T. Mailund, *Beginning Data Science in R 4*

https://doi.org/10.1007/978-1-4842-8155-0_1

1. Introduction to R Programming

Thomas Mailund[1]

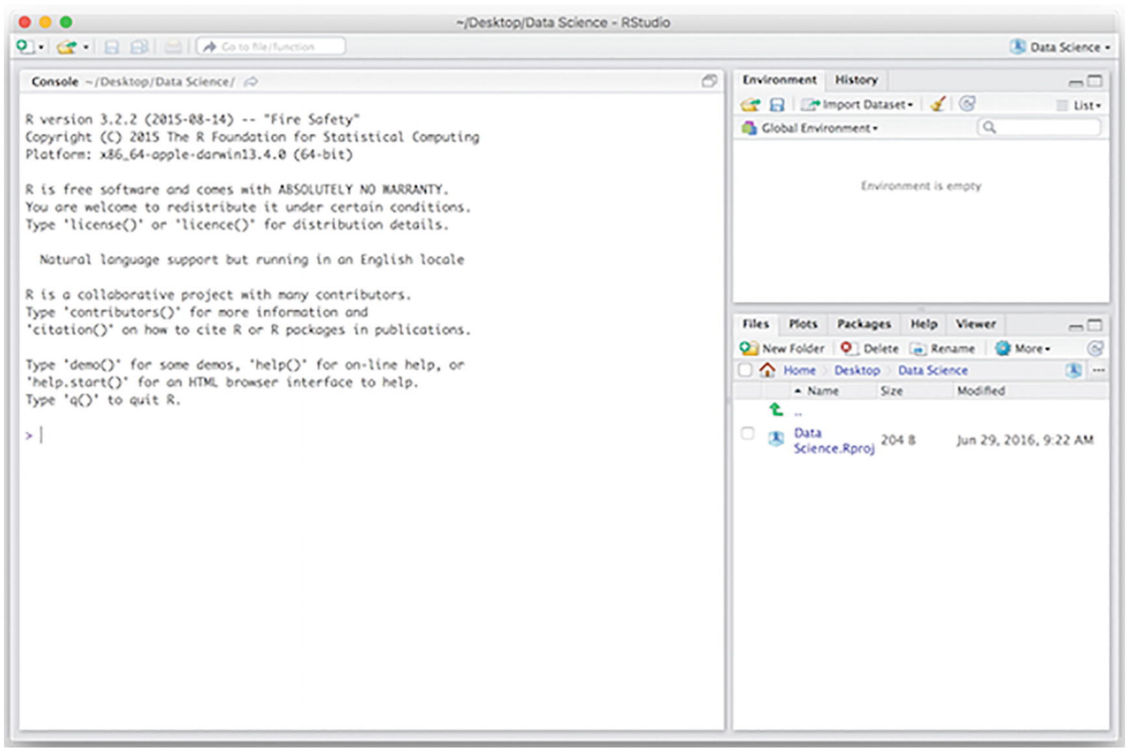
(1) Aarhus, Denmark

We will use R for our data analysis, so we need to know the basics of programming in the R language. R is a full programming language with both functional programming and object-oriented programming features, and learning the complete language is far beyond the scope of this chapter. We return to it later, when we have a little more experience using R. The good news is, though, that to use R for data analysis, we rarely need to do much programming. At least, if you do the right kind of programming, you won't need much.

For manipulating data—how to do this is the topic of the next chapter —you mainly have to string together a couple of operations, such as "group the data by this feature" followed by "calculate the mean value of these features within each group" and then "plot these means." Doing this used to be more complicated to do in R, but a couple of new ideas on how to structure data flow—and some clever implementations of these in packages such as `magrittr` and `dplyr`—have significantly simplified it. We will see some of this at the end of this chapter and more in the next chapter. First, though, we need to get a taste of R.

Basic Interaction with R

Start by downloading RStudio if you haven't done so already. If you open it, you should get a window similar to Figure 1-1. Well, except that you will be in an empty project while the figure shows (on the top right) that this RStudio is opened in a project called "Data Science." You always want to be working on a project. Projects keep track of the state of your analysis by remembering variables and functions you have written and keep track of which files you have opened and such. Go to File and then New Project to create a project. You can create a project from an existing directory, but if this is the first time you are working with R, you probably just want to create an empty project in a new directory, so do that.



```
R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (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.

> |
```

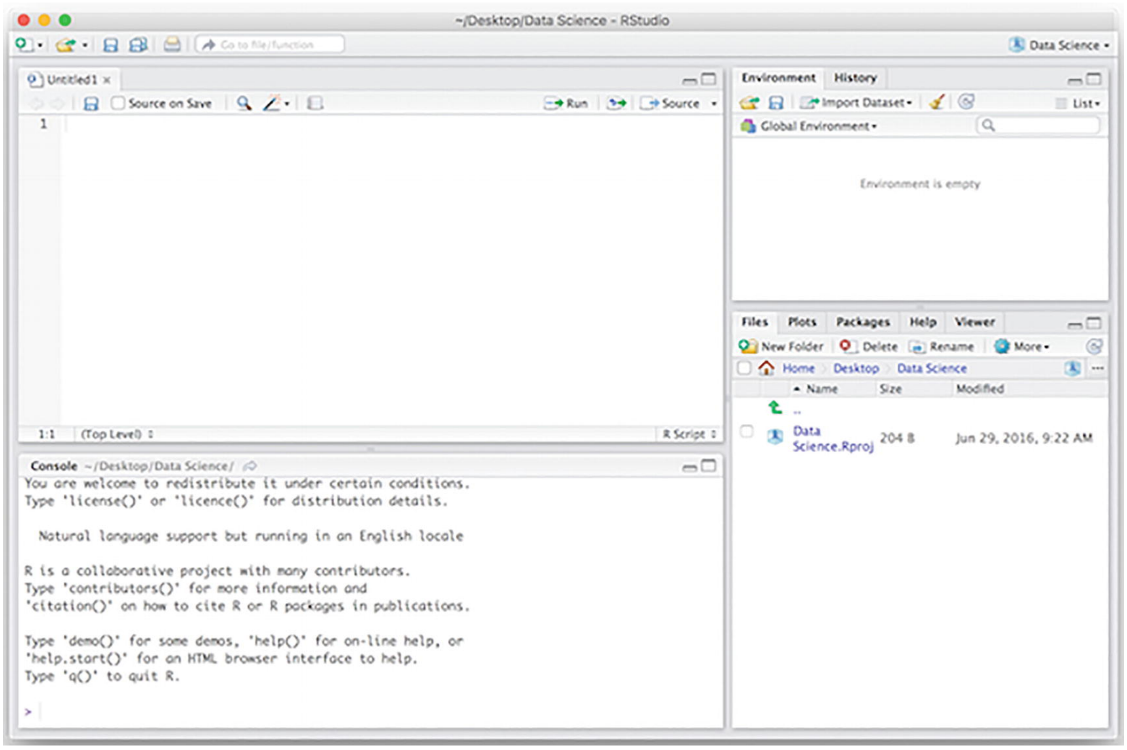
```
..
Data Science.Rproj   204 B   Jun 29, 2016, 9:22 AM
```

Figure 1-1 RStudio

Once you have RStudio opened, you can type R expressions into the console, which is the frame on the left of the RStudio window. When you write an expression there, R will read it, evaluate it, and print the result. When you assign values to variables, and we will see how to do this shortly, they will appear in the Environment frame on the top right. At the bottom right, you have the directory where the project lives, and files you create will go there.

To create a new file, you go to File and then New File.... There you can select several different file types. Those we are interested in are the R Script, R Notebook, and R Markdown types. The former is the file type for pure R code, while the latter two we use for creating reports where documentation text is mixed with R code. For data analysis projects, I would recommend using either Notebook or Markdown files. Writing documentation for what you are doing is helpful when you need to go back to a project several months down the line.

