05-11 2022-05-14 2022-05-15
         3          1          2          2          1          1          1
2022-05-17 2022-05-19 2022-05-25 2022-05-29 2023-04-15 2023-04-18 2023-04-20
         2          2          1          1          1          1          2
2023-04-22 2023-04-23 2023-04-24 2023-04-26 2023-04-27 2023-04-28 2023-04-29
         2          4          1          1          1          2          1
2023-04-30 2023-05-01 2023-05-03 2023-05-04 2023-05-06 2023-05-07 2023-05-10
         2          1          1          1          1          2          1
2023-05-11 2023-05-17 2023-05-19 2023-05-24 2023-05-25 2023-05-27 2023-05-28
         2          1          1          1          1          1          2
2023-07-23 2023-08-04 2028-04-19
         1          1          1

$last_observed

           2000-07-21 2000-08-05 2000-08-12 2000-08-13 2000-08-14 2000-08-17
       149          1          2          1          2          1          1
2000-08-18 2000-08-19 2000-08-20 2000-08-24 2000-08-26 2000-08-30 2000-08-31
         1          1          3          1          1          1          1
2000-09-01 2000-09-05 2000-09-06 2000-09-07 2000-09-15 2000-09-24 2004-06-14
         1          1          1          1          1          1          1
```

```
2004-07-01 2004-07-18 2004-07-21 2004-07-22 2004-07-23 2004-08-02 2004-08-06
         2          1          1          1          1          1          1
2004-08-09 2004-08-10 2004-08-13 2004-08-16 2004-08-19 2004-08-22 2004-08-23
         1          1          1          1          1          1          1
2004-08-24 2004-09-03 2004-09-09 2004-09-10 2005-06-10 2005-06-17 2005-06-24
         1          1          1          1          1          1          1
2005-06-29 2005-07-07 2005-07-23 2005-07-26 2005-07-28 2005-07-29 2005-07-31
         1          1          1          1          1          1          1
2005-08-03 2005-08-05 2005-08-13 2005-08-15 2005-08-17 2005-08-18 2005-08-24
         1          1          1          1          1          1          1
2005-08-26 2005-08-27 2005-08-29 2005-09-08 2005-09-12 2005-09-15 2006-06-18
         2          1          1          2          1          1          1
2006-06-27 2006-07-13 2006-07-23 2006-07-29 2006-08-05 2006-08-07 2006-08-09
         1          1          1          1          1          2          1
2006-08-11 2006-08-16 2006-08-17 2006-08-18 2006-08-31 2006-09-02 2006-09-05
         1          2          1          1          1          1          1
2006-09-10 2006-09-11 2006-09-13 2006-09-17 2006-09-18 2007-06-24 2007-07-18
         1          1          1          1          1          1          1
2007-08-01 2007-08-06 2007-08-08 2007-08-10 2007-08-11 2007-08-15 2007-08-17
         1          1          1          1          1          2          1
2007-08-19 2007-08-20 2007-08-22 2007-08-24 2007-08-25 2007-09-02 2007-09-06
         3          3          1          2          3          1          1
2007-09-08 2007-09-09 2007-09-12 2007-09-15 2008-06-09 2008-06-10 2008-06-14
         1          1          1          1          1          1          1
2008-06-27 2008-07-18 2008-07-31 2008-08-05 2008-08-07 2008-08-10 2008-08-13
         1          2          1          1          1          1          1
2008-08-15 2008-08-17 2008-08-19 2008-08-21 2008-08-24 2008-08-26 2008-08-28
         1          3          1          1          1          1          1
2008-08-29 2008-08-31 2008-09-01 2008-09-04 2008-09-19 2008-09-20 2009-06-10
         3          2          1          1          1          1          1
2009-07-13 2009-07-17 2009-07-18 2009-07-24 2009-07-26 2009-07-30 2009-08-01
         1          2          1          1          1          1          1
2009-08-05 2009-08-06 2009-08-07 2009-08-10 2009-08-11 2009-08-12 2009-08-13
         1          1          1          1          1          1          2
2009-08-14 2009-08-15 2009-08-19 2009-08-20 2009-08-21 2009-08-22 2009-08-24
         1          1          1          1          1          1          1
2009-08-25 2009-08-27 2009-09-05 2009-09-26 2009-09-27 2010-07-01 2010-07-17
         1          1          1          1          1          2          1
2010-07-21 2010-07-22 2010-07-27 2010-07-29 2010-07-30 2010-08-01 2010-08-04
         1          1          1          3          2          1          1
2010-08-05 2010-08-07 2010-08-09 2010-08-11 2010-08-12 2010-08-13 2010-08-14
         1          1          1          2          4          1          1
2010-08-15 2010-08-17 2010-08-18 2010-08-21 2010-08-23 2010-08-26 2010-09-01
```

```
         1          1          1          1          1          1          1 
2010-09-03 2011-06-25 2011-07-06 2011-07-19 2011-07-21 2011-07-26 2011-07-27 
         1          1          1          1          1          1          1 
2011-07-29 2011-07-31 2011-08-01 2011-08-02 2011-08-03 2011-08-04 2011-08-05 
         3          1          1          1          1          1          1 
2011-08-06 2011-08-10 2011-08-11 2011-08-13 2011-08-15 2011-08-17 2011-08-18 
         2          1          1          1          1          1          1 
2011-08-20 2011-08-22 2011-08-23 2011-08-24 2011-08-26 2011-08-27 2012-06-13 
         1          1          1          1          1          3          1 
2012-06-16 2012-06-19 2012-06-22 2012-07-01 2012-07-10 2012-07-12 2012-07-13 
         1          1          1          1          1          1          2 
2012-07-17 2012-07-19 2012-07-22 2012-07-24 2012-07-25 2012-07-27 2012-08-01 
         1          1          1          1          1          1          1 
2012-08-05 2012-08-06 2012-08-08 2012-08-09 2012-08-10 2012-08-11 2012-08-12 
         1          1          1          1          2          1          1 
2012-08-16 2012-08-25 2012-08-26 2012-08-31 2012-09-01 2012-09-15 2013-05-09 
         1          1          2          1          1          2          1 
2013-06-07 2013-06-09 2013-06-10 2013-06-23 2013-06-26 2013-06-28 2013-07-01 
         1          1          1          1          1          1          1 
2013-07-10 2013-07-11 2013-07-21 2013-07-26 2013-07-28 2013-08-01 2013-08-02 
         1          1          1          1          1          1          1 
2013-08-03 2013-08-08 2013-08-11 2013-08-12 2013-08-16 2013-08-17 2013-08-21 
         1          1          2          1          1          1          3 
2013-08-26 2013-08-30 2013-09-02 2014-04-21 2014-05-03 2014-06-13 2014-06-29 
         2          1          1          1          1          1          1 
2014-07-15 2014-07-16 2014-07-24 2014-07-25 2014-07-26 2014-07-27 2014-08-02 
         1          3          1          1          1          1          1 
2014-08-03 2014-08-06 2014-08-07 2014-08-09 2014-08-10 2014-08-13 2014-08-14 
         2          2          1          1          1          1          2 
2014-08-15 2014-08-16 2014-08-24 2014-08-30 2014-09-06 2014-09-11 2014-09-16 
         1          1          1          1          1          1          1 
2015-05-12 2015-06-19 2015-06-22 2015-07-05 2015-07-09 2015-07-16 2015-07-17 
         1          1          1          1          1          2          1 
2015-07-23 2015-07-24 2015-07-27 2015-07-28 2015-07-29 2015-07-30 2015-07-31 
         1          2          1          1          3          1          1 
2015-08-03 2015-08-04 2015-08-06 2015-08-07 2015-08-09 2015-08-14 2015-08-15 
         1          1          1          1          1          1          1 
2015-08-16 2015-08-19 2015-08-20 2015-08-21 2015-08-23 2015-08-30 2015-09-02 
         1          1          2          1          1          1          1 
2015-09-03 2015-09-06 2016-06-02 2016-07-07 2016-07-10 2016-07-13 2016-07-14 
         1          1          1          1          1          2          2 
2016-07-15 2016-07-17 2016-07-20 2016-07-27 2016-07-28 2016-07-30 2016-07-31 
         3          1          2          1          2          1          1 
```

```
2016-08-02 2016-08-03 2016-08-04 2016-08-06 2016-08-11 2016-08-12 2016-08-14
         1          2          1          3          1          1          1
2016-08-15 2016-08-18 2016-08-24 2016-08-25 2016-08-26 2016-08-28 2016-09-02
         1          3          2          1          1          1          1
2016-09-03 2017-05-17 2017-05-21 2017-06-07 2017-06-24 2017-07-12 2017-07-17
         1          1          1          1          1          2          1
2017-07-21 2017-07-22 2017-07-23 2017-07-24 2017-07-28 2017-07-29 2017-07-30
         1          1          1          1          1          1          1
2017-07-31 2017-08-03 2017-08-04 2017-08-05 2017-08-10 2017-08-12 2017-08-13
         1          2          1          1          2          1          3
2017-08-18 2017-08-19 2017-08-21 2017-08-23 2017-08-24 2017-08-26 2017-08-30
         1          1          1          1          1          2          1
2017-09-11 2018-06-21 2018-06-23 2018-06-25 2018-07-05 2018-07-13 2018-07-16
         2          1          1          1          1          1          1
2018-07-21 2018-07-22 2018-07-23 2018-07-25 2018-07-26 2018-07-28 2018-07-29
         1          1          1          2          1          1          1
2018-07-30 2018-08-01 2018-08-02 2018-08-04 2018-08-08 2018-08-12 2018-08-15
         1          1          2          2          1          1          1
2018-08-23 2018-08-26 2018-08-29 2018-09-03 2018-09-06 2018-09-10 2019-05-20
         1          1          3          1          1          1          1
2019-06-12 2019-07-03 2019-07-18 2019-07-21 2019-07-24 2019-07-25 2019-07-26
         2          1          1          1          1          1          1
2019-07-30 2019-07-31 2019-08-01 2019-08-08 2019-08-09 2019-08-10 2019-08-12
         1          1          1          1          2          1          1
2019-08-14 2019-08-17 2019-08-18 2019-08-19 2019-08-21 2019-08-24 2019-08-26
         2          1          1          2          2          2          1
2019-08-27 2019-08-31 2019-09-06 2019-09-08 2019-09-16 2020-06-21 2020-07-02
         1          1          1          1          1          1          1
2020-07-15 2020-07-23 2020-07-26 2020-08-03 2020-08-05 2020-08-09 2020-08-11
         1          3          1          1          1          1          2
2020-08-12 2020-08-13 2020-08-14 2020-08-16 2020-08-19 2020-08-20 2020-08-22
         2          3          1          1          2          1          1
2020-08-26 2020-08-27 2020-08-28 2020-08-30 2020-09-04 2020-09-09 2020-09-12
         2          2          1          1          1          1          2
2020-09-17 2021-05-12 2021-06-03 2021-06-04 2021-06-06 2021-06-08 2021-07-04
         1          1          1          1          1          1          1
2021-07-11 2021-07-14 2021-07-24 2021-07-29 2021-08-01 2021-08-02 2021-08-03
         1          1          1          1          1          1          1
2021-08-04 2021-08-05 2021-08-11 2021-08-12 2021-08-13 2021-08-15 2021-08-18
         1          1          2          1          2          1          2
2021-08-19 2021-08-20 2021-08-21 2021-08-22 2021-08-26 2021-08-29 2021-09-01
         1          2          1          1          1          1          1
2021-09-10 2021-09-18 2021-09-23 2022-05-15 2022-05-27 2022-06-15 2022-06-29
```

