6

Functions

Functions are a fundamental building block of R: to master many of the more advanced techniques in this book, you need a solid foundation in how functions work. You've probably already created many R functions, and you're familiar with the basics of how they work. The focus of this chapter is to turn your existing, informal knowledge of functions into a rigorous understanding of what functions are and how they work. You'll see some interesting tricks and techniques in this chapter, but most of what you'll learn will be more important as the building blocks for more advanced techniques.

The most important thing to understand about R is that functions are objects in their own right. You can work with them exactly the same way you work with any other type of object. This theme will be explored in depth in Chapter 10.

Quiz

Answer the following questions to see if you can safely skip this chapter. You can find the answers at the end of the chapter in Section 6.7.

1. What are the three components of a function?
2. What does the following code return?

```
y <- 10
f1 <- function(x) {
  function() {
    x + 10
  }
}
f1(1)()
```

3. How would you more typically write this code?

```
`+`(1, `*`(2, 3))
```

4. How could you make this call easier to read?

   ```
   mean(, TRUE, x = c(1:10, NA))
   ```

5. Does the following function throw an error when called? Why/why not?

   ```
   f2 <- function(a, b) {
     a * 10
   }
   f2(10, stop("This is an error!"))
   ```

6. What is an infix function? How do you write it? What's a replacement function? How do you write it?
7. What function do you use to ensure that a cleanup action occurs regardless of how a function terminates?

Outline

- Section 6.1 describes the three main components of a function.
- Section 6.2 teaches you how R finds values from names, the process of lexical scoping.
- Section 6.3 shows you that everything that happens in R is a result of a function call, even if it doesn't look like it.
- Section 6.4 discusses the three ways of supplying arguments to a function, how to call a function given a list of arguments, and the impact of lazy evaluation.
- Section 6.5 describes two special types of function: infix and replacement functions.
- Section 6.6 discusses how and when functions return values, and how you can ensure that a function does something before it exits.

Prerequisites

The only package you'll need is `pryr`, which is used to explore what happens when modifying vectors in place. Install it with `install.packages("pryr")`.

6.1 Function components

All R functions have three parts:

- the `body()`, the code inside the function.
- the `formals()`, the list of arguments which controls how you can call the function.
- the `environment()`, the "map" of the location of the function's variables.

When you print a function in R, it shows you these three important components. If the environment isn't displayed, it means that the function was created in the global environment.

```
f <- function(x) x^2
f
#> function(x) x^2

formals(f)
#> $x
body(f)
#> x^2
environment(f)
#> <environment: R_GlobalEnv>
```

The assignment forms of `body()`, `formals()`, and `environment()` can also be used to modify functions.

Like all objects in R, functions can also possess any number of additional `attributes()`. One attribute used by base R is "srcref", short for source reference, which points to the source code used to create the function. Unlike `body()`, this contains code comments and other formatting. You can also add attributes to a function. For example, you can set the `class()` and add a custom `print()` method.

6.1.1 Primitive functions

There is one exception to the rule that functions have three components. Primitive functions, like `sum()`, call C code directly with `.Primitive()`

and contain no R code. Therefore their `formals()`, `body()`, and `environment()` are all `NULL`:

```
sum
#> function (..., na.rm = FALSE)  .Primitive("sum")
formals(sum)
#> NULL
body(sum)
#> NULL
environment(sum)
#> NULL
```

Primitive functions are only found in the `base` package, and since they operate at a low level, they can be more efficient (primitive replacement functions don't have to make copies), and can have different rules for argument matching (e.g., `switch` and `call`). This, however, comes at a cost of behaving differently from all other functions in R. Hence the R core team generally avoids creating them unless there is no other option.

6.1.2 Exercises

1. What function allows you to tell if an object is a function? What function allows you to tell if a function is a primitive function?
2. This code makes a list of all functions in the base package.

   ```
   objs <- mget(ls("package:base"), inherits = TRUE)
   funs <- Filter(is.function, objs)
   ```

   Use it to answer the following questions:

   a. Which base function has the most arguments?
   b. How many base functions have no arguments? What's special about those functions?
   c. How could you adapt the code to find all primitive functions?
3. What are the three important components of a function?
4. When does printing a function not show what environment it was created in?