For most of this chapter, you can just write R code in the console, or you can create an R Script file. If you create an R Script file, it will show up on the top left; see Figure 1-2. You can evaluate single expressions using the Run button on the top right of this frame or evaluate the entire file using the Source button. For writing longer expressions, you might want to write them in an R Script file for now. In the next chapter, we will talk about R Markdown, which is the better solution for data science projects.



```
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 1-2 RStudio with a new R Script file open

Using R As a Calculator

You can use the R console as a calculator where you type in an expression you want to calculate, hit "enter," and R gives you the result. You can play around with that a little bit to get familiar with how to write expressions in R—there is some explanation for how to write them in the following—and then moving from using R as a calculator to writing more sophisticated analysis programs is only a matter of degree. A data analysis program is little more than a sequence of calculations, after all.

Simple Expressions

Simple arithmetic expressions are written, as in most other programming languages, in the typical mathematical notation that you are used to:

```
1 + 2
## [1] 3
4 / 2
## [1] 2
(2 + 2) * 3
## [1] 12
```

Here, the lines that start with `##` show the output that R will give you. By convention, and I don't really know why, these two hash symbols are often used to indicate that in R documentation.

It also works pretty much as you are used to, except, perhaps, that you might be used to integers behaving as integers in a division. At least in some programming languages, division between integers is integer division, but in R you can divide integers, and if there is a remainder, you will get a floating-point number back as the result:

```
4 / 3
## [1] 1.333333
```

When you write numbers like 4 and 3, they are always interpreted as floating-point numbers, even if they print as integers, that is, without a decimal point. To explicitly get an integer, you must write 4L and 3L:

```
class(4)
## [1] "numeric"
class(4L)
## [1] "integer"
```

It usually doesn't matter if you have an integer or a floating-point number, and everywhere you see numbers in R, they are likely to be floats.

You will still get a floating-point if you divide two integers, and there is no need to tell R explicitly that you want floating-point division. If you do want integer division, on the other hand, you need a different operator, `%/%`:

```
4 %/% 3
## [1] 1
```

In many languages, `%` is used for getting the remainder of a division, but this doesn't quite work with R where `%` is used for something else (creating new infix operators), so in R the operator for this is `%%`:

```
4 %% 3
```

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

In addition to the basic arithmetic operators—addition, subtraction, multiplication, division, and the modulus operator we just saw—you also have an exponentiation operator for taking powers. For this, you can use either `^` or `**` as infix operators:

```
2 ^ 2
## [1] 4
2 ** 2
## [1] 4
2 ^ 3
## [1] 8
2 ** 3
## [1] 8
```

There are some other data types besides numbers, but we won't go into an exhaustive list here. There are two types you do need to know about early, though, since they are frequently used and since not knowing about how they work can lead to all kinds of grief. Those are strings and "factors."

Strings work as you would expect. You write them in quotes, either double quotes or single quotes, and that is about it:

```
"Hello,"
## [1] "Hello,"
'world!'
## [1] "world!"
```

Strings are not particularly tricky, but I mention them because they look a lot like factors, but factors are not like strings, they just look sufficiently like them to cause some confusion. I will explain the factors a little later in this chapter when we have seen how functions and vectors work.

Assignments

To assign a value to a variable, you use the arrow operators. So to assign the value `2` to the variable `x`, you would write

```
x <- 2
```

and you can test that `x` now holds the value `2` by evaluating `x`:

```
x
## [1] 2
```

and of course, you can now use `x` in expressions:

```
2 * x
## [1] 4
```

You can assign with arrows in both directions, so you could also write

```
2 -> x
```

An assignment won't print anything if you write it into the R terminal, but you can get R to print it by putting the assignment in parentheses:

```
x <- "invisible"
(y <- "visible")
## [1] "visible"
```

Actually, all of the above are vectors of values…

If you were wondering why all the values printed earlier had a `[1]` in front of them, it is because we are usually not working with single values anywhere in R. We are working with vectors of values (and you will hear more about vectors in the next section). The vectors we have seen have length one—they consist of a single value—so there is nothing wrong about thinking about them as individual values. But they are vectors and what we can do with a single number we can do with multiple in the same way.

The `[1]` does not indicate that we are looking at a vector of length one. The `[1]` tells you that the first value after `[1]` is the first value in the vector. With longer vectors, you get the index each time R moves to the next line of output. This output makes it easier to count your way into a particular index.

You will see this if you make a longer vector, for example, we can make one of length 50 using the `:` operator:

```
1:50
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
## [16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
## [31] 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
## [46] 46 47 48 49 50
```

The `:` operator creates a sequence of numbers, starting at the number to the left of the colon and increasing by one until it reaches the number to the right of the colon, or just before if an increment of one would move past the last number:

```
-1:1
```

```
## [1] -1  0  1
0.1:2.9
## [1] 0.1 1.1 2.1
```

If you want other increments than 1, you can use the `seq` function instead:

```
seq(.1, .9, .2)
## [1] 0.1 0.3 0.5 0.7 0.9
```

Here, the first number is where we start, the second where we should stop, as with `:`, but the third number gives us the increment to use.

Because we are practically always working on vectors, there is one caveat I want to warn you about. If you want to know the length of a string, you might—reasonably enough—think you can get that using the `length` function. You would be wrong. That function gives you the length of a vector, so if you give it a single string, it will always return 1:

```
length("qax")
## [1] 1
length("quux")
## [1] 1
length(c("foo", "bar"))
## [1] 2
```

In the last expression, we used the function `c()` to concatenate two vectors of strings. Concatenating `"foo"` and `"bar"`

```
c("foo", "bar")
## [1] "foo" "bar"
```

creates a vector of two strings, and thus the result of calling `length` on that is 2. To get the length of the actual string, you want `nchar` instead:

```
nchar("qax")
## [1] 3
nchar("quux")
## [1] 4
nchar(c("foo", "bar"))
## [1] 3 3
```

If you wanted to concatenate the strings `"foo"` and `"bar"`, to get a vector with the single string `"foobar"`, you need to use `paste`:

```
paste("foo", "bar", sep = "")
## [1] "foobar"
```

The argument `sep = ""` tells `paste` not to put anything between the two strings. By default, it would put a space between them:

```
paste("foo", "bar")
## [1] "foo bar"
```

Indexing Vectors

If you have a vector and want the i'th element of that vector, you can index the vector to get it like this:

```
(v <- 1:5)
## [1] 1 2 3 4 5
v[1]
## [1] 1
v[3]
## [1] 3
```

We have parentheses around the first expression to see the output of the operation. An assignment is usually silent in R, but by putting the expression in parentheses, we make sure that R prints the result, which is the vector of integers from 1 to 5. Notice here that the first element is at index 1. Many programming languages start indexing at zero, but R starts indexing at one. A vector of length n is thus indexed from 1 to n, unlike in zero-indexed languages where the indices go from 0 to $n - 1$.

If you want to extract a subvector, you can also do this with indexing. You just use a vector of the indices you want inside the square brackets. We can use the `:` operator for this or the concatenate function, `c()`:

```
v[1:3]
## [1] 1 2 3
v[c(1,3,5)]
## [1] 1 3 5
```

You can use a vector of boolean values to pick out those values that are "true":

```
v[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
## [1] 1 3 5
```

This indexing is particularly useful when you combine it with expressions. We can, for example, get a vector of boolean values telling us which values of a vector are even numbers and then use that vector to pick them out:

```
v %% 2 == 0
## [1] FALSE  TRUE FALSE  TRUE FALSE
v[v %% 2 == 0]
## [1] 2 4
```

You can get the complement of a vector of indices if you change the sign of them:

```
v[-(1:3)]
```

```
## [1] 4 5
```

It is also possible to give vector indices names, and if you do, you can use those to index into the vector. You can set the names of a vector when constructing it or use the `names` function:

```
v <- c("A" = 1, "B" = 2, "C" = 3)
v
## A B C
## 1 2 3
v["A"]
## A
## 1
names(v) <- c("x", "y", "z")
v
## x y z
## 1 2 3
v["x"]
## x
## 1
```

Names can be handy for making tables where you can look up a value by a key.

Vectorized Expressions

Now, the reason that the expressions we saw earlier worked with vector values instead of single values is that in R, arithmetic expressions all work component-wise on vectors. When you write an expression such as

```
x <- 1:3 ; y <- 4:6
x ** 2 - y
## [1] -3 -1  3
```

you are telling R to take each element in the vector `x`, squaring it, and subtracting element-wise by `y`:

```
(x <- 1:3)
## [1] 1 2 3
x ** 2
## [1] 1 4 9
y <- 6:8
x ** 2 - y
## [1] -5 -3  1
```

This also works if the vectors have different lengths, as they do in the preceding example. The vector `2` is a vector of length 1 containing the number `2`. The way expressions work, when vectors do not have the same length, is you repeat the shorter vector as many times as you need to:

```
(x <- 1:4)
## [1] 1 2 3 4
(y <- 1:2)
## [1] 1 2
x - y
## [1] 0 0 2 2
```

If the length of the longer vector is not a multiple of the length of the shorter, you get a warning. The expression still repeats the shorter vector a number of times, just not an integer number of times:

```
(x <- 1:4)
## [1] 1 2 3 4
(y <- 1:3)
## [1] 1 2 3
x - y
## Warning in x - y: longer object length is not a
## multiple of shorter object length
## [1] 0 0 0 3
```

Here, `y` is used once against the `1:3` part of `x`, and the first element of `y` is then used for the `4` in `x`.

Comments

You probably don't want to write comments when you are just interacting with the R terminal, but in your code, you do. Comments let you describe what your code is intended to do, and how it is achieving it, so you don't have to work that out again when you return to it at a later point, having forgotten all the great thoughts you thought when you wrote it.

R interprets as comments everything that follows the `#` character. From a `#` to the end of the line, the R parser skips the text:

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

If you write your analysis code in R Markdown documents, which we will cover in the next chapter, you won't have much need for comments. In those kinds of files, you mix text and R code differently. But if you develop R code, you will likely need it, and now you know how to write comments.

Functions

You have already seen the use of functions, although you probably didn't think much about it when we saw expressions such as

