CHAPTER 15

Functions

Introduction

One of the best ways to improve your reach as a data scientist is to write functions. Functions allow you to automate common tasks in a more powerful and general way than copying and pasting. Writing a function has three big advantages over using copy-and-paste:

- You can give a function an evocative name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste (i.e., updating a variable name in one place, but not in another).

Writing good functions is a lifetime journey. Even after using R for many years I still learn new techniques and better ways of approaching old problems. The goal of this chapter is not to teach you every esoteric detail of functions but to get you started with some pragmatic advice that you can apply immediately.

As well as practical advice for writing functions, this chapter also gives you some suggestions for how to style your code. Good code style is like correct punctuation. Youcanmanagewithoutit, but it sure makes things easier to read! As with styles of punctuation, there are many possible variations. Here we present the style we use in our code, but the most important thing is to be consistent.

Prerequisites

The focus of this chapter is on writing functions in base R, so you won't need any extra packages.

When Should You Write a Function?

You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e., you now have three copies of the same code). For example, take a look at this code. What does it do?

```
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

You might be able to puzzle out that this rescales each column to have a range from 0 to 1. But did you spot the mistake? I made an error when copying and pasting the code for `df$b`: I forgot to change an `a` to a `b`. Extracting repeated code out into a function is a good idea because it prevents you from making this type of mistake.

To write a function you need to first analyze the code. How many inputs does it have?

```
(df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
```

This code only has one input: `df$a`. (If you're surprised that `TRUE` is not an input, you can explore why in the following exercise.) To make the inputs more clear, it's a good idea to rewrite the code using temporary variables with general names. Here this code only requires a single numeric vector, so I'll call it `x`:

```
x <- df$a
(x - min(x, na.rm = TRUE)) /
```

```
(max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612
#>  [10] 0.601
```

There is some duplication in this code. We're computing the range of the data three times, but it makes sense to do it in one step:

```
rng <- range(x, na.rm = TRUE)
(x - rng[1]) / (rng[2] - rng[1])
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612
#>  [10] 0.601
```

Pulling out intermediate calculations into named variables is a good practice because it makes it more clear what the code is doing. Now that I've simplified the code, and checked that it still works, I can turn it into a function:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0, 5, 10))
#> [1] 0.0 0.5 1.0
```

There are three key steps to creating a new function:

1. You need to pick a *name* for the function. Here I've used `rescale01` because this function rescales a vector to lie between 0 and 1.
2. You list the inputs, or *arguments*, to the function inside `function`. Here we have just one argument. If we had more the call would look like `function(x, y, z)`.
3. You place the code you have developed in the *body* of the function, a `{` block that immediately follows `function(...)`.

Note the overall process: I only made the function after I'd figured out how to make it work with a simple input. It's easier to start with working code and turn it into a function; it's harder to create a function and then try to make it work.

At this point it's a good idea to check your function with a few different inputs:

```
rescale01(c(-10, 0, 10))
#> [1] 0.0 0.5 1.0
rescale01(c(1, 2, 3, NA, 5))
#> [1] 0.00 0.25 0.50   NA 1.00
```

As you write more and more functions you'll eventually want to convert these informal, interactive tests into formal, automated tests. That process is called unit testing. Unfortunately, it's beyond the scope of this book, but you can learn about it at *http://r-pkgs.had.co.nz/tests.html*.

We can simplify the original example now that we have a function:

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

Compared to the original, this code is easier to understand and we've eliminated one class of copy-and-paste errors. There is still quite a bit of duplication since we're doing the same thing to multiple columns. We'll learn how to eliminate that duplication in Chapter 17, once you've learned more about R's data structures in Chapter 16.

Another advantage of functions is that if our requirements change, we only need to make the change in one place. For example, we might discover that some of our variables include infinite values, and `rescale01()` fails:

```
x <- c(1:10, Inf)
rescale01(x)
#>  [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

Because we've extracted the code into a function, we only need to make the fix in one place:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(x)
#>  [1] 0.000 0.111 0.222 0.333 0.444 0.556 0.667 0.778 0.889
#>  [10] 1.000   Inf
```

This is an important part of the "do not repeat yourself" (or DRY) principle. The more repetition you have in your code, the more places you need to remember to update when things change (and they always do!), and the more likely you are to create bugs over time.

Exercises

1. Why is `TRUE` not a parameter to `rescale01()`? What would happen if `x` contained a single missing value, and `na.rm` was `FALSE`?
2. In the second variant of `rescale01()`, infinite values are left unchanged. Rewrite `rescale01()` so that `-Inf` is mapped to 0, and `Inf` is mapped to 1.
3. Practice turning the following code snippets into functions. Think about what each function does. What would you call it? How many arguments does it need? Can you rewrite it to be more expressive or less duplicative?

```
mean(is.na(x))

x / sum(x, na.rm = TRUE)

sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)
```

4. Follow *http://nicercode.github.io/intro/writing-functions.html* to write your own functions to compute the variance and skew of a numeric vector.
5. Write `both_na()`, a function that takes two vectors of the same length and returns the number of positions that have an `NA` in both vectors.
6. What do the following functions do? Why are they useful even though they are so short?

```
is_directory <- function(x) file.info(x)$isdir
is_readable <- function(x) file.access(x, 4) == 0
```

7. Read the complete lyrics to "Little Bunny Foo Foo." There's a lot of duplication in this song. Extend the initial piping example to re-create the complete song, and use functions to reduce the duplication.

Functions Are for Humans and Computers

It's important to remember that functions are not just for the computer, but are also for humans. R doesn't care what your function is called, or what comments it contains, but these are important for human readers. This section discusses some things that you should bear in mind when writing functions that humans can understand.