6.2 Lexical scoping

Scoping is the set of rules that govern how R looks up the value of a symbol. In the example below, scoping is the set of rules that R applies to go from the symbol `x` to its value `10`:

```
x <- 10
x
#> [1] 10
```

Understanding scoping allows you to:

- build tools by composing functions, as described in Chapter 10.
- overrule the usual evaluation rules and do non-standard evaluation, as described in Chapter 13.

R has two types of scoping: **lexical scoping**, implemented automatically at the language level, and **dynamic scoping**, used in select functions to save typing during interactive analysis. We discuss lexical scoping here because it is intimately tied to function creation. Dynamic scoping is described in more detail in Section 13.3.

Lexical scoping looks up symbol values based on how functions were nested when they were created, not how they are nested when they are called. With lexical scoping, you don't need to know how the function is called to figure out where the value of a variable will be looked up. You just need to look at the function's definition.

The "lexical" in lexical scoping doesn't correspond to the usual English definition ("of or relating to words or the vocabulary of a language as distinguished from its grammar and construction") but comes from the computer science term "lexing", which is part of the process that converts code represented as text to meaningful pieces that the programming language understands.

There are four basic principles behind R's implementation of lexical scoping:

- name masking
- functions vs. variables

- a fresh start
- dynamic lookup

You probably know many of these principles already, although you might not have thought about them explicitly. Test your knowledge by mentally running through the code in each block before looking at the answers.

6.2.1 Name masking

The following example illustrates the most basic principle of lexical scoping, and you should have no problem predicting the output.

```
f <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
f()
rm(f)
```

If a name isn't defined inside a function, R will look one level up.

```
x <- 2
g <- function() {
  y <- 1
  c(x, y)
}
g()
rm(x, g)
```

The same rules apply if a function is defined inside another function: look inside the current function, then where that function was defined, and so on, all the way up to the global environment, and then on to other loaded packages. Run the following code in your head, then confirm the output by running the R code.

```
x <- 1
h <- function() {
  y <- 2
```

```
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
h()
rm(x, h)
```

The same rules apply to closures, functions created by other functions. Closures will be described in more detail in Chapter 10; here we'll just look at how they interact with scoping. The following function, `j()`, returns a function. What do you think this function will return when we call it?

```
j <- function(x) {
  y <- 2
  function() {
    c(x, y)
  }
}
k <- j(1)
k()
rm(j, k)
```

This seems a little magical (how does R know what the value of `y` is after the function has been called). It works because `k` preserves the environment in which it was defined and because the environment includes the value of `y`. Chapter 8 gives some pointers on how you can dive in and figure out what values are stored in the environment associated with each function.

6.2.2 Functions vs. variables

The same principles apply regardless of the type of associated value — finding functions works exactly the same way as finding variables:

```
l <- function(x) x + 1
m <- function() {
  l <- function(x) x * 2
  l(10)
```

```
}
m()
#> [1] 20
rm(l, m)
```

For functions, there is one small tweak to the rule. If you are using a name in a context where it's obvious that you want a function (e.g., `f(3)`), R will ignore objects that are not functions while it is searching. In the following example `n` takes on a different value depending on whether R is looking for a function or a variable.

```
n <- function(x) x / 2
o <- function() {
  n <- 10
  n(n)
}
o()
#> [1] 5
rm(n, o)
```

However, using the same name for functions and other objects will make for confusing code, and is generally best avoided.

6.2.3 A fresh start

What happens to the values in between invocations of a function? What will happen the first time you run this function? What will happen the second time? (If you haven't seen `exists()` before: it returns `TRUE` if there's a variable of that name, otherwise it returns `FALSE`.)

```
j <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  print(a)
}
j()
rm(j)
```

You might be surprised that it returns the same value, 1, every time. This is because every time a function is called, a new environment is created to host execution. A function has no way to tell what happened the last time it was run; each invocation is completely independent. (We'll see some ways to get around this in Section 10.3.2.)