```
length("qax")
```

You didn't think about it because there wasn't anything surprising about it. We just use the usual mathematical notation for functions: $f(x)$. If you want to call a function, you simply use this notation and give the function its parameters in parentheses.

In R, you can also use the names of the parameters when calling a function, in addition to the positions; we saw an example with `sep = ""` when we used `paste` to concatenate two strings.

If you have a function $f(x, y)$ of two parameters, x and y, calling $f(5, 10)$ means calling f with parameter x set to 5 and parameter y set to 10. In R, you can specify this explicitly, and these two function calls are equivalent:

```
f(5, 10)
f(x = 5, y = 10)
```

(Don't try to run this code; we haven't defined the function `f`, so calling it will fail. But if we had a function `f`, then the two calls would be equivalent.)

If you specify the names of the parameters, the order doesn't matter anymore, so another equivalent function call would be

```
f(y = 10, x = 5)
```

You can combine the two ways of passing parameters to functions as long as you put all the positional parameters before the named ones:

```
f(5, y = 10)
```

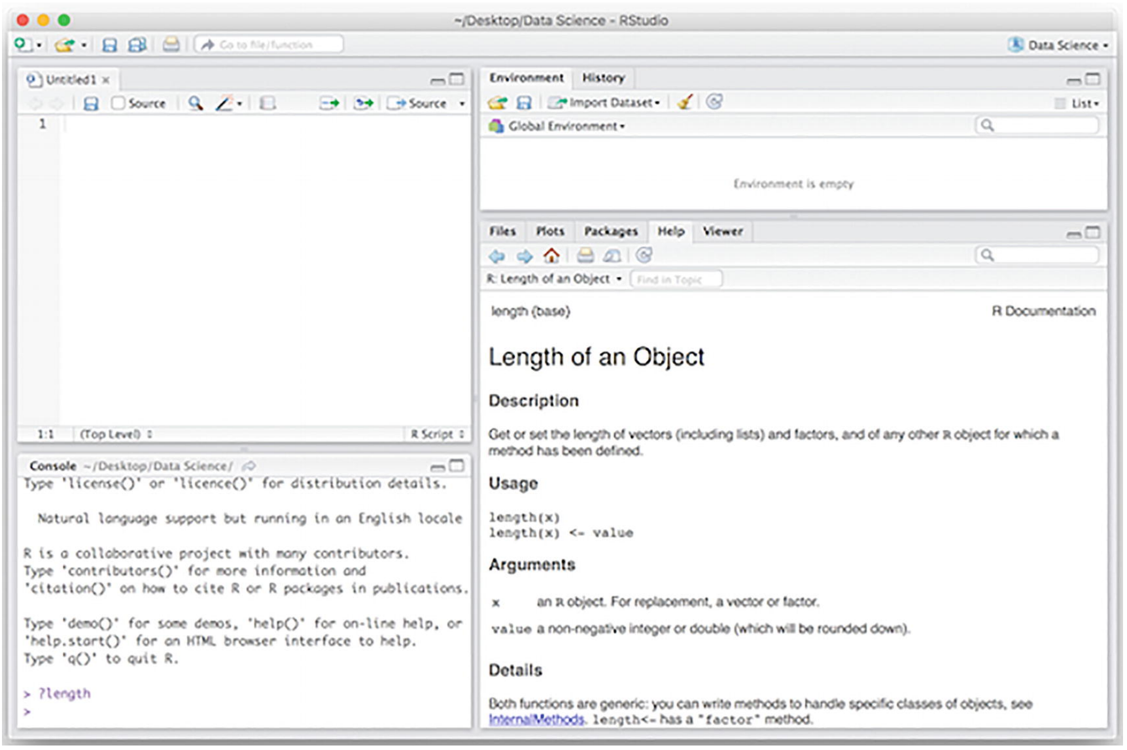
Except for maybe making the code slightly more readable—it is usually easier to remember what parameters do than which order they come in—there is not much need for this in itself. Where it becomes useful is when combined with default parameters.

A lot of functions in R take many parameters. More than you really can remember the use for and certainly the order of. They are a lot like programs that take a lot of options but where you usually just use the defaults unless you need to tweak something. These functions take a lot of parameters, but most of them have useful default values, and you typically do not have to specify the values to set them to. When you do need it, though, you can specify it with a named parameter.

Getting Documentation for Functions

Since it can be hard to remember the details of what a function does, and especially what all the parameters to a function do, you often have to look up the documentation for functions. Luckily, this is very easy to do in R and RStudio. Whenever you want to know what a function does, you can just ask R, and it will tell you (assuming that the author of the function has written the documentation).

Take the function `length` from the example we saw earlier. If you want to know what the function does, just write `?length` in the R terminal. If you do this in RStudio, it will show you the documentation in the frame on the right; see Figure **1-3**.



```
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.

> ?length
>
```

length (base) R Documentation

Length of an Object

Description

Get or set the length of vectors (including lists) and factors, and of any other R object for which a method has been defined.

Usage

```
length(x)
length(x) <- value
```

Arguments

`x`	an R object. For replacement, a vector or factor.
`value`	a non-negative integer or double (which will be rounded down).

Details

Both functions are generic: you can write methods to handle specific classes of objects, see InternalMethods. `length<-` has a `"factor"` method.

Figure 1-3 RStudio's help frame

Try looking up the documentation for a few functions, for example, the `nchar` function we also saw earlier.

All infix operators, like `+` or `%%`, are also functions in R, and you can read the documentation for them as well. But you cannot write `?+` in the R terminal and get the information. The R parser doesn't know how to deal with that. If you want help on an infix operator, you need to quote it, and you do that using back quotes. So to read the documentation for `+`, you would need to write

```
?`+`
```

You probably do not need help to figure out what addition does, but people can write new infix operators, so this is useful to know when you need help with those.

Writing Your Own Functions

You can easily write your own functions. You use `function` expressions to define a function and an assignment to give a function a name. For example, to write a function that computes the square of a number, or a vector number, you can write

```
square <- function(x) x**2
square(2)
## [1] 4
square(1:4)
## [1]  1  4  9 16
```

The "`function(x) x**2`" expression defines the function, and anywhere you would need a function, you can write the function explicitly like this. Assigning the function to a name lets you use the name to refer to the function, just like assigning any other value, like a number or a string to a name, will let you use the name for the value.

Functions you write yourself work just like any function already part of R or part of an R package, with one exception, though: you will not have documentation for your functions unless you write it, and that is beyond the scope of this chapter (but covered in the chapter on building packages).

The `square` function just does a simple arithmetic operation on its input. Sometimes, you want the function to do more than a single thing. If you want the function to do several operations on its input, you need several statements for the function. In that case, you need to give it a "body" of several statements, and such a body has to go in curly brackets:

```
square_and_subtract <- function(x, y) {
    squared <- x ** 2
    squared - y
}
square_and_subtract(1:5, rev(1:5))
## [1] -4 0 6 14 24
```

(Check the documentation for `rev` to see what is going on here. Make sure you understand what this example is doing.)

In this simple example, we didn't really need several statements. We could just have written the function as

```
square_and_subtract <- function(x, y) x ** 2 - y
```

As long as there is only a single expression in the function, we don't need the curly brackets. For more complex functions, you will need it, though.

The result of a function—what it returns as its value when you call it—is the last statement or expression (there actually isn't any difference between statements and expressions in R; they are the same thing). You can make the return value explicit, though, using the `return()` expression:

```
square_and_subtract <- function(x, y) return(x ** 2 - y)
```

Explicit returning is usually only used when you want to return a value before the end of the function. To see examples of this, we need control structures, so we will have to wait a little bit to see an example. It isn't used as much as in many other programming languages.

One crucial point here, though, if you are used to programming in other languages: The `return()` expression needs to include the parentheses. In most programming languages, you could just write

```
square_and_subtract <- function(x, y) return x ** 2 - y
```

Such an expression doesn't work for R. Try it, and you will get an error.

Summarizing and Vector Functions

As we have already seen, when we write arithmetic expressions such as `x**2 - y`, we have an expression that will work for both single numbers for `x` and `y`, but also element-wise for vectors `x` and `y`. If you write functions where the body consists of such expressions, the function will work element-wise as well. The `square` and `square_and_subtract` functions we wrote earlier work like that.

Now all functions work like this, however. While we often can treat data one element at a time, we also often need to extract some summary of a collection of data, and functions handle this as well.

Take, for example, the function `sum` which adds together all the values in a vector you give it as an argument (check `?sum` now to see the documentation):

```
sum(1:4)
## [1] 10
```

This function summarizes its input into a single value. There are many similar functions, and, naturally, these cannot be used element-wise on vectors; rather, they reduce an entire vector into some smaller summary statistics, here the sum of all elements.

Whether a function works on vector expressions or not depends on how it is defined. While there are exceptions, most functions in R either work on vectors or summarize vectors like `sum`. When you write your own functions, whether the function works element-wise on vectors or not depends on what you put in the body of the function. If you write a function that just does arithmetic on the input, like `square`, it will work in vectorized expressions. If you write a function that does some summary of the data, it will not. For example, if we write a function to compute the average of its input like this:

```
average <- function(x) {
    n <- length(x)
    sum(x) / n
}
average(1:5)
## [1] 3
```

This function will not give you values element-wise. Pretty obviously. It gets a little more complicated when the function you write contains control structures, which we will get to in the next section. In any case, this would be a nicer implementation since it only involves one expression:

```
average <- function(x) sum(x) / length(x)
```

Oh, and by the way, don't use this `average` function to compute the mean value of a vector. R already has a function for that, `mean`, that deals much better with special cases like missing data and vectors of length zero. Check out `?mean`.

Just because you are summarizing doesn't mean that you have to return a single value. In this function, we return both the mean and the standard deviation of the values in a vector:

```
mean_and_sd <- function(x) c(mean = mean(x), sd = sd(x))
mean_and_sd(1:10)
##     mean       sd
##  5.50000  3.02765
```

We use the functions `mean` and `sd` to compute the two summary statistics, and then we combine them into a vector (with named elements) that contains the two summaries. This isn't a vectorized function, because we do not process the values in the input element-wise. It doesn't compute a single summary, but returns something (ever so slightly) more complex. Complicated functions often return data more complex than vectors or single values, and we shall see examples in later chapters. If you can avoid it, though, do so. Simple functions, with simple input and output, are easier to use, and when we write functions, we want to make things as simple for us as we can. With this `mean_and_sd` function, we do not gain anything that we do not already have with the `mean` and `sd` function, and combining both operations in a single function only complicates things needlessly.

The rough classification of functions into the vectorized, which operate element-wise on data, and the summarizing functions, is only a classification of how we can use them. If you compute a value for each element in one or more vectors, you have the former, and if you summarize all the data in one or more vectors, you have the latter. The implementation of a function can easily combine both.

Imagine, for example, that we wish to normalize data by subtracting the mean from each element and then dividing by the standard deviation. We could implement it like this:

```
normalise <- function(x) (x - mean(x)) / sd(x)
normalise(1:10)
##  [1]  -1.4863011  -1.1560120 -0.8257228 -0.4954337
##  [5]  -0.1651446   0.1651446  0.4954337  0.8257228
##  [9]   1.1560120   1.4863011
```

We compute a value for each element in the input, so we have a vectorized function, but in the implementation, we use two summarizing functions, `mean` and `sd`. The expression `(x - mean(x)) / sd(x)` is a vector expression because `mean(x)` and `sd(x)` become vectors of length one, and we can use those in the expression involving `x` to get a value for each element.

A Quick Look at Control Flow

While you get very far just using expressions, for many computations, you need more complex programming. Not that it is particularly complex, but you do need to be able to select a choice of what to do based on data—selection or `if` statements—and ways of iterating through data, looping or `for` statements.

If statements work like this:

```
if (<boolean expression>) <expression>
```

If the boolean expression evaluates to true, the expression is evaluated; if not, it will not:

```
# this won't do anything
if (2 > 3) "false"
# this will
if (3 > 2) "true"
## [1] "true"
```

For expressions like these, where we do not alter the program state by evaluating the expression, there isn't much of an effect in evaluating the `if` expression. If we, for example, are assigning to a variable, there will be an effect:

```
x <- "foo"
if (2 > 3) x <- "bar"
x
```

```
## [1] "foo"
if (3 > 2) x <- "baz"
x
## [1] "baz"
```

If you want to have effects for both true and false expressions, you have this:

```
if (<boolean expression>) <true expression> else <false
expression>
if (2 > 3) "bar" else "baz"
## [1] "baz"
```

If you want newlines in `if` statements, whether you have an `else` part or not, you should use curly brackets.

You don't always have to. If you have a single expression in the `if` part, you can leave them out:

```
if (3 > 2)
    x <- "bar"
x
## [1] "bar"
```

or if you have a single statement in the `else` part, you can leave out the brackets:

```
if (2 > 3) {
    x <- "bar"
} else
    x <- "qux"
x
## [1] "qux"
```

but we did need the brackets in the preceding `if` part for R to recognize that an `else` bit was following. Without it, we would get an error:

```
if (2 > 3)
    x <- "bar"
else
    x <- "qux"
## Error: <text>:3:1: unexpected 'else'
## 2:     x <- "bar"
## 3: else
##    ^
```

If you always use brackets, you don't have to worry about when you strictly need them or when you do not, and a part can have multiple statements without you having to worry about it. If you put a newline in an `if` or `if-else` expression, I recommend that you always use brackets as well.

An `if` statement works like an expression:

```
if (2 > 3) "bar" else "baz"
## [1] "baz"
```

This evaluates to the result of the expression in the “if” or the “else” part, depending on the truth value of the condition:

```
x <- if (2 > 3) "bar" else "baz"
x
## [1] "baz"
```

It works just as well with braces:

```
x <- if (2 > 3) { "bar" } else { "baz" }
x
## [1] "baz"
```

but when the entire statement is on a single line, and the two parts are both a single expression, I usually do not bother with that.

You cannot use it for vectorized expressions, though, since the boolean expression, if you give it a vector, will evaluate the first element in the vector:

```
x <- 1:5
if (x > 3) "bar" else "baz"
## Warning in if (x > 3) "bar" else "baz": the
## condition has length > 1 and only the first
## element will be used
## [1] "baz"
```

If you want a vectorized version of `if` statements, you can instead use the `ifelse()` function:

```
x <- 1:5
ifelse(x > 3, "bar", "baz")
## [1] "baz" "baz" "baz" "bar" "bar"
```

(read the `?ifelse` documentation to get the details of this function).

This, of course, also has consequences for writing functions that use `if` statements. If your function contains a body that isn’t vectorized, your function won’t be either. So, if you have an `if` statement that depends on your input—and if it doesn’t depend on the input, it is rather useless—then that input shouldn’t be a vector:

```
maybe_square <- function(x) {
    if (x %% 2 == 0) x ** 2 else x
}
maybe_square(1:5)
## Warning in if (x%%2 == 0) x^2 else x: the
## condition has length > 1 and only the first
## element will be used
## [1] 1 2 3 4 5
```

This function was supposed to square even numbers, and it will if we give it a single number, but we gave it a vector. Since the first value in this vector, the only one that the `if` statement looked at, was 1, it decided that `x %% 2 == 0` was false—it is if `x[1]` is 1—and then none of the values were squared. Clearly not what we wanted, and the warning was warranted.

If you want a vectorized function, you need to use `ifelse()`:

```
maybe_square <- function(x) {
    ifelse(x %% 2 == 0, x ** 2, x)
}
maybe_square(1:5)
## [1] 1 4 3 16 5
```

or you can use the `Vectorize()` function to translate a function that isn't vectorized into one that is:

```
maybe_square <- function(x) {
    if (x %% 2 == 0) x ** 2 else x
}
maybe_square <- Vectorize(maybe_square)
maybe_square(1:5)
## [1] 1 4 3 16 5
```

The `Vectorize` function is what is known as a "functor"—a function that takes a function as input and returns a new function. It is beyond the scope of this chapter to cover how we can manipulate functions like other data, but it is a very powerful feature of R that we return to in later chapters.

For now, it suffices to know that `Vectorize` will take your function that can only take single values as input and then create a function that handles an entire vector by calling your function with each element. You only see one element at a time, and `Vectorize`'s function makes sure that you can handle an entire vector, one element at a time.

To loop over elements in a vector, you use `for` statements:

```
x <- 1:5
total <- 0
for (element in x) total <- total + element
total
## [1] 15
```

As with `if` statements, if you want the body to contain more than one expression, you need to put it in curly brackets.

The `for` statement runs through the elements of a vector. If you want the indices instead, you can use the `seq_along()` function, which given a vector as input returns a vector of indices:

```
x <- 1:5
total <- 0
for (index in seq_along(x)) {
    element <- x[index]
    total <- total + element
}
total
## [1] 15
```

There are also `while` statements for looping. These repeat as long as an expression is true:

```
x <- 1:5
total <- 0
index <- 1
while (index <= length(x)) {
    element <- x[index]
    index <- index + 1
    total <- total + element
}
total
## [1] 15
```

If you are used to zero-indexed vectors, pay attention to the `index <= length(x)` here. You would normally write `index < length(x)` in zero-indexed languages. Here, that would miss the last element.

There is also a `repeat` statement that loops until you explicitly exit using the `break` statement:

```
x <- 1:5
total <- 0
index <- 1
repeat {
    element <- x[index]
    total <- total + element
    index <- index + 1
    if (index > length(x)) break
}
total
## [1] 15
```

There is also a `next` statement that makes the loop jump to the next iteration.

Now that I have told you about loops, I feel I should also say that they generally are not used as much in R as in many other programming languages. Many actively discourage using loops, and they have a reputation for leading to slow code. The latter is not justified in itself, but it is easier to write slow code using loops than the alternatives. Instead, you use functions to take over the looping functionality. There is usually a function for doing whatever you want to accomplish using a loop, and when there is not, you can generally get what you want by combining the three functions `Map`, `Filter`, and `Reduce`.

But that is beyond the scope of this chapter; we return to it later in the book.

Factors

Now let us return to data types and the factors I hinted at a while ago. Factors are mostly just vectors but of categorical values. That just means that the elements of a factor should be considered as categories or classes and not as numbers or strings. For example, categories such as “small,” “medium,” and “large” could be encoded as numbers, but there aren’t any natural numbers to assign to them. We could encode soft drink sizes like 1, 2, and 3 for “small,” “medium,” and “large.” By doing this, we are implicitly saying that the difference between “small” and “medium” is half of the difference between “small” and “large” which may not be the case. Data with sizes “small,” “medium,” and “large” should be encoded as categorical data, not numbers, and in R that means encoding them as factors.

A factor is usually constructed by giving the `factor()` function a list of strings. The function translates these into the different categories, and the factor becomes a vector of the categories:

```
f <- factor(c("small", "small", "medium",
              "large", "small", "large"))
f
## [1] small  small  medium large   small  large
## Levels: large medium small
```

The categories are called “levels”:

```
levels(f)
## [1] "large"   "medium" "small"
```

By default, these are ordered alphabetical, which in this example gives us the order “large,” “medium,” “small.” You can change this order by specifying the levels when you create the factor:

```
ff <- factor(c("small", "small", "medium",
               "large", "small", "large"),
             levels = c("small", "medium", "large"))
ff
## [1] small  small  medium large  small  large
## Levels: small medium large
```

Changing the order of the levels like this changes how many functions handle the factor. Mostly it affects the order that summary statistics or plotting functions present results in.

```
summary(f)
##  large medium  small
##      2      1      3
summary(ff)
##  small medium  large
##      3      1      2
```

The `summary` function, when used on factors, just counts how many of each kind we see, and here we have three “small,” one “medium,” and two “large.” The only thing the order of the levels does is determine in which order `summary` prints the categories.

The order in which the levels are given shouldn’t be thought of as “ordering” the categories, though. It is just used for displaying results; there is not an order semantics given to the levels unless you explicitly specify this.

Some categorical data has a natural order, like “small,” “medium,” and “large.” Other categories are not naturally ordered. There is no natural way of ordering “red,” “green,” and “blue.” When we print data, it will always come out ordered since text always comes out ordered. When we plot data, it is usually also ordered. But in many mathematical models, we would treat ordered categorical data different from unordered categorical data, so the distinction is sometimes important.

By default, factors do not treat the levels as ordered, so they assume that categorical data is like “red,” “green,” and “blue,” rather than ordered like “small,” “medium,” and “large.” If you want to specify that the levels are ordered, you can do that using the `ordered` argument to the `factor()` function:

```
of <- factor(c("small", "small", "medium",
               "large", "small", "large"),
             levels = c("small", "medium", "large"),
             ordered = TRUE)
of
## [1] small   small   medium large   small   large
## Levels: small < medium < large
```

You can also use the `ordered()` function:

```
ordered(ff)
## [1] small  small  medium large  small  large
## Levels: small < medium < large
ordered(f, levels = c("small", "medium", "large"))
## [1] small  small  medium large  small  large
## Levels: small < medium < large
```

In many ways, you can work with a combination of strings and factors. For example, you can check if a factor value is from a certain level by comparing it with the string of that label:

```
f
```

```
## [1] small  small  medium large  small  large
## Levels: large medium small
f == "small"
## [1]  TRUE  TRUE FALSE FALSE  TRUE FALSE
```

Here, we test each of the elements in the factor f against the string “small,” and we get `TRUE` for those that have the level `small`. However, factors are not strings, and in some places they behave fundamentally different. The fact that they so often look like strings makes this extra tricky, when something that looks perfectly innocent can hide a fatal error.

The case where I have seen this the most is when R users try to use factors to index into vectors. While this is a little more advanced than most of what we see in this chapter, I want to show it early so you are aware of the dangers.

When we create a vector, we can give the indices names. We can do this in the same expression as we create the vector:

```
v <- c(a = 1, b = 2, c = 3, d = 4)
v
## a b c d
## 1 2 3 4
```

or we can add the names later:

```
v <- 1:4
names(v) <- letters[1:4]
v
## a b c d
## 1 2 3 4
```

(the `letters` vector contains all the lowercase letters, so `letters[1:4]` are `a`, `b`, `c`, and `d`).

If we have named the elements in the vector, we can use them to index, just as we can use numbers. If we want indices 2 and 3, we can index with `2:3`, but we could also index with `c("b", "c")`:

```
v[2:3]
## b c
## 2 3
v[c("b", "c")]
```

```
## b c
## 2 3
```

The indexing does not have to be in the same order as the elements are in vector, so we could, for example, extract indices 3 and 2, in that order with

```
v[c(3, 2)]
## c b
## 3 2
```

or using their names

```
v[c("c", "b")]
## c b
## 3 2
```

and if we repeat an index, we get the corresponding value more than once:

```
v[c("c", "b", "c")]
## c b c
## 3 2 3
```

Here, we are using a vector of strings to index, but what would happen if we used a factor?

A factor is not stored as strings, even though we create it from a vector of strings. It is stored as a vector of integers where the integers are indices into the levels. This representation can bite you if you try to use a factor to index into a vector.

Read the following code carefully. We have the vector `v` that can be indexed with the letters `A`, `B`, `C`, and `D` (`LETTERS` is a vector that contains the uppercase letters). We create a factor, `ff`, that consists of these four letters in that order. When we index with it, we get what we would expect. Since `ff` is the letters `A` to `D`, we pick out the values from `v` with those labels and in that order:

```
v <- 1:4
names(v) <- LETTERS[1:4]
v
## A B C D
## 1 2 3 4
(ff <- factor(LETTERS[1:4]))
## [1] A B C D
## Levels: A B C D
v[ff]
## A B C D
## 1 2 3 4
```

We are lucky to get the expected result, and it is only luck though, because this expression is not indexing using the names we might expect it to use. Read the following even more carefully!

```
(ff <- factor(LETTERS[1:4], levels = rev(LETTERS[1:4])))
## [1] A B C D
## Levels: D C B A
v[ff]
## D C B A
## 4 3 2 1
```

This time, `ff` is still a vector with the categories `A` to `D` in that order, but we have specified that the levels are `D`, `C`, `B`, and `A`, in that order. So the numerical values that the categories are stored as are actually these:

```
as.numeric(ff)
## [1] 4 3 2 1
```

What we get when we use it to index into `v` are those numerical indices—so we get the values pulled out of `v` in the reversed order from what we would expect if we didn't know this (which you now know).

The easiest way to deal with a factor as if it contained strings is to translate it into a vector of strings. You can use such a vector to index:

```
as.vector(ff)
## [1] "A" "B" "C" "D"
v[as.vector(ff)]
## A B C D
## 1 2 3 4
```

If you ever find yourself using a factor to index something—or in any other way treat a factor as if it was a vector of strings—you should stop and make sure that you explicitly convert it into a vector of strings. Treating a factor as if it was a vector of strings—when, in fact, it is a vector of integers—only leads to tears and suffering in the long run.

Data Frames

The vectors we have seen, whatever their type, are just sequences of data. There is no structure to them except for the sequence order, which may or may not be relevant for how to interpret the data. That is not how data we want to analyze look like. What we usually have is several related variables from some collection of observations. For each observed data point, you have a value for each of these variables (or missing data indications if some variables were not observed). Essentially, what you have is a table with a row per observation and a column per variable. The data type for such tables in R is the `data.frame`.

A data frame is a collection of vectors, where all must be of the same length, and you treat it as a two-dimensional table. We usually think of data frames as having each row correspond to some observation and each column correspond to some property of the observations. Treating data frames that way makes them extremely useful for statistical modelling and fitting.

You can create a data frame explicitly using the `data.frame` function:

```
df <- data.frame(a = 1:4, b = letters[1:4])
df
##   a b
## 1 1 a
## 2 2 b
## 3 3 c
## 4 4 d
```

but usually you will read in the data frame from files.

To get to the individual elements in a data frame, you must index it. Since it is a two-dimensional data structure, you should give it two indices:

```
df[1,1]
## [1] 1
```

You can, however, leave one of these empty, in which case you get an entire column or an entire row:

```
df[1,]
##   a b
## 1 1 a
df[,1]
## [1] 1 2 3 4
```

If the rows or columns are named, you can also use the names to index. This is mostly used for column names since it is the columns that correspond to the observed variables in a data sets. There are two ways to get to a column, but explicitly indexing

```
df[,"a"]
## [1] 1 2 3 4
```

or using the `$column_name` notation that does the same thing but lets you get at a column without having to use the `[]` operation and quote the name of a column:

```
df$b
## [1] "a" "b" "c" "d"
```

Before R version 4, a data frame would consider a character vector as a factor and implicitly convert it. It saves a little space, but was a source of errors as the one I described in the section on factors, so with R4 the default is now to keep string vectors as string vectors. If `df$b` was a factor when you run the preceding code, you are using an older version of R, and I suggest you update it.

Turning string vectors into factors, or keeping them as they are, is just the default behavior, though. You can control it with the `stringsAsFactors` parameter. If you set this to `TRUE`, you will get the old behavior that turns strings into factors:

```
data.frame(a = 1:4, b = letters[1:4],
           stringsAsFactors = TRUE)
##   a b
## 1 1 a
## 2 2 b
## 3 3 c
## 4 4 d
```

If you used `stringsAsFactors = FALSE`, you would get the now default behavior of keeping string vectors as strings.

You can combine two data frames row-wise or column-wise by using the `rbind` and `cbind` functions:

```
df2 <- data.frame(a = 5:7, b = letters[5:7])
rbind(df, df2)
##   a b
## 1 1 a
## 2 2 b
## 3 3 c
## 4 4 d
## 5 5 e
## 6 6 f
## 7 7 g
df3 <- data.frame(c = 5:8, d = letters[5:8])
cbind(df, df3)
##   a b c d
## 1 1 a 5 e
## 2 2 b 6 f
## 3 3 c 7 g
## 4 4 d 8 h
```

These data frames are built into R, but there are various alternatives implemented in packages. They differ from the built-in data frames by being optimized for certain use patterns or just based on programmer taste.

The most popular variant, which you are practically guaranteed to run into sooner rather than later, is the so-called “tibble.” You can get access to it by loading the package `tibble`:

```
library(tibble)
```

or by loading the large collection of packages known as the “tidyverse”:

```
library(tidyverse)
```

The tidyverse is a large framework for working with data in a structured way, implemented in numerous packages, but you can load all the common ones in a single instructing by loading `tidyverse`.

If these two commands did not work when you tried them, it is because you haven’t installed them yet. We return to working with packages shortly, but for now, you can just do

```
install.packages("tidyverse")
```

After that, both of the preceding `library(...)` commands should work.

Then, to create a tibble instead of a built-in data frame, you can use

```
tibble(a = 1:4, b = letters[1:4])
## # A tibble: 4 × 2
##       a b
##   <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
## 4     4 d
```

As you can see, the syntax is much the same as when you create a data frame with the `data.frame` function, and the result is similar as well. Generally, you can use tibbles as drop-in replacements for data frames. The operations you can do on data frames you can also do on tibbles.

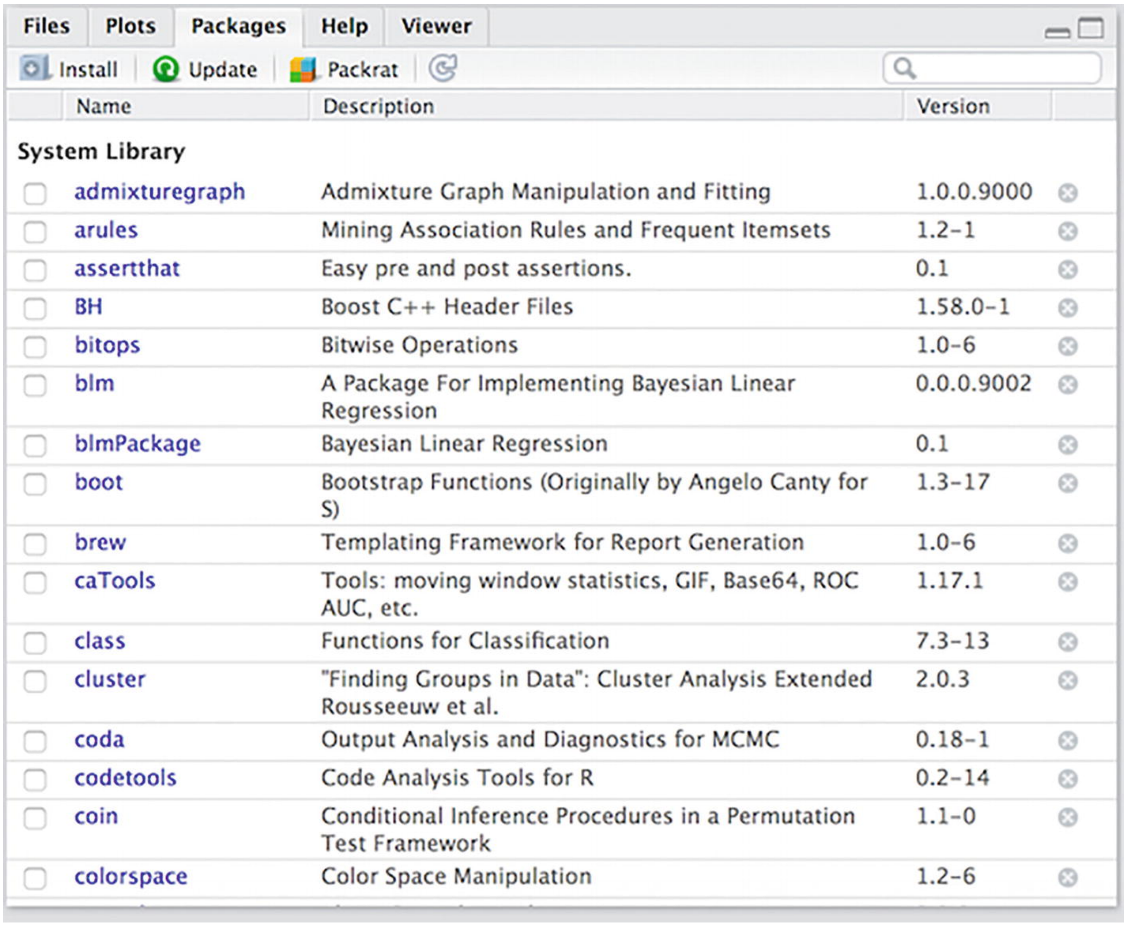
In day-to-day programming, there is not a big difference between data frames and tibbles, but the latter prints a little better by giving a nicer summary of large data collections, and as already mentioned, they are heavily used by the tidyverse framework, so you are more likely to use them than the classical data frames if you start using the packages there, which I strongly suggest that you do.

For more sophisticated manipulation of data frames, you really should use the `dplyr` package, also part of the tidyverse, but we will return to this in Chapter **3**.

Using R Packages

Out of the box, R has a lot of functionality, but where the real power comes in is through its package mechanism and the large collection of packages available for download and use.

When you install RStudio, you also install a set of default packages. You can see which packages are installed by clicking the Packages tab in the lower-right frame; see Figure **1-4**.



System Library

Name	Description	Version
admixturegraph	Admixture Graph Manipulation and Fitting	1.0.0.9000
arules	Mining Association Rules and Frequent Itemsets	1.2-1
assertthat	Easy pre and post assertions.	0.1
BH	Boost C++ Header Files	1.58.0-1
bitops	Bitwise Operations	1.0-6
blm	A Package For Implementing Bayesian Linear Regression	0.0.0.9002
blmPackage	Bayesian Linear Regression	0.1
boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-17
brew	Templating Framework for Report Generation	1.0-6
caTools	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.17.1
class	Functions for Classification	7.3-13
cluster	"Finding Groups in Data": Cluster Analysis Extended Rousseeuw et al.	2.0.3
coda	Output Analysis and Diagnostics for MCMC	0.18-1
codetools	Code Analysis Tools for R	0.2-14
coin	Conditional Inference Procedures in a Permutation Test Framework	1.1-0
colorspace	Color Space Manipulation	1.2-6

Figure 1-4 RStudio packages

From here, you can update packages—new versions of essential packages are released regularly—and you can install new packages. You might have already done this when we talked about tibbles, but try another one. Try installing the package `magrittr`. We are going to use it shortly.

You can also install packages from the R console. Just write

```
install.packages("magrittr")
```

Once you have installed a package, you have access to the functionality in it. You can get function `f` in `package` by writing `package::f()`, or you can load all functions from a package into your global namespace to have access to them without using the `package::` prefix.

Loading the functionality from the `magrittr` package is done like this:

```
library(magrittr)
```

Dealing with Missing Values

Most data sets have missing values—parameters that weren't observed or that were incorrectly recorded and had to be masked out. How you deal with missing data in an analysis depends on the data and the analysis, but it must be addressed, even if all you do is remove all observations with missing data.

Missing data is represented in R by the special value `NA` (not available). Values of any type can be missing and represented as `NA`, and importantly R knows that `NA` means missing values and treats `NA`s accordingly. You should always represent missing data as `NA` instead of some particular number (like -1 or 999 or whatever). R knows how to work with `NA` but has no way of knowing that `-1` means anything besides minus one.

Operations that involve `NA` are themselves `NA`—you cannot operate on missing data and get anything but more missing values in return. This also means that if you compare two `NA`s, you get `NA`. Because `NA` is missing information, it is not even equal to itself:

```
NA + 5
## [1] NA
NA == NA
## [1] NA
NA != NA
## [1] NA
```

If you want to check if a value is missing, you must use the function `is.na`:

```
is.na(NA)
## [1] TRUE
is.na(4)
## [1] FALSE
```

Functions such as `sum()` will by default return `NA` if its input contains `NA`:

```
v <- c(1,NA,2)
sum(v)
## [1] NA
```

If you want just to ignore the `NA` values, there is often a parameter for specifying this:

```
sum(v, na.rm = TRUE)
## [1] 3
```

Data Pipelines

Most data analysis consists of reading in some data, performing various operations on that data and, in the process, transforming it from its raw form into something we can start to extract meaning out of, and then doing some summarizing or visualization toward the end.

These steps in an analysis are typically expressed as a sequence of function calls that each change the data from one form to another. It could look like the following pseudocode:

```
my_data <- read_data("/some/path/some_file.data")
clean_data <- remove_dodgy_data(my_data)
data_summaries <- summarize(clean_data)
plot_important_things(data_summaries)
```

There isn't anything wrong with writing a data analysis in this way. But there are typically many more steps involved than listed here. When there is, you either have to get very inventive in naming the variables you are saving the data in or you have to overwrite variable names by reassigning to a variable after modifying the data. Both having many variable names and reassigning to variables can be problematic.

If you have many variables, it is easier accidentally to call a function on the wrong variable. For example, you might summarize the `my_data` variable instead of the `clean_data`. While you would get an error if you called a function with a variable name that doesn't exist, there is nothing to catch when you call a function with the wrong data. You will likely get the wrong result, and the error will not be easy to find. It would not be an error easy to debug later.

There is slightly less of a problem with reassigning to a variable. It is mostly an issue when you work with R interactively. There, if you want to go back and change part of the program you are writing, you have to go back to the start, where the data is imported. You cannot just start somewhere in the middle of the function calls with a variable that doesn't refer to the same data it did when you ran the program from scratch. It is less of a problem if you always run your R scripts from the beginning, but the typical use of R is to work with it in an interactive console or Markdown document, and there this can be a problem.

A solution, then, is not to call the functions one at a time and assign each temporary result to a variable. Instead of having four statements in the preceding example, one per function call, you would just feed the result of the first function call into the next:

```
plot_important_things(
    summarize(
        remove_dodgy_data(
            read_data("/some/path/some_file.data"))))
```

You get rid of all the variables, but the readability suffers, to put it mildly. You have to read the code from right to left and inside out.

Writing Pipelines of Function Calls

The `magrittr` package introduced a trick to alleviate this problem, which was later followed by a built-in solution in R 4.1. The solution is to introduce a "pipe operator," `%>%` in `magrittr` and `|>` in R 4.1, that lets you write the functions you want to combine from left to right but get the same effect as if you were calling one after the other and sending the result from one function to the input of the next function.

The operator works such that writing

```
x %>% f()
```

or

```
x |> f()
```

is equivalent to writing

```
f(x)
```

With the `magrittr` operator, you can leave out the parentheses, writing `x %>% f` instead, but with the built-in operator, `x |> f` is considered a syntax error.

How the two pipe operators work and how you can use them overlap, but they are not equivalent. The built-in operator is a little faster; when you write `x |> f()`, it is just syntactic sugar for `f(x)`, meaning that the two are completely equivalent and that there is no overhead in using the operator rather than a function call. With the `magrittr` operator, `x %>% f()`, you are calling a function, `%>%`, every time you use the operator, and there is some overhead to that. But there is also more flexibility to this, and the `%>%` operator is more flexible and can handle use cases that the `|>` operator cannot.

You can combine sequences of such operators such that writing

```
x |> f() |> g() |> h()
```

or

```
x %>% f() %>% g() %>% h()
```

or

```
x %>% f %>% g %>% h
```

is equivalent to writing

```
h(g(f(x)))
```

The preceding example would become

```
read_data("/some/path/some_file.data") %>%
    remove_dodgy_data %>%
    summarize %>%
    plot_important_things
```

with the `magrittr` operator, or

```
read_data("/some/path/some_file.data") |>
    remove_dodgy_data() |>
    summarize() |>
    plot_important_things()
```

with the built-in operator.

Reading code like this might still take some getting used to, but it is much easier to read than combining functions from the inside and out.

If you have ever used pipelines in UNIX shells, you should immediately see the similarities. It is the same approach for combining functions/programs. By combining several functions, which each do something relatively simple, you can create very powerful pipelines.

Writing pipelines using the `%>%` or `|>` operator is a relatively new idiom introduced to R programming, but one that is very powerful and is being used more and more in different R packages. We will use pipelines extensively in the coming chapters.

Incidentally, if you are wondering why the package that implements pipes in R is called `magrittr`, it refers to Belgian artist René Magritte who famously painted a pipe and wrote "Ceci n'est pas une pipe" ("This is not a pipe") below it. But enough about Belgian surrealists.

Writing Functions That Work with Pipelines

The pipeline operator actually does something very simple, which in turn makes it simple to write new functions that work well with it. It just takes whatever is computed on the left-hand side of it and inserts it as the first argument to the function given on the right-hand side, and it does this left to right. So `x %>% f` becomes `f(x)`, `x %>% f %>% g` becomes `f(x) %>% g` and then `g(f(x))`, and `x %>% f(y)` becomes `f(x,y)`. If you are providing additional parameters to a function in the pipeline, the left-hand side of `%>%` or `|>` is inserted before the explicit parameters passed to it.

If you want to write functions that work well with pipelines, you should, therefore, make sure that the most likely parameter to come through a pipeline is the first parameter of your function. Write your functions such that the first parameter is the data it operates on, and you have done most of the work.

For example, if you wanted a function that would sample `n` random rows of a data frame, you could write it such that it takes the data frame as the first argument and the parameter `n` as its second argument:

```
subsample_rows <- function(d, n) {
  rows <- sample(nrow(d), n)
  d[rows,]
}
```

and then you could simply pop it right into a pipeline:

```
d <- data.frame(x = rnorm(100), y = rnorm(100))
d %>% subsample_rows(n = 3)
##             x        y
## 31  0.3159150 1.3485491
## 76 -0.1553485 0.3320349
## 54 -0.5918348 0.8083360
```

or

```
d |> subsample_rows(n = 3)
##             x           y
## 69 -1.2723834 -0.01965686
## 25  0.8190089  1.05925039
## 87  2.3872326 -0.18939869
```

Since we are simulating random data here, your output will differ from mine, but you should see something similar.

The Magical "." Argument

Now, you cannot always be so lucky that all the functions you want to call in a pipeline take the left-hand side of the pipe operator as its first parameter. If this is the case, you can still use the function, though, but here the two operators differ in how easy they make it.

The operator from `magrittr`, but not the built-in operator, interprets the symbol "." in a special way. If you use "." in a function call in a pipeline, then that is where the left-hand side of the `%>%` operation goes instead of as the default first parameter of the right-hand side. So if you need the data to go as the second parameter, you put a "." there, since `x %>% f(y, .)` is equivalent to `f(y, x)`. The same goes when you need to provide the left-hand side as a named parameter since `x %>% f(y, z = .)` is equivalent to `f(y, z = x)`, something that is particularly useful when the left-hand side should be given to a model fitting function. Functions fitting a model to data are usually taking a model specification as their first parameter and the data they are fitted to as a named parameter called `data`:

```
d <- data.frame(x = rnorm(10), y = rnorm(10))
d %>% lm(y ~ x, data = .)
##
## Call:
## lm(formula = y ~ x, data = .)
##
## Coefficients:
## (Intercept)        x
##      0.2866   0.3833
```

We will return to model fitting, and what an expression such as `y ~ x` means, in a later chapter, so don't worry if it looks a little strange for now. If you are interested, you can always check the documentation for the `lm()` function.

The built-in operator does not interpret "." this way, and `d |> lm(y ~ x, data = .)` will give you an error (unless you have defined "." somewhere, which you probably shouldn't). The `|>` operator always puts the left-hand side as the first argument to the right-hand side. If that doesn't fit your function, you have to adapt the function.

With `lm`, the data is not the first argument, but we can make a function where it is:

```
my_lm <- function(d) lm(y ~ x, data = d)
d |> my_lm()
##
## Call:
## lm(formula = y ~ x, data = d)
##
## Coefficients:
## (Intercept)          x
##      0.2866     0.3833
```

We usually don't like having such specialized functions lying around, and we don't have any use for it outside of the pipeline, so this is not ideal. However, we don't have to first define the function, give it a name, and then use it. We could just use the function definition as it is:

```
d |> (function(d) lm(y ~ x, data = d))()
##
## Call:
## lm(formula = y ~ x, data = d)
##
## Coefficients:
## (Intercept)          x
##      0.2866     0.3833
```

The syntax here might look a little odd at first glance, with the function definition in parentheses and then the extra `()` after that, but it is really the same syntax as what we have been using so far. We have written pipes such as `d |> f()` where `f` refers to a function. It is the same now, but instead of a function name, we have an expression, `(function(d) lm(...))`, that gives us a function. It needs to be in parentheses so the `()` that comes after the function are not considered part of the function body. In other words, take `d |> f()` and put in `(function(d) lm(...))` instead of `f`, and you get the preceding expression.

Such functions that we do not give a name are called anonymous functions, or with a reference to theoretical computer science, lambda expressions. From R 4.1, perhaps to alleviate that using `|>` without the "." is cumbersome, there is a slightly shorter way to write them. Instead of writing `function(...)`, you can use `\(...)` and get