```
         1          1          2          1          1          1          1
2022-07-01 2022-07-09 2022-07-18 2022-07-19 2022-07-21 2022-07-22 2022-07-25
         1          1          1          1          1          1          1
2022-07-26 2022-07-27 2022-07-31 2022-08-01 2022-08-02 2022-08-05 2022-08-06
         1          1          2          1          1          1          2
2022-08-09 2022-08-10 2022-08-11 2022-08-12 2022-08-17 2022-08-18 2022-08-19
         1          1          1          1          1          1          1
2022-08-24 2022-08-25 2022-08-28 2022-09-01 2022-09-02 2022-09-13 2023-05-28
         1          1          2          1          1          1          1
2023-06-14 2023-07-15 2023-07-26 2023-07-28 2023-07-29 2023-07-30 2023-08-02
         1          1          2          1          1          1          1
2023-08-03 2023-08-04 2023-08-09 2023-08-16 2023-08-17 2023-08-19 2023-08-20
         1          2          2          2          1          4          1
2023-08-24 2023-08-25 2023-08-26 2023-08-27 2023-08-28 2023-08-31 2023-09-02
         1          1          1          1          1          2          2
2023-09-09 2023-09-11 2023-09-22
         2          1          2

$first_nest

           2000-05-05 2000-05-09 2000-05-10 2000-05-11 2000-05-13 2000-05-16
       189          1          2          3          1          3          1
2000-05-18 2000-05-19 2000-05-26 2000-05-28 2000-05-29 2000-05-31 2000-06-01
         2          1          1          2          1          1          1
2000-06-06 2000-06-08 2000-06-17 2004-05-05 2004-05-08 2004-05-09 2004-05-11
         1          1          1          1          1          1          1
2004-05-13 2004-05-14 2004-05-15 2004-05-16 2004-05-19 2004-05-20 2004-05-21
         2          1          1          2          2          1          1
2004-05-22 2004-05-26 2004-05-27 2004-05-28 2004-06-03 2004-06-06 2004-06-14
         1          1          1          1          1          1          1
2004-06-17 2005-05-04 2005-05-06 2005-05-07 2005-05-08 2005-05-11 2005-05-12
         1          1          3          1          1          1          3
2005-05-13 2005-05-14 2005-05-17 2005-05-18 2005-05-19 2005-05-20 2005-05-21
         1          2          1          1          1          1          1
2005-05-26 2005-05-28 2005-06-03 2005-06-10 2005-06-15 2005-06-16 2006-05-10
         2          1          1          1          1          1          1
2006-05-12 2006-05-14 2006-05-15 2006-05-17 2006-05-18 2006-05-20 2006-05-22
         3          1          1          2          1          3          3
2006-05-23 2006-05-25 2006-05-27 2006-06-01 2006-06-09 2006-06-10 2006-06-21
         1          1          2          1          1          1          1
2007-05-12 2007-05-14 2007-05-16 2007-05-17 2007-05-18 2007-05-19 2007-05-22
         1          1          3          4          1          2          1
2007-05-23 2007-05-25 2007-05-26 2007-05-31 2007-06-02 2007-06-04 2007-06-05
```

```
         1          1          2          1          1          1          1
2007-06-06 2007-06-08 2007-06-17 2007-06-21 2007-06-27 2008-05-06 2008-05-07
         1          2          1          1          1          1          1
2008-05-09 2008-05-10 2008-05-12 2008-05-15 2008-05-16 2008-05-17 2008-05-18
         1          1          2          2          3          2          3
2008-05-21 2008-05-23 2008-05-26 2008-05-28 2008-05-31 2008-06-05 2008-06-11
         2          1          1          1          1          3          1
2008-06-19 2009-05-03 2009-05-04 2009-05-06 2009-05-07 2009-05-09 2009-05-10
         1          1          1          1          1          2          1
2009-05-12 2009-05-13 2009-05-14 2009-05-18 2009-05-20 2009-05-21 2009-05-24
         2          2          4          1          2          1          2
2009-06-03 2009-06-06 2009-06-08 2009-06-14 2009-07-29 2010-05-01 2010-05-05
         1          2          1          1          1          1          2
2010-05-06 2010-05-07 2010-05-08 2010-05-09 2010-05-13 2010-05-14 2010-05-16
         3          1          1          1          3          4          2
2010-05-21 2010-05-22 2010-05-25 2010-05-27 2010-05-28 2010-06-04 2010-06-05
         1          2          1          2          1          2          1
2010-06-06 2010-06-09 2010-06-12 2010-06-17 2010-06-18 2011-04-30 2011-05-02
         1          1          1          1          1          1          1
2011-05-04 2011-05-06 2011-05-08 2011-05-09 2011-05-12 2011-05-14 2011-05-15
         1          2          1          2          2          2          3
2011-05-17 2011-05-18 2011-05-20 2011-05-21 2011-05-28 2011-05-29 2011-06-02
         1          2          1          1          2          3          1
2011-06-07 2011-06-08 2011-06-11 2012-04-30 2012-05-02 2012-05-04 2012-05-07
         1          1          1          1          1          3          1
2012-05-08 2012-05-09 2012-05-10 2012-05-11 2012-05-12 2012-05-13 2012-05-14
         1          1          4          4          1          1          2
2012-05-15 2012-05-16 2012-05-20 2012-05-24 2012-05-26 2012-05-29 2012-05-30
         1          1          2          1          1          1          1
2012-06-01 2012-06-11 2013-05-02 2013-05-03 2013-05-06 2013-05-09 2013-05-10
         1          1          1          2          1          1          2
2013-05-11 2013-05-12 2013-05-13 2013-05-14 2013-05-15 2013-05-16 2013-05-17
         2          1          1          2          1          3          1
2013-05-26 2013-06-01 2013-06-02 2013-06-06 2013-06-07 2013-06-29 2013-08-15
         3          2          1          1          2          1          1
2014-04-29 2014-05-01 2014-05-03 2014-05-08 2014-05-09 2014-05-10 2014-05-11
         1          1          1          3          4          2          1
2014-05-12 2014-05-14 2014-05-16 2014-05-17 2014-05-21 2014-05-22 2014-05-24
         2          2          2          1          1          2          2
2014-05-28 2014-05-29 2014-05-31 2014-06-01 2014-07-05 2015-04-27 2015-05-03
         1          2          1          1          1          1          1
2015-05-06 2015-05-07 2015-05-08 2015-05-09 2015-05-11 2015-05-12 2015-05-14
         3          2          1          3          4          1          2
```

```
2015-05-15 2015-05-16 2015-05-19 2015-05-20 2015-05-21 2015-05-24 2015-05-28 
         2          1          1          2          1          1          1 
2015-05-30 2015-06-05 2015-06-13 2015-07-11 2016-04-27 2016-04-29 2016-05-01 
         1          1          1          1          1          1          1 
2016-05-04 2016-05-05 2016-05-06 2016-05-08 2016-05-11 2016-05-12 2016-05-13 
         1          2          1          1          2          3          2 
2016-05-14 2016-05-16 2016-05-18 2016-05-19 2016-05-22 2016-05-25 2016-05-26 
         2          1          1          1          1          3          1 
2016-05-28 2016-05-29 2016-06-02 2016-06-04 2016-06-16 2016-06-18 2017-04-28 
         1          1          1          1          1          2          1 
2017-04-30 2017-05-03 2017-05-04 2017-05-06 2017-05-07 2017-05-08 2017-05-10 
         2          1          1          1          2          1          2 
2017-05-11 2017-05-12 2017-05-13 2017-05-19 2017-05-21 2017-05-24 2017-05-26 
         2          3          1          3          1          1          1 
2017-05-28 2017-05-31 2017-06-02 2017-06-03 2017-06-04 2017-06-14 2018-05-03 
         1          1          2          1          1          1          1 
2018-05-04 2018-05-06 2018-05-10 2018-05-11 2018-05-12 2018-05-14 2018-05-16 
         1          2          2          2          2          2          3 
2018-05-17 2018-05-18 2018-05-19 2018-05-20 2018-05-23 2018-05-29 2018-05-30 
         2          1          1          2          1          1          2 
2018-05-31 2018-06-01 2018-06-02 2018-06-15 2019-05-03 2019-05-04 2019-05-05 
         3          1          1          1          1          1          1 
2019-05-09 2019-05-10 2019-05-11 2019-05-14 2019-05-16 2019-05-17 2019-05-22 
         4          3          5          1          1          1          1 
2019-05-23 2019-05-24 2019-05-25 2019-06-01 2019-06-02 2019-06-05 2019-06-06 
         1          1          2          1          1          1          2 
2019-06-12 2019-07-06 2020-05-07 2020-05-10 2020-05-11 2020-05-13 2020-05-14 
         1          1          1          1          1          2          2 
2020-05-15 2020-05-16 2020-05-17 2020-05-19 2020-05-20 2020-05-21 2020-05-22 
         3          1          1          1          1          2          3 
2020-05-23 2020-05-24 2020-05-28 2020-05-29 2020-05-30 2020-06-03 2020-06-11 
         1          1          1          2          2          1          1 
2020-06-12 2020-06-27 2020-07-03 2021-05-05 2021-05-06 2021-05-08 2021-05-09 
         1          1          1          2          1          1          2 
2021-05-10 2021-05-12 2021-05-13 2021-05-14 2021-05-17 2021-05-19 2021-05-20 
         1          2          1          3          1          4          2 
2021-05-21 2021-05-25 2021-05-27 2021-05-28 2021-05-30 2021-06-02 2021-06-04 
         1          1          1          1          1          2          2 
2021-06-17 2022-05-06 2022-05-07 2022-05-08 2022-05-09 2022-05-11 2022-05-14 
         1          1          1          3          1          1          1 
2022-05-15 2022-05-16 2022-05-17 2022-05-18 2022-05-19 2022-05-20 2022-05-21 
         1          1          1          1          2          2          3 
2022-05-25 2022-05-26 2022-05-27 2022-05-29 2022-06-03 2022-06-08 2022-06-11 
```

```
         1          1          1          2          2          1          1
2022-06-25 2022-06-27 2022-07-01 2023-05-07 2023-05-10 2023-05-11 2023-05-12
         1          1          1          2          1          1          2
2023-05-13 2023-05-14 2023-05-16 2023-05-19 2023-05-21 2023-05-24 2023-05-26
         1          3          2          1          1          2          1
2023-05-28 2023-05-30 2023-06-01 2023-06-02 2023-06-03 2023-06-07 2023-06-09
         1          1          2          2          1          2          1
2023-06-10 2023-06-17 2023-07-23
         1          1          1