6.2.4 Dynamic lookup

Lexical scoping determines where to look for values, not when to look for them. R looks for values when the function is run, not when it's created. This means that the output of a function can be different depending on objects outside its environment:

```
f <- function() x
x <- 15
f()
#> [1] 15

x <- 20
f()
#> [1] 20
```

You generally want to avoid this behaviour because it means the function is no longer self-contained. This is a common error — if you make a spelling mistake in your code, you won't get an error when you create the function, and you might not even get one when you run the function, depending on what variables are defined in the global environment.

One way to detect this problem is the `findGlobals()` function from `codetools`. This function lists all the external dependencies of a function:

```
f <- function() x + 1
codetools::findGlobals(f)
#> [1] "+" "x"
```

Another way to try and solve the problem would be to manually change the environment of the function to the `emptyenv()`, an environment which contains absolutely nothing:

```
environment(f) <- emptyenv()
f()
#> Error: could not find function "+"
```

This doesn't work because R relies on lexical scoping to find *everything*, even the + operator. It's never possible to make a function completely self-contained because you must always rely on functions defined in base R or other packages.

You can use this same idea to do other things that are extremely ill-advised. For example, since all of the standard operators in R are functions, you can override them with your own alternatives. If you ever are feeling particularly evil, run the following code while your friend is away from their computer:

```
`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
#>  [1] 3 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 3 3 3 4 3
#> [30] 3 3 3 3 3 4 3 3 4 3 3 3 3 3 3 3 3 3 3 3
rm("(")
```

This will introduce a particularly pernicious bug: 10% of the time, 1 will be added to any numeric calculation inside parentheses. This is another good reason to regularly restart with a clean R session!

6.2.5 Exercises

1. What does the following code return? Why? What does each of the three `c`'s mean?

   ```
   c <- 10
   c(c = c)
   ```

2. What are the four principles that govern how R looks for values?
3. What does the following function return? Make a prediction before running the code yourself.

   ```
   f <- function(x) {
     f <- function(x) {
       f <- function(x) {
   ```

```
      x ^ 2
    }
    f(x) + 1
  }
  f(x) * 2
}
f(10)
```

6.3 Every operation is a function call

"To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call."

— John Chambers

The previous example of redefining `(` works because every operation in R is a function call, whether or not it looks like one. This includes infix operators like `+`, control flow operators like `for`, `if`, and `while`, subsetting operators like `[]` and `$`, and even the curly brace `{`. This means that each pair of statements in the following example is exactly equivalent. Note that `` ` ``, the backtick, lets you refer to functions or variables that have otherwise reserved or illegal names:

```
x <- 10; y <- 5
x + y
#> [1] 15
`+`(x, y)
#> [1] 15

for (i in 1:2) print(i)
#> [1] 1
#> [1] 2
`for`(i, 1:2, print(i))
#> [1] 1
#> [1] 2

if (i == 1) print("yes!") else print("no.")
```

```
#> [1] "no."
`if`(i == 1, print("yes!"), print("no."))
#> [1] "no."

x[3]
#> [1] NA
`[`(x, 3)
#> [1] NA

{ print(1); print(2); print(3) }
#> [1] 1
#> [1] 2
#> [1] 3
`{`(print(1), print(2), print(3))
#> [1] 1
#> [1] 2
#> [1] 3
```

It is possible to override the definitions of these special functions, but this is almost certainly a bad idea. However, there are occasions when it might be useful: it allows you to do something that would have otherwise been impossible. For example, this feature makes it possible for the `dplyr` package to translate R expressions into SQL expressions. Chapter 15 uses this idea to create domain specific languages that allow you to concisely express new concepts using existing R constructs.

It's more often useful to treat special functions as ordinary functions. For example, we could use `sapply()` to add 3 to every element of a list by first defining a function `add()`, like this:

```
add <- function(x, y) x + y
sapply(1:10, add, 3)
#>  [1]  4  5  6  7  8  9 10 11 12 13
```

But we can also get the same effect using the built-in `+` function.

```
sapply(1:5, `+`, 3)
#> [1] 4 5 6 7 8
sapply(1:5, "+", 3)
#> [1] 4 5 6 7 8
```

Note the difference between `` `+` `` and `"+"`. The first one is the value of the object called `+`, and the second is a string containing the character `+`. The

second version works because `lapply` can be given the name of a function instead of the function itself: if you read the source of `lapply()`, you'll see the first line uses `match.fun()` to find functions given their names.