```
d |> (\(d) lm(y ~ x, data = d))()
##
## Call:
## lm(formula = y ~ x, data = d)
##
## Coefficients:
## (Intercept)            x
##      0.2866       0.3833
```

The notation `\()` is supposed to look like the Greek letter lambda, λ.

Of course, even with the shorter syntax for anonymous functions, writing `d |> (\(d) lm(y ~ x, data = d))()` instead of just `lm(y ~ x, data = d)` doesn't give us much, and it is more cumbersome to use the `|>` operator when you try to pipe together functions where the data doesn't flow from the first argument to the first argument from function to function. If you are in a situation like that, you will enjoy using the `magrittr` pipe more.

Anonymous functions do have their uses, though, both for the built-in and `magrittr`'s pipe operator. Pipelines are great when you can call existing functions one after another, but what happens if you need a step in the pipeline where there is no function doing what you want? Here, anonymous functions usually are the right solution.

As an example, consider a function that plots the variable `y` against the variable `x` and fits and plots a linear model of `y` against `x`. We can define and name such a function to get the following code:

```
plot_and_fit <- function(d) {
  plot(y ~ x, data = d)
  abline(lm(y ~ x, data = d))
}
x <- rnorm(20)
y <- x + rnorm(20)
data.frame(x, y) |> plot_and_fit()
```

Since giving the function a name doesn't affect how the function works, it isn't necessary to do so; we can just put the code that defined the function where the name of the function goes to get this:

```
data.frame(x, y) |> (\(d) {
    plot(y ~ x, data = d)
    abline(lm(y ~ x, data = d))
})()
```

with the built-in operator, or like this

```
data.frame(x, y) %>% (\(d) {
    plot(y ~ x, data = d)
    abline(lm(y ~ x, data = d))
})()
```

with the `%>%` operator.

With the `magrittr` pipe operator, we could also leave out the final `()` and do simply:

```
data.frame(x, y) %>% (\(d) {
    plot(y ~ x, data = d)
    abline(lm(y ~ x, data = d))
})
```

This is because the `%>%` operator takes both a function call and a function on the right-hand side, so we can write `x %>% f()` or `x %>% f`, and similarly we can write `x %>% (\(x) ...)()` or `x %>% (\(x) ...)`. You cannot leave out the parentheses around the function definition, though. A function definition is also a function call, and the `%>%` operator would try to put the left-hand side of the operator into that function call, which would give you an error. The `|>` operator explicitly checks if you are trying to define a function as the right-hand side and tells you that this is not allowed.

With the `magrittr` operator, though, you do not need to explicitly define an anonymous function this way; you can use "." to simulate the same effect:

```
data.frame(x, y) %>% {
  plot(y ~ x, data = .)
  abline(lm(y ~ x, data = .))
}
```

By putting the two operations in curly braces, we effectively make a function, and the first argument of the function goes where we put the ".".

The `magrittr` operator does more with "." than just changing the order of parameters. You can use "." more than once when calling a function, and you can use it in expressions or in function calls:

```
rnorm(4) %>% data.frame(x = ., is_negative = . < 0)
##            x is_negative
## 1 -1.5782344        TRUE
## 2 -0.1215720        TRUE
## 3 -1.7966768        TRUE
## 4 -0.4755915        TRUE
rnorm(4) %>% data.frame(x = ., y = abs(.))
##            x         y
## 1 -0.8841023 0.8841023
## 2 -3.4980590 3.4980590
## 3 -0.3819834 0.3819834
## 4  0.9776881 0.9776881
```

There is one caveat: if "." only appears in function calls, it is still given as the first argument to the function on the right-hand side of `%>%`:

```
rnorm(4) %>% data.frame(x = sin(.), y = cos(.))
##            .          x          y
## 1 -0.5580409 -0.5295254  0.8482941
## 2 -0.6264551 -0.5862767  0.8101109
## 3 -0.5304512 -0.5059226  0.8625789
## 4  1.8976216  0.9470663 -0.3210380
```

The reason is that it is more common to see expressions with function calls like this when the full data is also needed than when it is not. So by default `f(g(.),h(.))` gets translated into `f(.,g(.),h(.))`. If you want to avoid this behavior, you can put curly brackets around the function call since `{f(g(.),h(.))}` is equivalent to `f(g(.),h(.))`. (I will explain the meaning of the curly brackets later). You can get both the behavior `f(.,g(.),h(.))` and the behavior `{f(g(.),h(.))}` in function calls in a pipeline; the default is just the most common case.

Other Pipeline Operations

The `%>%` and `|>` operators are a very powerful mechanism for specifying data analysis pipelines, but there are some special cases where a slightly different behavior is needed, and the `magrittr` package provides some of these. To get them, you need to import the package with `library(magrittr)`. If you use `library(tidyverse)` to load the tidy-verse framework, you only get `%>%`; you need to explicitly load `magrittr` to get the other operators.

One case is when you need to refer to the parameters in a data frame you get from the left-hand side of the pipe expression directly. In many functions, you can get to the parameters of a data frame just by naming them, as we have seen earlier in `lm` and `plot`, but there are cases where that is not so simple.

You can do that by indexing "." like this:

```
d <- data.frame(x = rnorm(10), y = 4 + rnorm(10))
d %>% {data.frame(mean_x = mean(.$x), mean_y = mean(.$y))}
##       mean_x   mean_y
## 1 0.09496017 3.538881
```

but if you use the operator `%$%` instead of `%>%`, you can get to the variables just by naming them instead:

```
d %$% data.frame(mean_x = mean(x), mean_y = mean(y))
##       mean_x   mean_y
## 1 0.09496017 3.538881
```

Another common case is when you want to output or plot some intermediate result of a pipeline. You can, of course, write the first part of a pipeline, run data through it and store the result in a parameter, output or plot what you want, and then continue from the stored data. But you can also use the `%T>%` (tee) operator. It works like the `%>%` operator, but where `%>%` passes the result of the right-hand side of the expression on, `%T>%` passes on the result of the left-hand side. The right-hand side is computed but not passed on, which is perfect if you only want a step for its side effect, like printing some summary:

```
d <- data.frame(x = rnorm(10), y = rnorm(10))
d %T>% plot(y ~ x, data = .) %>% lm(y ~ x, data = .)
```

The final operator is `%<>%`, which does something I warned against earlier—it assigns the result of a pipeline back to a variable on the left. Sometimes, you do want this behavior—for instance, if you do some data cleaning right after loading the data and you never want to use anything between the raw and the cleaned data, you can use `%<>%`:

```
d <- read_my_data("/path/to/data")
d %<>% clean_data
```

I use it sparingly and would prefer just to pass this case through a pipeline:

```
d <- read_my_data("/path/to/data") %>% clean_data
```

Coding and Naming Conventions

People have been developing R code for a long time, and they haven't been all that consistent in how they do it. So as you use R packages, you will see many different conventions on how code is written and especially how variables and functions are named.

How you choose to write your code is entirely up to you as long as you are consistent with it. It helps somewhat if your code matches the packages you use, just to make everything easier to read, but it is up to you.

A few words on naming are worth going through, though. There are three ways people typically name their variables, data, or functions, and these are

```
underscore_notation(x, y)
camelBackNotation(x, y)
dot.notation(x, y)
```

You are probably familiar with the first two notations, but if you have used Python or Java or C/C++ before, the dot notation looks like method calls in object-oriented programming. It is not (although it is related to it). The dot in the name doesn't mean method call. R just allows you to use dots in variable and function names.

I will mostly use the underscore notation in this book, but you can do whatever you want. I would recommend that you stay away from the dot notation, though. There are good reasons for this. R put some interpretation into what dots mean in function names, as we will see when we visit object-oriented programming in the second part of the book, so you can get into some trouble. The built-in functions in R often use dots in function names, but it is a dangerous path so you should probably stay away from it unless you are absolutely sure that you are avoiding the pitfalls that are in it.

Exercises

Mean of Positive Values

You can simulate values from the normal distribution using the `rnorm()` function. Its first argument is the number of samples you want, and if you do not specify other values, it will sample from the $N\,(0, 1)$ distribution.

Write a pipeline that takes samples from this function as input, remove the negative values, and compute the mean of the rest. Hint: One way to remove values is to replace them with missing values (`NA`); if a vector has missing values, the `mean()` function can ignore them if you give it the option `na.rm = TRUE`.

Root Mean Square Error

If you have "true" values, t = (t_1, ... , t_n), and "predicted" values, y = (y_1, ... , y_n), then the root mean square error is defined as

$$\mathrm{RMSE}\,(\mathrm{t}, \mathrm{y}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(t_i - y_i\right)2}.$$

Write a pipeline that computes this from a data frame containing the t and y values. Remember that you can do this by first computing the square difference in one expression, then computing the mean of that in the next step, and finally computing the square root of this. The R function for computing the square root is `sqrt()`.