$first_chick

           2000-05-28 2000-05-31 2000-06-01 2000-06-02 2000-06-03 2000-06-04
       262          1          1          1          2          3          1
2000-06-06 2000-06-07 2000-06-08 2000-06-09 2000-06-18 2000-06-20 2000-06-22
         1          1          1          1          1          2          1
2000-06-24 2000-06-26 2000-06-28 2000-07-22 2004-05-31 2004-06-04 2004-06-05
         1          1          1          1          2          2          1
2004-06-06 2004-06-09 2004-06-11 2004-06-14 2004-06-16 2004-06-18 2004-06-20
         1          1          2          1          1          1          1
2004-06-21 2004-06-22 2004-06-27 2004-07-04 2004-07-18 2005-05-27 2005-05-28
         1          2          1          1          1          1          1
2005-06-01 2005-06-02 2005-06-03 2005-06-05 2005-06-06 2005-06-09 2005-06-10
         2          3          1          1          1          1          4
2005-06-11 2005-06-12 2005-06-15 2005-06-26 2005-07-01 2005-07-21 2006-06-02
         1          1          1          2          1          1          1
2006-06-03 2006-06-05 2006-06-07 2006-06-09 2006-06-10 2006-06-11 2006-06-13
         1          1          1          1          1          1          1
2006-06-14 2006-06-16 2006-06-17 2006-06-20 2006-06-24 2006-06-30 2006-07-03
         2          2          2          1          2          1          1
2006-07-14 2007-06-06 2007-06-07 2007-06-09 2007-06-10 2007-06-11 2007-06-12
         1          2          2          1          3          1          2
2007-06-13 2007-06-14 2007-06-15 2007-06-20 2007-06-27 2007-06-28 2007-06-29
         1          2          3          3          1          1          1
2007-07-02 2007-07-15 2007-07-19 2008-05-30 2008-06-02 2008-06-04 2008-06-05
         1          1          1          1          1          2          4
2008-06-07 2008-06-08 2008-06-09 2008-06-12 2008-06-17 2008-06-18 2008-06-25
         1          2          1          2          1          1          1
2008-06-26 2008-06-27 2008-06-29 2008-07-06 2008-07-10 2009-05-27 2009-05-28
         2          1          1          1          1          1          2
2009-05-29 2009-05-30 2009-05-31 2009-06-03 2009-06-04 2009-06-05 2009-06-06
         1          1          2          1          4          1          3
2009-06-10 2009-06-11 2009-06-17 2009-06-20 2009-06-24 2009-06-25 2009-06-26
```

```
         1          1          1          1          1          1          1
2009-06-29 2009-07-15 2010-05-25 2010-05-26 2010-05-27 2010-05-30 2010-06-01
         1          1          1          2          3          3          1
2010-06-03 2010-06-04 2010-06-09 2010-06-10 2010-06-11 2010-06-16 2010-06-17
         3          1          1          2          1          2          1
2010-06-18 2010-06-25 2010-06-30 2010-07-01 2010-07-02 2010-07-07 2010-07-09
         1          1          2          1          1          1          1
2011-05-25 2011-05-26 2011-05-27 2011-05-29 2011-05-31 2011-06-02 2011-06-03
         1          1          1          1          1          2          1
2011-06-04 2011-06-05 2011-06-06 2011-06-08 2011-06-09 2011-06-10 2011-06-11
         1          1          1          2          1          1          1
2011-06-12 2011-06-14 2011-06-16 2011-06-17 2011-06-18 2011-06-30 2011-07-01
         1          1          1          1          1          1          1
2011-07-10 2011-07-13 2011-07-21 2012-05-20 2012-05-23 2012-05-24 2012-05-26
         1          1          1          1          1          1          1
2012-05-27 2012-05-28 2012-05-31 2012-06-01 2012-06-02 2012-06-04 2012-06-06
         1          2          1          2          1          1          3
2012-06-08 2012-06-10 2012-06-15 2012-06-20 2012-06-23 2012-06-24 2012-07-01
         1          1          2          1          3          1          1
2013-05-11 2013-05-26 2013-05-27 2013-05-29 2013-05-31 2013-06-01 2013-06-03
         1          3          1          1          1          1          1
2013-06-06 2013-06-07 2013-06-08 2013-06-09 2013-06-12 2013-06-13 2013-06-19
         2          3          1          2          1          1          1
2013-06-21 2013-06-22 2013-06-23 2013-06-26 2013-06-29 2014-05-22 2014-05-24
         1          1          1          1          1          1          1
2014-05-25 2014-05-29 2014-05-31 2014-06-01 2014-06-02 2014-06-03 2014-06-05
         1          2          2          2          1          1          3
2014-06-06 2014-06-09 2014-06-11 2014-06-13 2014-06-14 2014-06-15 2014-06-19
         2          1          1          1          2          1          1
2014-06-20 2014-06-21 2014-06-29 2014-07-02 2014-07-26 2014-08-01 2015-05-22
         1          1          1          1          1          1          1
2015-05-25 2015-05-26 2015-05-28 2015-05-29 2015-05-30 2015-05-31 2015-06-04
         1          1          2          1          1          2          6
2015-06-05 2015-06-06 2015-06-08 2015-06-10 2015-06-11 2015-06-12 2015-06-19
         1          1          1          1          1          1          1
2015-06-26 2015-06-27 2015-08-02 2016-05-22 2016-05-25 2016-05-26 2016-05-27
         1          1          1          1          1          2          1
2016-05-28 2016-05-29 2016-06-02 2016-06-04 2016-06-05 2016-06-06 2016-06-08
         2          3          1          1          1          1          5
2016-06-09 2016-06-10 2016-06-12 2016-06-16 2016-06-17 2016-06-18 2016-07-13
         1          1          2          2          1          1          1
2016-07-14 2016-07-15 2016-07-16 2017-05-22 2017-05-25 2017-05-26 2017-05-28
         1          1          1          1          3          1          3
```

```
2017-05-30 2017-06-01 2017-06-02 2017-06-03 2017-06-08 2017-06-11 2017-06-12
         1          3          3          1          1          1          1
2017-06-14 2017-06-16 2017-06-17 2017-06-21 2017-06-22 2017-06-24 2018-05-27
         2          1          1          2          1          1          1
2018-05-30 2018-05-31 2018-06-02 2018-06-03 2018-06-04 2018-06-05 2018-06-07
         1          3          1          1          1          1          4
2018-06-08 2018-06-10 2018-06-13 2018-06-14 2018-06-18 2018-06-20 2018-06-23
         2          2          1          1          1          1          1
2018-06-25 2018-06-28 2018-07-13 2019-05-30 2019-05-31 2019-06-02 2019-06-03
         1          1          1          2          2          1          2
2019-06-04 2019-06-05 2019-06-06 2019-06-07 2019-06-09 2019-06-16 2019-06-18
         1          1          1          1          1          3          1
2019-06-20 2019-06-23 2019-06-26 2019-06-27 2019-06-30 2019-07-02 2019-07-03
         1          1          2          2          1          1          2
2019-07-04 2020-05-31 2020-06-01 2020-06-03 2020-06-04 2020-06-05 2020-06-07
         1          1          1          1          3          1          1
2020-06-12 2020-06-13 2020-06-14 2020-06-19 2020-06-20 2020-06-21 2020-06-23
         2          1          2          2          1          2          1
2020-06-27 2020-07-02 2020-07-03 2020-07-05 2020-07-09 2020-07-16 2020-07-20
         1          1          2          2          1          1          1
2020-07-30 2021-05-28 2021-05-30 2021-05-31 2021-06-03 2021-06-04 2021-06-05
         1          1          1          2          3          1          1
2021-06-06 2021-06-07 2021-06-11 2021-06-12 2021-06-14 2021-06-16 2021-06-18
         1          1          2          2          1          1          1
2021-06-27 2021-06-30 2021-07-01 2021-07-02 2021-07-15 2022-05-27 2022-06-02
         2          1          2          1          1          1          5
2022-06-03 2022-06-04 2022-06-06 2022-06-08 2022-06-11 2022-06-12 2022-06-13
         1          1          1          1          1          2          1
2022-06-15 2022-06-16 2022-06-17 2022-06-19 2022-06-20 2022-06-22 2022-06-23
         1          1          3          1          1          1          2
2022-06-25 2023-05-28 2023-05-31 2023-06-01 2023-06-04 2023-06-05 2023-06-09
         1          1          1          1          1          4          1
2023-06-10 2023-06-14 2023-06-17 2023-06-18 2023-06-19 2023-06-23 2023-06-25
         2          2          1          1          1          1          1
2023-06-29 2023-07-05 2023-07-09 2023-07-27 2023-08-02 2023-08-04 3000-06-02
         1          1          1          1          1          1          1

$first_fledge

           2000-06-20 2000-06-21 2000-06-22 2000-06-24 2000-06-25 2000-06-26
       387          1          1          2          3          2          2
2000-06-30 2000-07-04 2000-07-08 2000-07-13 2000-07-15 2000-07-16 2000-07-17
         2          1          1          1          1          1          1
```

```
2000-07-20 2000-07-24 2000-08-06 2004-06-23 2004-06-25 2004-06-26 2004-06-30
         1          1          1          1          1          1          1
2004-07-02 2004-07-04 2004-07-10 2004-07-13 2004-07-14 2004-07-17 2004-07-18
         1          2          1          1          2          1          1
2004-07-25 2005-06-22 2005-06-23 2005-06-24 2005-06-25 2005-06-30 2005-07-01
         1          3          1          2          3          1          1
2005-07-02 2005-07-07 2005-07-08 2005-07-13 2005-07-18 2005-07-20 2005-07-22
         1          1          1          1          1          1          1
2005-08-27 2006-06-23 2006-06-25 2006-06-29 2006-06-30 2006-07-01 2006-07-04
         1          1          1          1          1          1          2
2006-07-07 2006-07-12 2006-07-13 2006-07-15 2006-07-17 2006-07-24 2006-07-28
         3          1          1          2          1          1          1
2006-08-02 2006-08-11 2007-06-26 2007-06-27 2007-06-28 2007-06-29 2007-06-30
         1          1          1          1          2          1          1
2007-07-02 2007-07-04 2007-07-05 2007-07-06 2007-07-10 2007-07-11 2007-07-12
         2          3          1          1          1          2          2
2007-07-13 2007-07-18 2007-07-23 2007-08-08 2008-06-20 2008-06-22 2008-06-25
         2          2          1          2          1          1          2
2008-06-26 2008-06-27 2008-06-28 2008-06-29 2008-07-02 2008-07-15 2008-07-17
         3          1          2          1          1          1          1
2008-07-18 2008-07-20 2008-07-24 2008-07-30 2008-07-31 2008-08-17 2009-06-16
         1          2          1          1          1          1          1
2009-06-20 2009-06-21 2009-06-24 2009-06-25 2009-06-26 2009-06-29 2009-06-30
         2          1          3          3          1          1          1
2009-07-01 2009-07-03 2009-07-05 2009-07-10 2009-07-15 2009-07-16 2009-07-20
         1          1          1          1          2          1          1
2009-08-07 2010-06-09 2010-06-16 2010-06-17 2010-06-18 2010-06-20 2010-06-22
         1          1          1          2          3          2          1
2010-06-24 2010-06-25 2010-06-27 2010-06-30 2010-07-01 2010-07-09 2010-07-14
         1          2          1          1          1          2          2
2010-07-21 2010-07-23 2010-07-27 2010-07-29 2010-07-30 2010-08-05 2011-06-14
         1          1          1          1          1          1          1
2011-06-17 2011-06-19 2011-06-22 2011-06-23 2011-06-25 2011-06-27 2011-06-28
         1          1          1          4          2          1          1
2011-07-01 2011-07-03 2011-07-06 2011-07-09 2011-07-11 2011-07-21 2011-07-22
         3          1          1          1          1          1          1
2011-07-24 2012-06-15 2012-06-17 2012-06-19 2012-06-21 2012-06-22 2012-06-23
         1          1          2          1          1          1          1
2012-06-24 2012-06-26 2012-06-27 2012-06-28 2012-06-29 2012-07-01 2012-07-03
         1          1          1          1          1          1          1
2012-07-07 2012-07-14 2012-07-15 2012-07-19 2012-07-21 2012-08-17 2013-06-20
         1          1          1          1          1          1          1
2013-06-21 2013-06-22 2013-06-23 2013-06-26 2013-06-27 2013-06-29 2013-06-30
```