A more useful application is to combine `lapply()` or `sapply()` with subsetting:

```
x <- list(1:3, 4:9, 10:12)
sapply(x, "[", 2)
#> [1]  2  5 11

# equivalent to
sapply(x, function(x) x[2])
#> [1]  2  5 11
```

Remembering that everything that happens in R is a function call will help you in Chapter 14.

6.4 Function arguments

It's useful to distinguish between the formal arguments and the actual arguments of a function. The formal arguments are a property of the function, whereas the actual or calling arguments can vary each time you call the function. This section discusses how calling arguments are mapped to formal arguments, how you can call a function given a list of arguments, how default arguments work, and the impact of lazy evaluation.

6.4.1 Calling functions

When calling a function you can specify arguments by position, by complete name, or by partial name. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.

```
f <- function(abcdef, bcde1, bcde2) {
  list(a = abcdef, b1 = bcde1, b2 = bcde2)
}
```

```
str(f(1, 2, 3))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3
str(f(2, 3, abcdef = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# Can abbreviate long argument names:
str(f(2, 3, a = 1))
#> List of 3
#>  $ a : num 1
#>  $ b1: num 2
#>  $ b2: num 3

# But this doesn't work because abbreviation is ambiguous
str(f(1, 3, b = 1))
#> Error: argument 3 matches multiple formal arguments
```

Generally, you only want to use positional matching for the first one or two arguments; they will be the most commonly used, and most readers will know what they are. Avoid using positional matching for less commonly used arguments, and only use readable abbreviations with partial matching. (If you are writing code for a package that you want to publish on CRAN you can not use partial matching, and must use complete names.) Named arguments should always come after unnamed arguments. If a function uses `...` (discussed in more detail below), you can only specify arguments listed after `...` with their full name.

These are good calls:

```
mean(1:10)
mean(1:10, trim = 0.05)
```

This is probably overkill:

```
mean(x = 1:10)
```

And these are just confusing:

```
mean(1:10, n = T)
mean(1:10, , FALSE)
mean(1:10, 0.05)
mean(, TRUE, x = c(1:10, NA))
```

6.4.2 Calling a function given a list of arguments

Suppose you had a list of function arguments:

```
args <- list(1:10, na.rm = TRUE)
```

How could you then send that list to `mean()`? You need `do.call()`:

```
do.call(mean, list(1:10, na.rm = TRUE))
#> [1] 5.5
# Equivalent to
mean(1:10, na.rm = TRUE)
#> [1] 5.5
```

6.4.3 Default and missing arguments

Function arguments in R can have default values.

```
f <- function(a = 1, b = 2) {
  c(a, b)
}
f()
#> [1] 1 2
```

Since arguments in R are evaluated lazily (more on that below), the default value can be defined in terms of other arguments:

```
g <- function(a = 1, b = a * 2) {
  c(a, b)
}
g()
#> [1] 1 2
g(10)
#> [1] 10 20
```

Default arguments can even be defined in terms of variables created within the function. This is used frequently in base R functions, but I think it is bad practice, because you can't understand what the default values will be without reading the complete source code.

```
h <- function(a = 1, b = d) {
  d <- (a + 1) ^ 2
  c(a, b)
}
h()
#> [1] 1 4
h(10)
#> [1]  10 121
```

You can determine if an argument was supplied or not with the `missing()` function.

```
i <- function(a, b) {
  c(missing(a), missing(b))
}
i()
#> [1] TRUE TRUE
i(a = 1)
#> [1] FALSE  TRUE
i(b = 2)
#> [1]  TRUE FALSE
i(1, 2)
#> [1] FALSE FALSE
```

Sometimes you want to add a non-trivial default value, which might take several lines of code to compute. Instead of inserting that code in the function definition, you could use `missing()` to conditionally compute it if needed. However, this makes it hard to know which arguments are required and which are optional without carefully reading the documentation. Instead, I usually set the default value to `NULL` and use `is.null()` to check if the argument was supplied.

6.4.4 Lazy evaluation

By default, R function arguments are lazy — they're only evaluated if they're actually used:

```
f <- function(x) {
  10
}
f(stop("This is an error!"))
#> [1] 10
```

If you want to ensure that an argument is evaluated you can use `force()`:

```
f <- function(x) {
  force(x)
  10
}
f(stop("This is an error!"))
#> Error: This is an error!
```

This is important when creating closures with `lapply()` or a loop:

```
add <- function(x) {
  function(y) x + y
}
adders <- lapply(1:10, add)
adders[[1]](10)
#> [1] 20
adders[[10]](10)
#> [1] 20
```

`x` is lazily evaluated the first time that you call one of the adder functions. At this point, the loop is complete and the final value of `x` is 10. Therefore all of the adder functions will add 10 on to their input, probably not what you wanted! Manually forcing evaluation fixes the problem:

```
add <- function(x) {
  force(x)
  function(y) x + y
}
adders2 <- lapply(1:10, add)
adders2[[1]](10)
#> [1] 11
adders2[[10]](10)
#> [1] 20
```

This code is exactly equivalent to

```
add <- function(x) {
  x
  function(y) x + y
}
```

because the force function is defined as `force <- function(x) x`. However, using this function clearly indicates that you're forcing evaluation, not that you've accidentally typed `x`.

Default arguments are evaluated inside the function. This means that if the expression depends on the current environment the results will differ depending on whether you use the default value or explicitly provide one.

```
f <- function(x = ls()) {
  a <- 1
  x
}

# ls() evaluated inside f:
f()
#> [1] "a" "x"

# ls() evaluated in global environment:
f(ls())
#>  [1] "add"      "adders"   "adders2"  "args"     "f"
#>  [6] "funs"     "g"        "h"        "i"        "metadata"
#> [11] "objs"     "x"        "y"
```

More technically, an unevaluated argument is called a **promise**, or (less commonly) a thunk. A promise is made up of two parts:

- The expression which gives rise to the delayed computation. (It can be accessed with `substitute()`. See Chapter 13 for more details.)
- The environment where the expression was created and where it should be evaluated.

The first time a promise is accessed the expression is evaluated in the environment where it was created. This value is cached, so that subsequent

access to the evaluated promise does not recompute the value (but the original expression is still associated with the value, so `substitute()` can continue to access it). You can find more information about a promise using `pryr::promise_info()`. This uses some C++ code to extract information about the promise without evaluating it, which is impossible to do in pure R code.

Laziness is useful in if statements — the second statement below will be evaluated only if the first is true. If it wasn't, the statement would return an error because `NULL > 0` is a logical vector of length 0 and not a valid input to `if`.

```
x <- NULL
if (!is.null(x) && x > 0) {

}
```

We could implement "&&" ourselves:

```
`&&` <- function(x, y) {
  if (!x) return(FALSE)
  if (!y) return(FALSE)

  TRUE
}
a <- NULL
!is.null(a) && a > 0
#> [1] FALSE
```

This function would not work without lazy evaluation because both `x` and `y` would always be evaluated, testing `a > 0` even when `a` was NULL.

Sometimes you can also use laziness to eliminate an if statement altogether. For example, instead of:

```
if (is.null(a)) stop("a is null")
#> Error: a is null
```

You could write:

```
!is.null(a) || stop("a is null")
#> Error: a is null
```

6.4.5 ...

There is a special argument called ... . This argument will match any arguments not otherwise matched, and can be easily passed on to other functions. This is useful if you want to collect arguments to call another function, but you don't want to prespecify their possible names. ... is often used in conjunction with S3 generic functions to allow individual methods to be more flexible.