```
         1          2          1          1          2          2          1 
2013-07-02 2013-07-04 2013-07-05 2013-07-10 2013-07-11 2013-07-17 2013-07-20 
         1          1          1          2          1          1          1 
2013-07-31 2013-08-02 2014-06-15 2014-06-16 2014-06-18 2014-06-19 2014-06-21 
         1          1          1          2          1          3          1 
2014-06-22 2014-06-23 2014-06-27 2014-06-28 2014-07-03 2014-07-04 2014-07-05 
         2          1          3          1          1          2          2 
2014-07-09 2014-07-12 2014-07-16 2014-08-09 2014-08-15 2015-06-13 2015-06-14 
         1          1          1          1          1          1          3 
2015-06-17 2015-06-18 2015-06-20 2015-06-22 2015-06-25 2015-06-26 2015-06-27 
         1          1          1          1          2          3          1 
2015-06-30 2015-07-02 2015-07-03 2015-07-04 2015-07-17 2015-08-19 2016-06-09 
         1          1          2          1          1          1          1 
2016-06-14 2016-06-15 2016-06-17 2016-06-19 2016-06-23 2016-06-24 2016-06-27 
         1          1          1          3          4          1          1 
2016-06-29 2016-07-02 2016-07-03 2016-07-06 2016-07-07 2016-07-08 2016-07-09 
         2          3          3          2          2          2          1 
2016-08-05 2017-06-11 2017-06-15 2017-06-17 2017-06-18 2017-06-20 2017-06-21 
         1          2          1          1          1          1          1 
2017-06-22 2017-06-23 2017-06-24 2017-06-27 2017-06-29 2017-07-02 2017-07-05 
         3          1          1          2          1          1          1 
2017-07-06 2017-07-07 2017-07-11 2017-07-12 2017-07-14 2017-07-25 2017-08-05 
         1          1          1          1          1          1          1 
2018-06-21 2018-06-22 2018-06-23 2018-06-24 2018-06-26 2018-06-27 2018-06-28 
         2          1          2          2          1          2          3 
2018-06-29 2018-07-04 2018-07-05 2018-07-06 2018-07-12 2018-07-14 2018-07-19 
         1          2          1          1          1          1          1 
2018-08-03 2019-06-19 2019-06-20 2019-06-21 2019-06-23 2019-06-26 2019-06-27 
         1          1          1          1          1          1          2 
2019-06-28 2019-06-29 2019-06-30 2019-07-01 2019-07-03 2019-07-04 2019-07-07 
         1          1          1          1          1          1          1 
2019-07-08 2019-07-10 2019-07-12 2019-07-13 2019-07-18 2019-07-19 2019-07-20 
         1          1          2          1          1          1          1 
2019-07-21 2019-08-01 2020-06-20 2020-06-24 2020-06-25 2020-07-02 2020-07-03 
         3          1          1          2          3          1          1 
2020-07-04 2020-07-08 2020-07-09 2020-07-13 2020-07-14 2020-07-15 2020-07-17 
         1          1          3          1          1          2          2 
2020-07-24 2020-07-26 2020-08-06 2020-08-13 2020-08-23 2022-06-25 
         1          2          1          1          1          1 
```

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 least terns data set has 6 different categories in the `region_3` column. If we want all of the rows for one region, one way to get them is by indexing:

```r
southern = terns[terns$region_3 == "SOUTHERN", ]
head(southern)
```

```
   year                             site_name
8  2000 SANTA CLARA RIVER MCGRATH STATE BEACH
9  2000                          ORMOND BEACH
10 2000                       NBVC POINT MUGU
11 2000                          VENICE BEACH
12 2000                             LA HARBOR
13 2000               SEAL BEACH NWR ANAHEIM BAY
                   site_name_2013_2018  site_name_1988_2001    site_abbr
8                    Santa Clara River NA_2013_2018 POLYGON   S_CLAR_MCG
9                         Ormond Beach NA_2013_2018 POLYGON       ORMOND
10                     NBVC Point Mugu NA_2013_2018 POLYGON      PT_MUGU
11                        Venice Beach NA_2013_2018 POLYGON      VEN_BCH
12                          Port of LA NA_2013_2018 POLYGON      LA_HARB
13 Seal Beach National Wildlife Refuge NA_2013_2018 POLYGON SEAL_BCH_NWR
   region_3 region_4   event bp_min bp_max fl_min fl_max total_nests
8  SOUTHERN  VENTURA LA_NINA     21     21      9      9          22
9  SOUTHERN  VENTURA LA_NINA     73     73     60     65          73
10 SOUTHERN  VENTURA LA_NINA    166    167     64     64         252
11 SOUTHERN SOUTHERN LA_NINA    274    294    150    200         308
12 SOUTHERN SOUTHERN LA_NINA    437    437    570    570         565
13 SOUTHERN SOUTHERN LA_NINA    107    107    180    180         107
   nonpred_eggs nonpred_chicks nonpred_fl nonpred_ad pred_control pred_eggs
8             4              3         NA         NA                     NA
9             2              0          0          0                     NA
10           NA             NA         NA         NA                     NA
11           26             NA         NA         NA                     32
12           77             NA         NA         NA                     24
13           10              3         NA         NA                     NA
   pred_chicks pred_fl pred_ad pred_pefa pred_coy_fox pred_meso pred_owlspp
8           NA      NA      NA
9           NA      NA      NA         N            N         Y           N
```

```
10          NA      NA      NA
11          20      20       3         Y            N         Y            N
12          NA      15      NA         Y            N         N            N
13          NA      NA      NA
   pred_corvid pred_other_raptor pred_other_avian pred_misc total_pefa
8                                                                  NA
9            N                 Y                N         N        NA
10                                                                 NA
11           Y                 N                N         N         2
12           N                 Y                Y         N        17
13                                                                 NA
   total_coy_fox total_meso total_owlspp total_corvid total_other_raptor
8             NA         NA           NA           NA                 NA
9             NA         NA           NA           NA                 NA
10            NA         NA           NA           NA                 NA
11             0         42            0           31                  0
12             0          0            0            0                  4
13            NA         NA           NA           NA                 NA
   total_other_avian total_misc first_observed last_observed first_nest
8                 NA         NA     2000-06-06    2000-09-05 2000-06-06
9                 NA         NA                              2000-06-08
10                NA         NA     2000-05-21    2000-08-12 2000-06-01
11                 0          0     2000-04-19    2000-08-20 2000-05-29
12                24          0     2000-04-28    2000-08-20 2000-05-10
13                NA         NA
   first_chick first_fledge
8   2000-06-28   2000-07-24
9   2000-06-26   2000-07-17
10  2000-06-24   2000-07-16
11
12  3000-06-02   2000-06-22
13
```

To get all 6 regions separately, we'd have to do this 6 times. If we want to compute something for each region, say the median of the `total_nests` column, we also have to repeat that computation 6 times. Here's what it would look like for just the `SOUTHERN` region:

```r
median(southern$total_nests, na.rm = TRUE)
```

```
[1] 64
```

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 `median` 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 medians of `total_nests` broken down by country. First, we split the data by region Since we only want to compute on the `total_nests` column, we only split that column:

```r
by_region = split(terns$total_nests, terns$region_3)
class(by_region)
```

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

```r
names(by_region)
```

```
[1] "ARIZONA"    "CENTRAL"    "KINGS"      "S.F._BAY"   "SACRAMENTO"
[6] "SOUTHERN"
```

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

```r
head(by_region$SOUTHERN)
```

```
[1]  22  73 252 308 565 107
```

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_region, median, na.rm = TRUE)
```

```
   ARIZONA    CENTRAL      KINGS   S.F._BAY SACRAMENTO   SOUTHERN 
         3         17          1         35          1         64 
```

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 `total_nests` medians by `region_3` for the least terns data set:

```
tapply(terns$total_nests, terns$region_3, median, na.rm = TRUE)
```

```
   ARIZONA    CENTRAL      KINGS    S.F._BAY SACRAMENTO   SOUTHERN
         3         17          1          35          1         64
```

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 `total_nests` medians:

```
aggregate(terns$total_nests, list(terns$region_3), mean)
```

```
     Group.1          x
1    ARIZONA  3.0000000
2    CENTRAL 21.5584416
3      KINGS  0.8666667
4   S.F._BAY         NA
5 SACRAMENTO  0.9000000
6   SOUTHERN         NA
```

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

1. Compute the total number of fledglings (with `fl_min`) for each year and region combination.
2. Another way to present the data in Section 3.3.7 is with a line plot. Use the data from part 1 to make this plot with points for each total and lines connecting the totals. Hint: find the appropriate geometries in the ggplot2 documentation.

3.5.2 Exercise

1. Compute the number of sites with no egg mortalities due to predation.
2. Of those, how many had at least one fledgling?

3.5.3 Exercise

1. Compute the range (minimum and maximum) of `year` for each site.
2. How many many sites have observations over the entire range of the data set (2000 and 2004-2023)? Hint: use the `unique` and `length` functions to find the number of unique years for each site.

4 Organizing Code

Learning Goals

After completing this chapter, learners should be able to:

- Write functions to organize and encapsulate reusable code
- Create code that only runs when a condition is satisfied
- Identify when a problem requires iteration
- Select appropriate iteration strategies for problems

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 Functions

The main way to interact with R is by calling functions, which was first explained way back in Section 1.3.4. This section explains how to write your own functions.

The Parts of a Function

Think of a function as a factory. It takes raw materials, runs them through some machinery, and produces a final product:

1. Raw materials: the inputs to a function are called **arguments**. Each argument is assigned to a **parameter**, a placeholder variable.