One relatively sophisticated user of ... is the base `plot()` function. `plot()` is a generic method with arguments `x`, `y` and ... . To understand what ... does for a given function we need to read the help: "Arguments to be passed to methods, such as graphical parameters". Most simple invocations of `plot()` end up calling `plot.default()` which has many more arguments, but also has ... . Again, reading the documentation reveals that ... accepts "other graphical parameters", which are listed in the help for `par()`. This allows us to write code like:

```
plot(1:5, col = "red")
plot(1:5, cex = 5, pch = 20)
```

This illustrates both the advantages and disadvantages of ...: it makes `plot()` very flexible, but to understand how to use it, we have to carefully read the documentation. Additionally, if we read the source code for `plot.default`, we can discover undocumented features. It's possible to pass along other arguments to `Axis()` and `box()`:

```
plot(1:5, bty = "u")
plot(1:5, labels = FALSE)
```

To capture ... in a form that is easier to work with, you can use `list(...)`. (See Section 13.5.2 for other ways to capture ... without evaluating the arguments.)

```
f <- function(...) {
  names(list(...))
}
f(a = 1, b = 2)
#> [1] "a" "b"
```

Using ... comes at a price — any misspelled arguments will not raise an error, and any arguments after ... must be fully named. This makes it easy for typos to go unnoticed:

```
sum(1, 2, NA, na.mr = TRUE)
#> [1] NA
```

It's often better to be explicit rather than implicit, so you might instead ask users to supply a list of additional arguments. That's certainly easier if you're trying to use ... with multiple additional functions.

6.4.6 Exercises

1. Clarify the following list of odd function calls:

   ```
   x <- sample(replace = TRUE, 20, x = c(1:10, NA))
   y <- runif(min = 0, max = 1, 20)
   cor(m = "k", y = y, u = "p", x = x)
   ```

2. What does this function return? Why? Which principle does it illustrate?

   ```
   f1 <- function(x = {y <- 1; 2}, y = 0) {
     x + y
   }
   f1()
   ```

3. What does this function return? Why? Which principle does it illustrate?

   ```
   f2 <- function(x = z) {
     z <- 100
     x
   }
   f2()
   ```

6.5 Special calls

R supports two additional syntaxes for calling special types of functions: infix and replacement functions.

6.5.1 Infix functions

Most functions in R are "prefix" operators: the name of the function comes before the arguments. You can also create infix functions where the function name comes in between its arguments, like `+` or `-`. All user created infix functions must start and end with `%` and R comes with the following infix functions predefined: `%%`, `%*%`, `%/%`, `%in%`, `%o%`, `%x%`. (The complete list of built-in infix operators that don't need `%` is: `::`, `:::`, `$`, `@`, `^`, `*`, `/`, `+`, `-`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&`, `&&`, `|`, `||`, `~`, `<-`, `<<-`)

For example, we could create a new operator that pastes together strings:

```
`%+%` <- function(a, b) paste(a, b, sep = "")
"new" %+% " string"
#> [1] "new string"
```

Note that when creating the function, you have to put the name in backticks because it's a special name. This is just a syntactic sugar for an ordinary function call; as far as R is concerned there is no difference between these two expressions:

```
"new" %+% " string"
#> [1] "new string"
`%+%`("new", " string")
#> [1] "new string"
```

Or indeed between

```
1 + 5
#> [1] 6
`+`(1, 5)
#> [1] 6
```

The names of infix functions are more flexible than regular R functions: they can contain any sequence of characters (except "%", of course). You will need to escape any special characters in the string used to define the function, but not when you call it:

```
`% %` <- function(a, b) paste(a, b)
`%'%` <- function(a, b) paste(a, b)
`%/\\%` <- function(a, b) paste(a, b)
```

```
"a" % % "b"
#> [1] "a b"
"a" %'% "b"
#> [1] "a b"
"a" %/\% "b"
#> [1] "a b"
```

R's default precedence rules mean that infix operators are composed from left to right:

```
`%-%` <- function(a, b) paste0("(", a, " %-% ", b, ")")
"a" %-% "b" %-% "c"
#> [1] "((a %-% b) %-% c)"
```

There's one infix function that I use very often. It's inspired by Ruby's `||` logical or operator, although it works a little differently in R because Ruby has a more flexible definition of what evaluates to `TRUE` in an if statement. It's useful as a way of providing a default value in case the output of another function is `NULL`:

```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

6.5.2 Replacement functions

Replacement functions act like they modify their arguments in place, and have the special name `xxx<-`. They typically have two arguments (`x` and `value`), although they can have more, and they must return the modified object. For example, the following function allows you to modify the second element of a vector:

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
x
#>  [1]  1  5  3  4  5  6  7  8  9 10
```

When R evaluates the assignment `second(x) <- 5`, it notices that the left hand side of the `<-` is not a simple name, so it looks for a function named `second<-` to do the replacement.

I say they "act" like they modify their arguments in place, because they actually create a modified copy. We can see that by using `pryr::address()` to find the memory address of the underlying object.

```
library(pryr)
x <- 1:10
address(x)
#> [1] "0x7fb3024fad48"
second(x) <- 6L
address(x)
#> [1] "0x7fb3059d9888"
```

Built-in functions that are implemented using `.Primitive()` will modify in place:

```
x <- 1:10
address(x)
#> [1] "0x103945110"

x[2] <- 7L
address(x)
#> [1] "0x103945110"
```

It's important to be aware of this behaviour since it has important performance implications.

If you want to supply additional arguments, they go in between `x` and `value`:

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- 10
x
#>  [1] 10  6  3  4  5  6  7  8  9 10
```

When you call `modify(x, 1) <- 10`, behind the scenes R turns it into:

```
x <- `modify<-`(x, 1, 10)
```

This means you can't do things like:

```
modify(get("x"), 1) <- 10
```

because that gets turned into the invalid code:

```
get("x") <- `modify<-`(get("x"), 1, 10)
```

It's often useful to combine replacement and subsetting:

```
x <- c(a = 1, b = 2, c = 3)
names(x)
#> [1] "a" "b" "c"
names(x)[2] <- "two"
names(x)
#> [1] "a"   "two" "c"
```

This works because the expression `names(x)[2] <- "two"` is evaluated as if you had written:

```
`*tmp*` <- names(x)
`*tmp*`[2] <- "two"
names(x) <- `*tmp*`
```

(Yes, it really does create a local variable named `*tmp*`, which is removed afterwards.)

6.5.3 Exercises

1. Create a list of all the replacement functions found in the base package. Which ones are primitive functions?
2. What are valid names for user created infix functions?
3. Create an infix `xor()` operator.
4. Create infix versions of the set functions `intersect()`, `union()`, and `setdiff()`.
5. Create a replacement function that modifies a random location in a vector.

6.6 Return values

The last expression evaluated in a function becomes the return value, the result of invoking the function.

```
f <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
f(5)
#> [1] 0
f(15)
#> [1] 10
```

Generally, I think it's good style to reserve the use of an explicit `return()` for when you are returning early, such as for an error, or a simple case of the function. This style of programming can also reduce the level of indentation, and generally make functions easier to understand because you can reason about them locally.

```
f <- function(x, y) {
  if (!x) return(y)

  # complicated processing here
}
```

Functions can return only a single object. But this is not a limitation because you can return a list containing any number of objects.

The functions that are the easiest to understand and reason about are pure functions: functions that always map the same input to the same output and have no other impact on the workspace. In other words, pure functions have no **side effects**: they don't affect the state of the world in any way apart from the value they return.

R protects you from one type of side effect: most R objects have copy-on-modify semantics. So modifying a function argument does not change the original value:

```
f <- function(x) {
  x$a <- 2
  x
}
x <- list(a = 1)
f(x)
#> $a
#> [1] 2
x$a
#> [1] 1
```

(There are two important exceptions to the copy-on-modify rule: environments and reference classes. These can be modified in place, so extra care is needed when working with them.)

This is notably different to languages like Java where you can modify the inputs of a function. This copy-on-modify behaviour has important performance consequences which are discussed in depth in Chapter 17. (Note that the performance consequences are a result of R's implementation of copy-on-modify semantics; they are not true in general. Clojure is a new language that makes extensive use of copy-on-modify semantics with limited performance consequences.)

Most base R functions are pure, with a few notable exceptions:

- `library()` which loads a package, and hence modifies the search path.
- `setwd()`, `Sys.setenv()`, `Sys.setlocale()` which change the working directory, environment variables, and the locale, respectively.
- `plot()` and friends which produce graphical output.
- `write()`, `write.csv()`, `saveRDS()`, etc. which save output to disk.
- `options()` and `par()` which modify global settings.
- S4 related functions which modify global tables of classes and methods.
- Random number generators which produce different numbers each time you run them.

It's generally a good idea to minimise the use of side-effects, and where possible, to minimise the footprint of side effects by separating pure from impure functions. Pure functions are easier to test (because all you need to worry about are the input values and the output), and are less likely

to work differently on different versions of R or on different platforms. For example, this is one of the motivating principles of ggplot2: most operations work on an object that represents a plot, and only the final `print` or `plot` call has the side effect of actually drawing the plot.

Functions can return `invisible` values, which are not printed out by default when you call the function.