2. Machinery: code in the **body** of a function computes something from the arguments.

3. Final product: the output of a function is called the **return value**.

Functions are building blocks for solving larger problems. Take a divide-and-conquer approach, breaking large problems into smaller steps. Write a short function for each step. This approach makes it easier to:

- Test that each step works correctly.
- Modify, reuse, or repurpose a step.

The `function` keyword defines 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. Generally, when you define a function, you should assign it to a variable, so that you can use it later.

Tip

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.

For example, let's create a function that detects negative numbers. It should take a vector of numbers as input, compare them to zero, and then return the logical result from the comparison as output. Here's the code:

```
is_negative = function(x) x < 0
```

The name of the function, `is_negative`, describes what the function does and includes a verb. The parameter `x` is the input. The return value is the result `x < 0`.

Any time you write a function, the first thing you should do afterwards is test that it actually works. Try the `is_negative` function out on a few test cases:

```
is_negative(6)
```

```
[1] FALSE
```

```r
is_negative(-1.1)
```

```
[1] TRUE
```

```r
x = c(5, -1, -2, 0, 3)
is_negative(x)
```

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

Tip

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.

Viewing Function Definitions

Almost every command in R is a function, even the arithmetic operators and the parentheses! You can view the definition of a function by typing its name without trailing parentheses (in contrast to how you call functions).
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: 0x563b21de61c8>
<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:

```
sum
```

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

4.1.1 Example: Getting Largest Values

Let's write a another function. This one will get 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:

```
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).

Try the `get_largest` function on a few test cases to make sure it works correctly:

```
x = c(1, 10, 20, -3)
get_largest(x, 2)
```

```
[1] 20 10
```

```
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: 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 workshop 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 `get_largest` with the default `n = 5`:

```r
get_largest = function(vec, n = 5) {
  sorted = sort(vec, decreasing = TRUE)
  head(sorted, n)
}
```

After making a change, it's a good idea to test the function again:

```r
get_largest(x)
```

```
[1] 20 10  1 -3
```

```
get_largest(y)
```

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

```
get_largest(z)
```

```
[1] "t" "l" "d" "b" "a"
```

The `return` Keyword

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:

```
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
```

```
get_largest(z)
```

```
[1] "t" "l" "d" "b" "a"
```

```
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.

4.1.2 Example: Returning Multiple Values

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 Conditional Expressions

Sometimes you'll need code to do different things, depending on a condition. You can use an **if-expression** to write conditional code.

For example, suppose you want your code to generate a different greeting depending on an input name:

```
name = "Nick"

# Default greeting
greeting = "Nice to meet you!"

if (name == "Nick") {
   greeting = "Hi Nick, nice to see you again!"
}

greeting
```

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

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

Warning

The condition in an if-expression has to have length 1:

```
name = c("Nick", "Susan")

# Default greeting
greeting = "Nice to meet you!"

if (name == "Nick") {
   greeting = "Hi Nick, nice to see you again!"
}
```

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

Use the `else` keyword if you want to add an alternative when the condition is `FALSE`. So the previous code can also be written as:

```
name = "Nick"

if (name == "Nick") {
   greeting = "Hi Nick, nice to see you again!"
} else {
  # Default greeting
  greeting = "Nice to meet you!"
}

greeting
```

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

Use `else if` with a condition if you want to add an alternative to the first condition that also has its own condition. Only the first case where a condition is `TRUE` will run. You can use `else if` as many times as you want, and can also use `else`. For example:

```
name = "Susan"

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

greeting
```

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

You can create compound conditions with R's logic operators `!`, `&`, and `|` (see Section 2.4.5). For example:

```
name1 = "Wesley"
name2 = "Nick"

if (name1 == "Wesley" & name2 == "Nick") {
  greeting = "These are the authors."
} else {
  greeting = "Who are these people?!"
}

greeting
```

```
[1] "These are the authors."
```

You can write an if-statement inside of another if-statement. This is called **nesting** if-statements. Nesting is useful when you want to check a condition, do some computations, and then check another condition under the assumption that the first condition was `True`.

> Tip
>
> If-statements correspond to **special cases** in your code. Lots of special cases in code makes the code harder to understand and maintain. If you find yourself using lots of if-statements, especially nested if-statements, consider whether there is a more general strategy or way to write the code.

4.3 Iteration

R is powerful tool for automating tasks that have repetitive steps. For example, you can:

- Apply a transformation to an entire column of data.
- Compute distances between all pairs from a set of points.
- Read a large collection of files from disk in order to combine and analyze the data they contain.
- Simulate how a system evolves over time from a specific set of starting parameters.
- Scrape data from the pages of a website.

You can implement concise, efficient solutions for these kinds of tasks in R by using **iteration**, which means repeating a computation many times. R provides four different strategies for writing iterative code:

1. Vectorization, where a function is implicitly called on each element of a vector. This was introduced in Section 2.1.3.
2. Apply functions, where a function is explicitly called on each element of a data structure. This was introduced in Section 3.4.
3. Loops, where an expression is evaluated repeatedly until some condition is met.
4. Recursion, where a function calls itself.

Vectorization is the most efficient and concise iteration strategy, but also the least flexible, because it only works with specific functions and with vectors. Apply functions are more flexible—they work with any function and any data structure with elements—but less efficient and less concise. Loops and recursion provide the most flexibility but are the least concise. Recursion tends to be the least efficient iteration strategy in R.

The rest of this section explains how to write loops and how to choose which iteration strategy to use. We assume you're already comfortable with vectorization and have at least some familiarity with apply functions.

4.3.1 For-Loops

A **for-loop** evaluates the expressions in its body once for each element of a data structure. The syntax of a for-loop is:

```r
# DATA is the data structure
#
# X is one element of DATA, assigned automatically at the beginning of each
# iteration
#
for (X in DATA) {
  # Expression(s) to repeat.
}
```

For example, to print out all the column names of the `terns` data, you can write:

```r
for (name in names(terns)) {
  message(name)
}
```

```
year

site_name

site_name_2013_2018

site_name_1988_2001

site_abbr

region_3

region_4

event

bp_min

bp_max
```

fl_min

fl_max

total_nests

nonpred_eggs

nonpred_chicks

nonpred_fl

nonpred_ad

pred_control

pred_eggs

pred_chicks

pred_fl

pred_ad

pred_pefa

pred_coy_fox

pred_meso

pred_owlspp

pred_corvid

pred_other_raptor

pred_other_avian

```
pred_misc

total_pefa

total_coy_fox

total_meso

total_owlspp

total_corvid

total_other_raptor

total_other_avian

total_misc

first_observed

last_observed

first_nest

first_chick

first_fledge
```

Within the body of a for-loop, you can compute values, check conditions, etc.

Oftentimes you'll want to save the result of the code you perform within a for-loop. The easiest way to do this is by creating an empty vector and using `c` or `append` to append values to it:

```
values = c(10, 12, 11, 2, 3)
diffs = c()

for (i in seq(1, length(values) - 1)) {
  a = values[[i]]
  b = values[[i + 1]]
  diffs = append(diffs, a - b)
}

diffs
```

```
[1] -2  1  9 -1
```

This example is just for demonstration, since R's built-in `diff` function is a much more concise and efficient way to compute differences. The point is that when you use a loop, you have to create a data structure to store the results yourself.

⚠ Warning

Appending values to a vector, as in the example, is quite inefficient. If you need to use a loop to produce a lot of results (say, more than 100):

1. Before the loop, create a result vector that's the expected length of the final result. Fill vector with 0s or some other placeholder value. You can use functions such as `integer`, `numeric`, and `character` to do this.
2. In the loop, use indexing to replace the elements of the result vector as they're computed.

Using this strategy, which is called **pre-allocation** the code in the example becomes:

```
values = c(10, 12, 11, 2, 3)
diffs = integer(length(values) - 1)

for (i in seq(1, length(values) - 1)) {
  a = values[[i]]
  b = values[[i + 1]]
  diffs[[i]] = a - b
}

diffs
```

```
[1] -2  1  9 -1
```

4.3.2 Planning for Iteration

At first it might seem difficult to decide if and what kind of iteration to use. Start by thinking about whether you need to do something over and over. If you don't, then you probably don't need to use iteration. If you do, then try iteration strategies in this order:

1. Vectorization
2. Apply functions (or the purrr package's `map` functions)
    - Try an apply function if iterations are independent.
3. Loops
    - Try a for-loop if some iterations depend on others.
4. Recursion (which isn't covered here)
    - Convenient for naturally recursive tasks (like Fibonacci), but often there are faster solutions.

Start by writing the code for just one iteration. Make sure that code works; it's easy to test code for one iteration.

When you have one iteration working, then try using the code with an iteration strategy (you will have to make some small changes). If it doesn't work, try to figure out which iteration is causing the problem. One way to do this is to use `message` to print out information. Then try to write the code for the broken iteration, get that iteration working, and repeat this whole process.

4.4 Case Study: CA Hospital Utilization

The California Department of Health Care Access and Information (HCAI) requires hospitals in the state to submit detailed information each year about how many beds they have and the total number of days for which each bed was occupied. The HCAI publishes the data to the California Open Data Portal. Let's use R to read data from 2016 to 2023 and investigate whether hospital utilization is noticeably different in and after 2020.

The data set consists of a separate Microsoft Excel file for each year. Before 2018, HCAI used a data format (in Excel) called ALIRTS. In 2018, they started collecting more data and switched to a data format called SIERA. The 2018 data file contains a **crosswalk** that shows the correspondence between SIERA columns and ALIRTS columns.

! Important

Click here to download the CA Hospital Utilization data set (8 Excel files).
If you haven't already, we recommend you create a directory for this workshop. In your workshop directory, create a `data/ca_hospitals` subdirectory. Download and save the data set in the `data/ca_hospitals` subdirectory.

When you need to solve a programming problem, get started by writing some comments that describe the problem, the inputs, and the expected output. Try to be concrete. This will help you clarify what you're trying to achieve and serve as a guiding light while you work.

As a programmer (or any kind of problem-solver), you should always be on the lookout for ways to break problems into smaller, simpler steps. Think about this when you frame a problem. Small steps are easier to reason about, implement, and test. When you complete one, you also get a nice sense of progress towards your goal.

For the CA Hospital Utilization data set, our goal is to investigate whether there was a change in hospital utilization in 2020. Before we can do any investigation, we need to read the files into R. The files all contain tabular data and have similar formats, so let's try to combine them into a single data frame. We'll say this in the framing comments:

```
# Read the CA Hospital Utilization data set into R. The inputs are yearly Excel
# files (2016-2023) that need to be combined. The pre-2018 files have a
# different format from the others. The result should be a single data frame
# with information about bed and patient counts.
#
# After reading the data set, we'll investigate utilization in 2020.
```

"Investigate utilization" is a little vague, but for an exploratory data analysis, it's hard to say exactly what to do until you've started working with the data.

We need to read multiple files, but we can simplify the problem by starting with just one. Let's start with the 2023 data. It's in an Excel file, which you can read with the `read_excel` function from the readxl package. If it's your first time using the readxl package, you'll need to install it:

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

The `read_excel` function requires the path to the file as the first argument. You can optionally provide the sheet name or number (starting from 1) as the second argument. Open up the Excel file in your computer's spreadsheet program and take a look. There are multiple sheets, and the data about beds and patients are in the second sheet. Back in R, read just the second sheet:

```r
library("readxl")

path = "data/ca_hospitals/hosp23_util_data_final.xlsx"
sheet = read_excel(path, sheet = 2)
```

```
New names:
* `` -> `...355`
```

```r
head(sheet)
```

```
# A tibble: 6 x 355
  Description            FAC_NO FAC_NAME FAC_STR_ADDR FAC_CITY FAC_ZIP FAC_PHONE
  <chr>                  <chr>  <chr>    <chr>        <chr>    <chr>   <chr>
1 FINAL 2023 UTILIZATIO~ FINAL~ FINAL 2~ FINAL 2023 ~ FINAL 2~ FINAL ~ FINAL 20~
2 Page                   1.0    1.0      1.0          1.0      1.0     1.0
3 Column                 1.0    1.0      1.0          1.0      1.0     1.0
4 Line                   2.0    1.0      3.0          4.0      5.0     6.0
5 <NA>                   10601~ ALAMEDA~ 2070 CLINTO~ ALAMEDA  94501   51023337~
6 <NA>                   10601~ ALTA BA~ 2450 ASHBY ~ BERKELEY 94705   510-
655-~
# i 348 more variables: FAC_ADMIN_NAME <chr>, FAC_OPERATED_THIS_YR <chr>,
#   FAC_OP_PER_BEGIN_DT <chr>, FAC_OP_PER_END_DT <chr>,
#   FAC_PAR_CORP_NAME <chr>, FAC_PAR_CORP_BUS_ADDR <chr>,
#   FAC_PAR_CORP_CITY <chr>, FAC_PAR_CORP_STATE <chr>, FAC_PAR_CORP_ZIP <chr>,
#   REPT_PREP_NAME <chr>, SUBMITTED_DT <chr>, REV_REPT_PREP_NAME <chr>,
#   REVISED_DT <chr>, CORRECTED_DT <chr>, LICENSE_NO <chr>,
#   LICENSE_EFF_DATE <chr>, LICENSE_EXP_DATE <chr>, LICENSE_STATUS <chr>, ...
```

The first four rows contain metadata about the columns. The first hospital, Alameda Hospital, is listed in the fifth row. So let's remove the first four rows:

```r
sheet = sheet[-(1:4), ]
head(sheet)
```

```
# A tibble: 6 x 355
  Description FAC_NO    FAC_NAME         FAC_STR_ADDR FAC_CITY FAC_ZIP FAC_PHONE
  <chr>       <chr>     <chr>            <chr>        <chr>    <chr>   <chr>
1 <NA>        106010735 ALAMEDA HOSPITAL 2070 CLINTO~ ALAMEDA  94501   51023337~
2 <NA>        106010739 ALTA BATES SUMM~ 2450 ASHBY ~ BERKELEY 94705   510-
655-~
```

```
3 <NA>        106010776 UCSF BENIOFF CH~ 747 52ND ST~ OAKLAND  94609   510-
428-~
4 <NA>        106010811 FAIRMONT HOSPIT~ 15400 FOOTH~ SAN LEA~ 94578   51043748~
5 <NA>        106010844 ALTA BATES SUMM~ 2001 DWIGHT~ BERKELEY 94704   510-
655-~
6 <NA>        106010846 HIGHLAND HOSPIT~ 1411 EAST 3~ OAKLAND  94602   51043748~
# i 348 more variables: FAC_ADMIN_NAME <chr>, FAC_OPERATED_THIS_YR <chr>,
#   FAC_OP_PER_BEGIN_DT <chr>, FAC_OP_PER_END_DT <chr>,
#   FAC_PAR_CORP_NAME <chr>, FAC_PAR_CORP_BUS_ADDR <chr>,
#   FAC_PAR_CORP_CITY <chr>, FAC_PAR_CORP_STATE <chr>, FAC_PAR_CORP_ZIP <chr>,
#   REPT_PREP_NAME <chr>, SUBMITTED_DT <chr>, REV_REPT_PREP_NAME <chr>,
#   REVISED_DT <chr>, CORRECTED_DT <chr>, LICENSE_NO <chr>,
#   LICENSE_EFF_DATE <chr>, LICENSE_EXP_DATE <chr>, LICENSE_STATUS <chr>, ...
```

Some data sets also have metadata in the last rows, so let's check for that here:

```
tail(sheet)
```

```
# A tibble: 6 x 355
  Description FAC_NO    FAC_NAME         FAC_STR_ADDR FAC_CITY FAC_ZIP FAC_PHONE
  <chr>       <chr>     <chr>            <chr>        <chr>    <chr>   <chr>
1 <NA>        106574010 SUTTER DAVIS HO~ 2000 SUTTER~ DAVIS    95616   (530) 75~
2 <NA>        106580996 ADVENTIST HEALT~ 726 FOURTH ~ MARYSVI~ 95901   530-
751-~
3 <NA>        206100718 COMMUNITY SUBAC~ 3003 NORTH ~ FRESNO   93703   559-
459-~
4 <NA>        206274027 WESTLAND HOUSE   100 BARNET ~ MONTEREY 93940   83162453~
5 <NA>        206351814 HAZEL HAWKINS M~ 900 SUNSET ~ HOLLIST~ 95023   831-
635-~
6 <NA>        <NA>      n = 506          <NA>         <NA>     <NA>    <NA>
# i 348 more variables: FAC_ADMIN_NAME <chr>, FAC_OPERATED_THIS_YR <chr>,
#   FAC_OP_PER_BEGIN_DT <chr>, FAC_OP_PER_END_DT <chr>,
#   FAC_PAR_CORP_NAME <chr>, FAC_PAR_CORP_BUS_ADDR <chr>,
#   FAC_PAR_CORP_CITY <chr>, FAC_PAR_CORP_STATE <chr>, FAC_PAR_CORP_ZIP <chr>,
#   REPT_PREP_NAME <chr>, SUBMITTED_DT <chr>, REV_REPT_PREP_NAME <chr>,
#   REVISED_DT <chr>, CORRECTED_DT <chr>, LICENSE_NO <chr>,
#   LICENSE_EFF_DATE <chr>, LICENSE_EXP_DATE <chr>, LICENSE_STATUS <chr>, ...
```

Sure enough, the last row contains what appears to be a count of the hospitals rather than a hospital. Let's remove it by calling `head` with a negative number of elements, which removes that many elements from the end:

```r
sheet = head(sheet, -1)
tail(sheet)
```

```
# A tibble: 6 x 355
  Description FAC_NO    FAC_NAME         FAC_STR_ADDR FAC_CITY FAC_ZIP FAC_PHONE
  <chr>       <chr>     <chr>            <chr>        <chr>    <chr>   <chr>
1 <NA>        106571086 WOODLAND MEMORI~ 1325 COTTON~ WOODLAND 95695   (530) 66~
2 <NA>        106574010 SUTTER DAVIS HO~ 2000 SUTTER~ DAVIS    95616   (530) 75~
3 <NA>        106580996 ADVENTIST HEALT~ 726 FOURTH ~ MARYSVI~ 95901   530-
751-~
4 <NA>        206100718 COMMUNITY SUBAC~ 3003 NORTH ~ FRESNO   93703   559-
459-~
5 <NA>        206274027 WESTLAND HOUSE   100 BARNET ~ MONTEREY 93940   83162453~
6 <NA>        206351814 HAZEL HAWKINS M~ 900 SUNSET ~ HOLLIST~ 95023   831-
635-~
# i 348 more variables: FAC_ADMIN_NAME <chr>, FAC_OPERATED_THIS_YR <chr>,
#   FAC_OP_PER_BEGIN_DT <chr>, FAC_OP_PER_END_DT <chr>,
#   FAC_PAR_CORP_NAME <chr>, FAC_PAR_CORP_BUS_ADDR <chr>,
#   FAC_PAR_CORP_CITY <chr>, FAC_PAR_CORP_STATE <chr>, FAC_PAR_CORP_ZIP <chr>,
#   REPT_PREP_NAME <chr>, SUBMITTED_DT <chr>, REV_REPT_PREP_NAME <chr>,
#   REVISED_DT <chr>, CORRECTED_DT <chr>, LICENSE_NO <chr>,
#   LICENSE_EFF_DATE <chr>, LICENSE_EXP_DATE <chr>, LICENSE_STATUS <chr>, ...
```

There are a lot of columns in sheet, so let's make a list of just a few that we'll use for analysis. We'll keep:

- Columns with facility name, location, and operating status
- All of the columns whose names start with `TOT`, because these are totals for number of beds, number of census-days, and so on.
- Columns about acute respiratory beds, with names that contain `RESPIRATORY`, because they might also be relevant.

We can use the stringr package, which provides string processing functions, to help us get the `TOT` and `RESPIRATORY` column names. If it's your first time using the stringr package, you'll have to install it:

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

We can use stringr's `str_starts` function to check whether column names start with `TOT` and its `str_detect` function to check whether column names contain `RESPIRATORY`:

```r
library("stringr")
```

```
Warning: package 'stringr' was built under R version 4.5.2
```

```r
facility_cols = c(
  "FAC_NAME", "FAC_CITY", "FAC_ZIP", "FAC_OPERATED_THIS_YR", "FACILITY_LEVEL",
  "TEACH_HOSP", "COUNTY", "PRIN_SERVICE_TYPE"
)

cols = names(sheet)
tot_cols = cols[str_starts(cols, "TOT")]
respiratory_cols = cols[str_detect(cols, "RESPIRATORY")]
```

The `TOT` and `RESPIRATORY` columns all contain numbers, but the element type is `character`, so let's cast to numbers them with `as.numeric`. We'll also add a column with the year:

```r
numeric_cols = c(tot_cols, respiratory_cols)
sheet[numeric_cols] = lapply(sheet[numeric_cols], as.numeric)

sheet$year = 2023
```

Now we'll select only the columns we identified as useful:

```r
sheet = sheet[c("year", facility_cols, numeric_cols)]

head(sheet)
```

```
# A tibble: 6 x 22
   year FAC_NAME FAC_CITY FAC_ZIP FAC_OPERATED_THIS_YR FACILITY_LEVEL TEACH_HOSP
  <dbl> <chr>    <chr>    <chr>   <chr>                <chr>          <chr>
1  2023 ALAMEDA~ ALAMEDA  94501   Yes                  Parent Facili~ No
2  2023 ALTA BA~ BERKELEY 94705   Yes                  Parent Facili~ No
3  2023 UCSF BE~ OAKLAND  94609   Yes                  Parent Facili~ No
4  2023 FAIRMON~ SAN LEA~ 94578   Yes                  Consolidated ~ No
5  2023 ALTA BA~ BERKELEY 94704   Yes                  Consolidated ~ No
6  2023 HIGHLAN~ OAKLAND  94602   Yes                  Parent Facili~ No
# i 15 more variables: COUNTY <chr>, PRIN_SERVICE_TYPE <chr>,
#   TOT_LIC_BEDS <dbl>, TOT_LIC_BED_DAYS <dbl>, TOT_DISCHARGES <dbl>,
#   TOT_CEN_DAYS <dbl>, TOT_ALOS_CY <dbl>, TOT_ALOS_PY <dbl>,
#   ACUTE_RESPIRATORY_CARE_LIC_BEDS <dbl>,
#   ACUTE_RESPIRATORY_CARE_LIC_BED_DAYS <dbl>,
#   ACUTE_RESPIRATORY_CARE_DISCHARGES <dbl>,
#   ACUTE_RESPIRATORY_CARE_INTRA_TRANSFERS <dbl>, ...
```

Lowercase names are easier to type, so let's also make all of the names lowercase with stringr's `str_to_lower` function:

```r
names(sheet) = str_to_lower(names(sheet))
head(sheet)
```

```
# A tibble: 6 x 22
   year fac_name fac_city fac_zip fac_operated_this_yr facility_level teach_hosp
  <dbl> <chr>    <chr>    <chr>   <chr>                <chr>          <chr>
1  2023 ALAMEDA~ ALAMEDA  94501   Yes                  Parent Facili~ No
2  2023 ALTA BA~ BERKELEY 94705   Yes                  Parent Facili~ No
3  2023 UCSF BE~ OAKLAND  94609   Yes                  Parent Facili~ No
4  2023 FAIRMON~ SAN LEA~ 94578   Yes                  Consolidated ~ No
5  2023 ALTA BA~ BERKELEY 94704   Yes                  Consolidated ~ No
6  2023 HIGHLAN~ OAKLAND  94602   Yes                  Parent Facili~ No
# i 15 more variables: county <chr>, prin_service_type <chr>,
#   tot_lic_beds <dbl>, tot_lic_bed_days <dbl>, tot_discharges <dbl>,
#   tot_cen_days <dbl>, tot_alos_cy <dbl>, tot_alos_py <dbl>,
#   acute_respiratory_care_lic_beds <dbl>,
#   acute_respiratory_care_lic_bed_days <dbl>,
#   acute_respiratory_care_discharges <dbl>,
#   acute_respiratory_care_intra_transfers <dbl>, ...
```

We've successfully read one of the files! Since the 2018-2023 files all have the same format, it's likely that we can use almost the same code for all of them. Any time you want to reuse code, it's a sign that you should write a function, so that's what we'll do. We'll take all of the code we have so far and put it in the body of a function called `read_hospital_data`, adding some comments to indicate the steps:

```r
read_hospital_data = function() {
  # Read the 2nd sheet of the file.
  path = "data/ca_hospitals/hosp23_util_data_final.xlsx"
  sheet = read_excel(path, sheet = 2)

  # Remove the first 4 and last row.
  sheet = sheet[-(1:4), ]
  sheet = head(sheet, -1)
  # Select only a few columns of interest.
  facility_cols = c(
    "FAC_NAME", "FAC_CITY", "FAC_ZIP", "FAC_OPERATED_THIS_YR",
    "FACILITY_LEVEL", "TEACH_HOSP", "COUNTY", "PRIN_SERVICE_TYPE"
  )

  cols = names(sheet)
  tot_cols = cols[str_starts(cols, "TOT")]
  respiratory_cols = cols[str_detect(cols, "RESPIRATORY")]

  numeric_cols = c(tot_cols, respiratory_cols)
  sheet[numeric_cols] = lapply(sheet[numeric_cols], as.numeric)

  sheet$year = 2023

  sheet = sheet[c("year", facility_cols, numeric_cols)]

  # Rename the columns to lowercase.
  names(sheet) = str_to_lower(names(sheet))

  sheet
}
```

As it is, the function still only reads the 2023 file. The other files have different paths, so the first thing we need to do is make the `path` variable a parameter. We'll also make a `year` parameter, for the year value inserted as a column:

```r
read_hospital_data = function(path, year) {
  # Read the 2nd sheet of the file.
  sheet = read_excel(path, sheet = 2)

  # Remove the first 4 and last row.
  sheet = sheet[-(1:4), ]
  sheet = head(sheet, -1)

  # Select only a few columns of interest.
  facility_cols = c(
    "FAC_NAME", "FAC_CITY", "FAC_ZIP", "FAC_OPERATED_THIS_YR",
    "FACILITY_LEVEL", "TEACH_HOSP", "COUNTY", "PRIN_SERVICE_TYPE"
  )
  cols = names(sheet)
  tot_cols = cols[str_starts(cols, "TOT")]
  respiratory_cols = cols[str_detect(cols, "RESPIRATORY")]

  numeric_cols = c(tot_cols, respiratory_cols)
  sheet[numeric_cols] = lapply(sheet[numeric_cols], as.numeric)

  sheet$year = year

  sheet = sheet[c("year", facility_cols, numeric_cols)]

  # Rename the columns to lowercase.
  names(sheet) = str_to_lower(names(sheet))

  sheet
}
```

Test the function out on a few of the files to make sure it works correctly:

```
head(
  read_hospital_data(
    "data/ca_hospitals/hosp23_util_data_final.xlsx", 2023
  )
)
```

```
New names:
* `` -> `...355`
```

```
# A tibble: 6 x 22
   year fac_name fac_city fac_zip fac_operated_this_yr facility_level teach_hosp
  <dbl> <chr>    <chr>    <chr>   <chr>                <chr>          <chr>
1  2023 ALAMEDA~ ALAMEDA  94501   Yes                  Parent Facili~ No
2  2023 ALTA BA~ BERKELEY 94705   Yes                  Parent Facili~ No
3  2023 UCSF BE~ OAKLAND  94609   Yes                  Parent Facili~ No
4  2023 FAIRMON~ SAN LEA~ 94578   Yes                  Consolidated ~ No
5  2023 ALTA BA~ BERKELEY 94704   Yes                  Consolidated ~ No
6  2023 HIGHLAN~ OAKLAND  94602   Yes                  Parent Facili~ No
# i 15 more variables: county <chr>, prin_service_type <chr>,
#   tot_lic_beds <dbl>, tot_lic_bed_days <dbl>, tot_discharges <dbl>,
#   tot_cen_days <dbl>, tot_alos_cy <dbl>, tot_alos_py <dbl>,
#   acute_respiratory_care_lic_beds <dbl>,
#   acute_respiratory_care_lic_bed_days <dbl>,
#   acute_respiratory_care_discharges <dbl>,
#   acute_respiratory_care_intra_transfers <dbl>, ...
```

```r
head(
  read_hospital_data(
    "data/ca_hospitals/hosp21_util_data_final-revised-06.15.2023.xlsx", 2021
  )
)
```

```
# A tibble: 6 x 22
   year fac_name fac_city fac_zip fac_operated_this_yr facility_level teach_hosp
  <dbl> <chr>    <chr>    <chr>   <chr>                <chr>          <chr>
1  2021 ALAMEDA~ ALAMEDA  94501   Yes                  Parent Facili~ No
2  2021 ALTA BA~ BERKELEY 94705   Yes                  Parent Facili~ No
3  2021 UCSF BE~ OAKLAND  94609   Yes                  Parent Facili~ No
4  2021 FAIRMON~ SAN LEA~ 94578   Yes                  Consolidated ~ No
5  2021 ALTA BA~ BERKELEY 94704   Yes                  Consolidated ~ No
6  2021 HIGHLAN~ OAKLAND  94602   Yes                  Parent Facili~ No
# i 15 more variables: county <chr>, prin_service_type <chr>,
#   tot_lic_beds <dbl>, tot_lic_bed_days <dbl>, tot_discharges <dbl>,
#   tot_cen_days <dbl>, tot_alos_cy <dbl>, tot_alos_py <dbl>,
#   acute_respiratory_care_lic_beds <dbl>,
#   acute_respiratory_care_lic_bed_days <dbl>,
#   acute_respiratory_care_discharges <dbl>,
#   acute_respiratory_care_intra_transfers <dbl>, ...
```

The function appears to work correctly for two of the files, so let's work towards trying it on all of the 2018-2023 files. We can use the built-in `list.files` function to get the paths to the files by setting `full.names = TRUE` (otherwise it just returns the names of the files):

```r
paths = list.files("data/ca_hospitals/", full.names = TRUE)
paths
```

```
[1] "data/ca_hospitals//hosp16_util_data_final.xlsx"
[2] "data/ca_hospitals//hosp17_util_data_final.xlsx"
[3] "data/ca_hospitals//hosp18_util_data_final.xlsx"
[4] "data/ca_hospitals//hosp19_util_data_final.xlsx"
[5] "data/ca_hospitals//hosp20_util_data_final-revised-06.15.2023.xlsx"
[6] "data/ca_hospitals//hosp21_util_data_final-revised-06.15.2023.xlsx"
[7] "data/ca_hospitals//hosp22_util_data_final_revised_11.28.2023.xlsx"
[8] "data/ca_hospitals//hosp23_util_data_final.xlsx"
```

In addition to the file paths, we also need the year for each file. Fortunately, the last two digits of the year are included in each file's name. We can write a function to get these. We'll use R's built-in `basename` function to get the file names from the paths, and stringr's `str_sub` function to get a substring (the year) from the name:

```
get_hospital_year = function(path) {
  name = basename(path)
  year = str_sub(name, 5, 6)
  as.integer(year) + 2000
}

get_hospital_year(paths[1])
```

```
[1] 2016
```

With that done, we need to read the pre-2018 files. In the 2018 file, the fourth sheet is a crosswalk that shows which columns in the pre-2018 files correspond to columns in the later files. Let's write some code to read the crosswalk. First, read the sheet:

```
path = "data/ca_hospitals/hosp18_util_data_final.xlsx"
cwalk = read_excel(path, sheet = 4)
head(cwalk)
```

```
# A tibble: 6 x 6
   Page  Line Column `SIERA Dataset Header (2018)` ALIRTS Dataset Header~1 Notes
  <dbl> <dbl>  <dbl> <chr>                         <chr>                   <chr>
1     1     1      1 FAC_NAME                      FAC_NAME                <NA>
2     1     2      1 FAC_NO                        OSHPD_ID                <NA>
3     1     3      1 FAC_STR_ADDR                  <NA>                    Addr~
4     1     4      1 FAC_CITY                      FAC_CITY                <NA>
5     1     5      1 FAC_ZIP                       FAC_ZIPCODE             <NA>
6     1     6      1 FAC_PHONE                     FAC_PHONE               <NA>
# i abbreviated name: 1: `ALIRTS Dataset Header (2017)`
```

The new and old column names are in the fourth and fifth columns, respectively, so we'll get just those:

```
cwalk = cwalk[, 4:5]
head(cwalk)
```

```
# A tibble: 6 x 2
  `SIERA Dataset Header (2018)` `ALIRTS Dataset Header (2017)`
  <chr>                         <chr>
1 FAC_NAME                      FAC_NAME
2 FAC_NO                        OSHPD_ID
3 FAC_STR_ADDR                  <NA>
4 FAC_CITY                      FAC_CITY
5 FAC_ZIP                       FAC_ZIPCODE
6 FAC_PHONE                     FAC_PHONE
```

Finally, let's turn the `cwalk` data frame into a named vector. We'll make the old column names the names and the new column names the elements. This way we can easily look up the new name for any of the old columns by indexing. Some of the column names in `cwalk` have extra spaces at the end, so we'll use stringr's `str_trim` function to remove them:

```r
cwalk_names = str_trim(cwalk[[2]])
cwalk = str_trim(cwalk[[1]])
names(cwalk) = cwalk_names

head(cwalk)
```

```
     FAC_NAME       OSHPD_ID           <NA>       FAC_CITY    FAC_ZIPCODE
   "FAC_NAME"       "FAC_NO" "FAC_STR_ADDR"     "FAC_CITY"      "FAC_ZIP"
    FAC_PHONE
  "FAC_PHONE"
```

We can now define a new version of the `read_hospital_data` function that uses the crosswalk to change the column names when `year < 2018`. Let's also change function to exclude columns with `ALOS` in the name, because they have no equivalent in the pre-2018 files:

```r
read_hospital_data = function(path, year) {
  # Read the 2nd sheet of the file.
  sheet = read_excel(path, sheet = 2)

  # Remove the first 4 and last row.
  sheet = sheet[-(1:4), ]
  sheet = head(sheet, -1)

  # Fix pre-2018 column names.
  if (year < 2018) {
    new_names = cwalk[names(sheet)]
    new_names[is.na(new_names)] = names(sheet)[is.na(new_names)]
    names(sheet) = new_names
  }

  # Select only a few columns of interest.
  facility_cols = c(
    "FAC_NAME", "FAC_CITY", "FAC_ZIP", "FAC_OPERATED_THIS_YR",
    "FACILITY_LEVEL", "TEACH_HOSP", "COUNTY", "PRIN_SERVICE_TYPE"
  )

  cols = names(sheet)
  tot_cols = cols[str_starts(cols, "TOT")]
  respiratory_cols = cols[str_detect(cols, "RESPIRATORY")]

  numeric_cols = c(tot_cols, respiratory_cols)
  numeric_cols = numeric_cols[!str_detect(numeric_cols, "ALOS")]
  sheet[numeric_cols] = lapply(sheet[numeric_cols], as.numeric)

  sheet$year = year

  sheet = sheet[c("year", facility_cols, numeric_cols)]

  # Rename the columns to lowercase.
  names(sheet) = str_to_lower(names(sheet))

  sheet
}
```

Now we can test the function on all of the files. We'll use a for-loop to iterate over all of the paths, read the data for each one, and store the result in a list:

```
hosps = list()

for (i in seq_along(paths)) {
  path = paths[[i]]
  year = get_hospital_year(path)
  hosp = read_hospital_data(path, year)
  hosps[[i]] = hosp
}
```

```
New names:
```

```
New names:
New names:
* `PSY_CENS_PATIENT_TOTL` -> `PSY_CENS_PATIENT_TOTL...126`
* `PSY_CENS_PATIENT_TOTL` -> `PSY_CENS_PATIENT_TOTL...130`
* `PSY_CENS_PATIENT_TOTL` -> `PSY_CENS_PATIENT_TOTL...141`
```

```r
length(hosps)
```

```
[1] 8
```

We can use the do.call and rbind functions to bind the rows, or stack, the list of data frames:

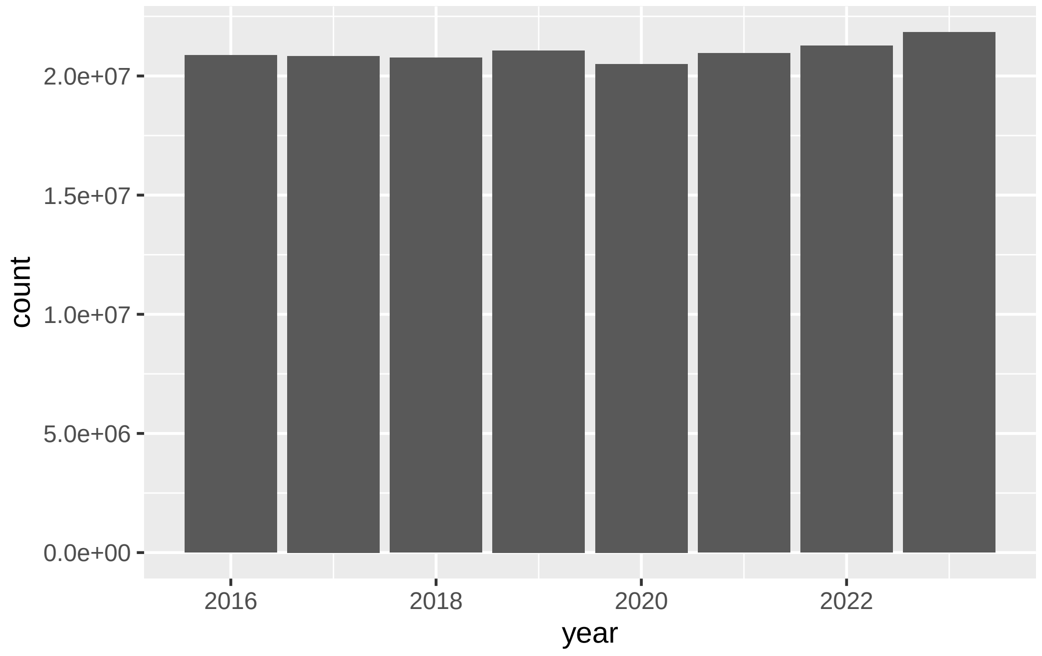
```r
hosps = do.call(rbind, hosps)
head(hosps)
```

```
# A tibble: 6 x 18
   year fac_name fac_city fac_zip fac_operated_this_yr facility_level teach_hosp
  <dbl> <chr>    <chr>    <chr>   <chr>                <chr>          <chr>
1  2016 ALTA BA~ BERKELEY 94705   Yes                  Parent Facili~ NO
2  2016 CHILDRE~ OAKLAND  94609   Yes                  Parent Facili~ NO
3  2016 THUNDER~ OAKLAND  94609   Yes                  Parent Facili~ NO
4  2016 FAIRMON~ SAN LEA~ 94578   Yes                  Consolidated ~ NO
5  2016 ALTA BA~ BERKELEY 94704   Yes                  Consolidated ~ NO
6  2016 HIGHLAN~ OAKLAND  94602   Yes                  Parent Facili~ YES
# i 11 more variables: county <chr>, prin_service_type <chr>,
#   tot_lic_beds <dbl>, tot_lic_bed_days <dbl>, tot_discharges <dbl>,
#   tot_cen_days <dbl>, acute_respiratory_care_lic_beds <dbl>,
#   acute_respiratory_care_lic_bed_days <dbl>,
#   acute_respiratory_care_discharges <dbl>,
#   acute_respiratory_care_intra_transfers <dbl>,
#   acute_respiratory_care_cen_days <dbl>
```

We've finally got all of the data in a single data frame!

To begin to address whether hospital utilization changed in 2020, let's make a bar plot of total census-days:

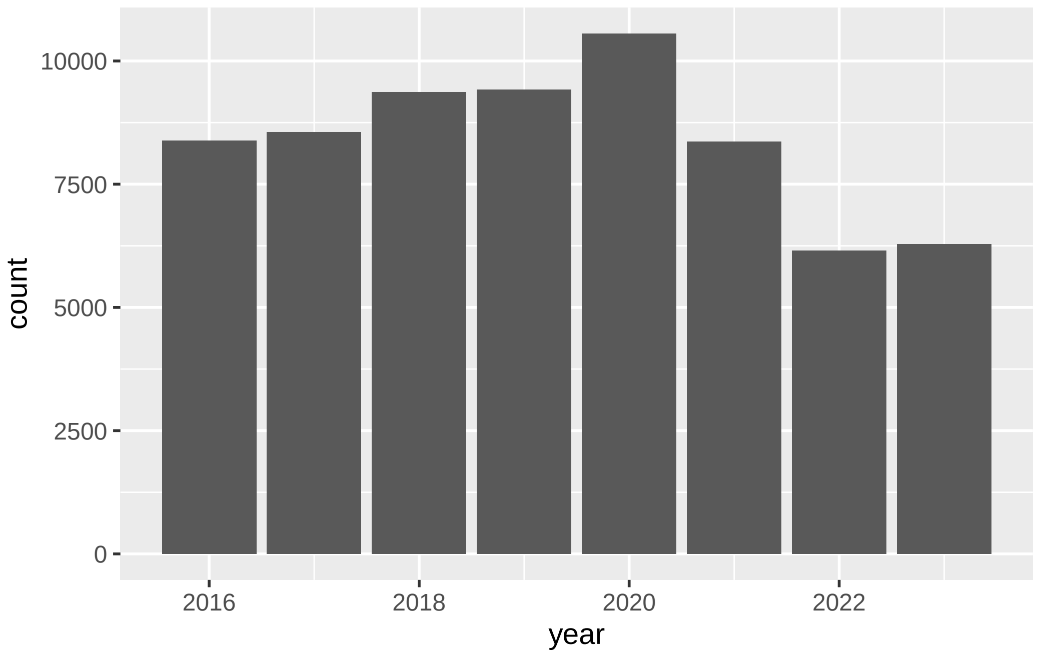
```r
library("ggplot2")

(
    ggplot(hosps) +
    aes(x = year, weight = tot_cen_days) +
    geom_bar()
)
```



According to the plot, total census-days was slightly lower in 2020 than in 2019. This is a bit surprising, but it's possible that California hospitals typically operate close to maximum capacity and were not able to substantially increase the number of beds in 2020 in response to the COVID-19 pandemic. You can use other columns in the data set, such as `tot_lic_beds` or `tot_lic_bed_days`, to check this.

Let's also look at census-days for acute respiratory care:

```r
(
    ggplot(hosps) +
    aes(x = year, weight = acute_respiratory_care_cen_days) +
    geom_bar()
)
```



In this plot, there's a clear uptick in census-days in 2020, and then an interesting decrease to below 2019 levels in the years following. Again, you could use other columns in the data set to investigate this further. We'll end this case study here, having accomplished the difficult task of reading the data and the much easier task of doing a cursory preliminary analysis of the data.

4.5 Exercises

4.5.1 Exercise

1. Write a function `is_leap` that detects leap years. The input to your function should be a numeric year and the output should be a logical value. 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. Test your function on these years, as well as the years 400 and 1997.

   Hint: the modulo operator %% returns the remainder after dividing a number, so for example 4 %% 3 returns 1.

2. Is your `is_leap` function vectorized (see Section 2.1.3)? Explain how you can tell. If it's not vectorized, make a modified version of the function that is.

4.5.2 Exercise

Note

This exercise is meant to challenge you and is quite difficult compared to the previous ones. Don't get disheartened, and if you're able to complete it, excellent work!

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"`).

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 1.3.3. 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
```

```r
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:

```r
x = 0.5 - 0.3
y = 0.3 - 0.1

# FALSE on most machines:
x == y
```

```
[1] FALSE
```

```r
# 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:

```r
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 = terns[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 = terns[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.