```
f1 <- function() 1
f2 <- function() invisible(1)

f1()
#> [1] 1
f2()
f1() == 1
#> [1] TRUE
f2() == 1
#> [1] TRUE
```

You can force an invisible value to be displayed by wrapping it in parentheses:

```
(f2())
#> [1] 1
```

The most common function that returns invisibly is `<-`:

```
a <- 2
(a <- 2)
#> [1] 2
```

This is what makes it possible to assign one value to multiple variables:

```
a <- b <- c <- d <- 2
```

because this is parsed as:

```
(a <- (b <- (c <- (d <- 2))))
#> [1] 2
```

6.6.1 On exit

As well as returning a value, functions can set up other triggers to occur when the function is finished using `on.exit()`. This is often used as a way to guarantee that changes to the global state are restored when the function exits. The code in `on.exit()` is run regardless of how the function exits, whether with an explicit (early) return, an error, or simply reaching the end of the function body.

```
in_dir <- function(dir, code) {
  old <- setwd(dir)
  on.exit(setwd(old))

  force(code)
}
getwd()
#> [1] "/Users/hadley/Documents/adv-r/adv-r"
in_dir("~", getwd())
#> [1] "/Users/hadley"
```

The basic pattern is simple:

- We first set the directory to a new location, capturing the current location from the output of `setwd()`.
- We then use `on.exit()` to ensure that the working directory is returned to the previous value regardless of how the function exits.
- Finally, we explicitly force evaluation of the code. (We don't actually need `force()` here, but it makes it clear to readers what we're doing.)

Caution: If you're using multiple `on.exit()` calls within a function, make sure to set `add = TRUE`. Unfortunately, the default in `on.exit()` is `add = FALSE`, so that every time you run it, it overwrites existing exit expressions. Because of the way `on.exit()` is implemented, it's not possible to create a variant with `add = TRUE`, so you must be careful when using it.

6.6.2 Exercises

1. How does the `chdir` parameter of `source()` compare to `in_dir()`? Why might you prefer one approach to the other?

2. What function undoes the action of `library()`? How do you save and restore the values of `options()` and `par()`?
3. Write a function that opens a graphics device, runs the supplied code, and closes the graphics device (always, regardless of whether or not the plotting code worked).
4. We can use `on.exit()` to implement a simple version of `capture.output()`.

```
capture.output2 <- function(code) {
  temp <- tempfile()
  on.exit(file.remove(temp), add = TRUE)

  sink(temp)
  on.exit(sink(), add = TRUE)

  force(code)
  readLines(temp)
}
capture.output2(cat("a", "b", "c", sep = "\n"))
#> [1] "a" "b" "c"
```

   Compare `capture.output()` to `capture.output2()`. How do the functions differ? What features have I removed to make the key ideas easier to see? How have I rewritten the key ideas to be easier to understand?

6.7 Quiz answers

1. The three components of a function are its body, arguments, and environment.
2. `f1(1)()` returns 11.
3. You'd normally write it in infix style: `1 + (2 * 3)`.
4. Rewriting the call to `mean(c(1:10, NA), na.rm = TRUE)` is easier to understand.
5. No, it does not throw an error because the second argument is never used so it's never evaluated.
6. See Section 6.5.1 and Section 6.5.2.
7. You use `on.exit()`; see Section 6.6.1 for details.