The name of a function is important. Ideally, the name of your function will be short, but clearly evoke what the function does. That's hard! But it's better to be clear than short, as RStudio's autocomplete makes it easy to type long names.

Generally, function names should be verbs, and arguments should be nouns. There are some exceptions: nouns are OK if the function computes a very well known noun (i.e., `mean()` is better than `compute_mean()`), or is accessing some property of an object (i.e., `coef()` is better than `get_coefficients()`). A good sign that a noun might be a better choice is if you're using a very broad verb like "get," "compute," "calculate," or "determine." Use your best judgment and don't be afraid to rename a function if you figure out a better name later:

```
# Too short
f()

# Not a verb, or descriptive
my_awesome_function()

# Long, but clear
impute_missing()
collapse_years()
```

If your function name is composed of multiple words, I recommend using "snake_case," where each lowercase word is separated by an underscore. camelCase is a popular alternative. It doesn't really matter which one you pick; the important thing is to be consistent: pick one or the other and stick with it. R itself is not very consistent, but there's nothing you can do about that. Make sure you don't fall into the same trap by making your code as consistent as possible:

```
# Never do this!
col_mins <- function(x, y) {}
rowMaxes <- function(y, x) {}
```

If you have a family of functions that do similar things, make sure they have consistent names and arguments. Use a common prefix to indicate that they are connected. That's better than a common suffix because autocomplete allows you to type the prefix and see all the members of the family:

```
# Good
input_select()
input_checkbox()
input_text()
```

```
# Not so good
select_input()
checkbox_input()
text_input()
```

A good example of this design is the **stringr** package: if you don't remember exactly which function you need, you can type `str_` and jog your memory.

Where possible, avoid overriding existing functions and variables. It's impossible to do in general because so many good names are already taken by other packages, but avoiding the most common names from base R will avoid confusion:

```
# Don't do this!
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

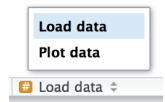
Use comments, lines starting with #, to explain the "why" of your code. You generally should avoid comments that explain the "what" or the "how." If you can't understand what the code does from reading it, you should think about how to rewrite it to be more clear. Do you need to add some intermediate variables with useful names? Do you need to break out a subcomponent of a large function so you can name it? However, your code can never capture the reasoning behind your decisions: why did you choose this approach instead of an alternative? What else did you try that didn't work? It's a great idea to capture that sort of thinking in a comment.

Another important use of comments is to break up your file into easily readable chunks. Use long lines of - or = to make it easy to spot the breaks:

```
# Load data --------------------------------------

# Plot data --------------------------------------
```

RStudio provides a keyboard shortcut to create these headers (Cmd/Ctrl-Shift-R), and will display them in the code navigation drop-down at the bottom-left of the editor:



Exercises

1. Read the source code for each of the following three functions, puzzle out what they do, and then brainstorm better names:

```
f1 <- function(string, prefix) {
  substr(string, 1, nchar(prefix)) == prefix
}
f2 <- function(x) {
  if (length(x) <= 1) return(NULL)
  x[-length(x)]
}
f3 <- function(x, y) {
  rep(y, length.out = length(x))
}
```

2. Take a function that you've written recently and spend five minutes brainstorming a better name for it and its arguments.
3. Compare and contrast `rnorm()` and `MASS::mvrnorm()`. How could you make them more consistent?
4. Make a case for why `norm_r()`, `norm_d()`, etc., would be better than `rnorm()`, `dnorm()`. Make a case for the opposite.

Conditional Execution

An `if` statement allows you to conditionally execute code. It looks like this:

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

To get help on `if` you need to surround it in backticks: `` ?`if` ``. The help isn't particularly helpful if you're not already an experienced programmer, but at least you know how to get to it!

Here's a simple function that uses an `if` statement. The goal of this function is to return a logical vector describing whether or not each element of a vector is named:

```
has_name <- function(x) {
  nms <- names(x)
  if (is.null(nms)) {
    rep(FALSE, length(x))
```

```
  } else {
    !is.na(nms) & nms != ""
  }
}
```

This function takes advantage of the standard return rule: a function returns the last value that it computed. Here that is either one of the two branches of the `if` statement.

Conditions

The `condition` must evaluate to either `TRUE` or `FALSE`. If it's a vector, you'll get a warning message; if it's an `NA`, you'll get an error. Watch out for these messages in your own code:

```
if (c(TRUE, FALSE)) {}
#> Warning in if (c(TRUE, FALSE)) {:
#> the condition has length > 1 and only the
#> first element will be used
#> NULL

if (NA) {}
#> Error in if (NA) {: missing value where TRUE/FALSE needed
```

You can use `||` (or) and `&&` (and) to combine multiple logical expressions. These operators are "short-circuiting": as soon as `||` sees the first `TRUE` it returns `TRUE` without computing anything else. As soon as `&&` sees the first `FALSE` it returns `FALSE`. You should never use `|` or `&` in an `if` statement: these are vectorized operations that apply to multiple values (that's why you use them in `filter()`). If you do have a logical vector, you can use `any()` or `all()` to collapse it to a single value.

Be careful when testing for equality. `==` is vectorized, which means that it's easy to get more than one output. Either check the length is already 1, collapse with `all()` or `any()`, or use the nonvectorized `identical()`. `identical()` is very strict: it always returns either a single `TRUE` or a single `FALSE`, and doesn't coerce types. This means that you need to be careful when comparing integers and doubles:

```
identical(0L, 0)
#> [1] FALSE
```

You also need to be wary of floating-point numbers:

```
x <- sqrt(2) ^ 2
x
#> [1] 2
```

```
x == 2
#> [1] FALSE
x - 2
#> [1] 4.44e-16
```

Instead use `dplyr::near()` for comparisons, as described in "Comparisons" on page 46.

And remember, `x == NA` doesn't do anything useful!

Multiple Conditions

You can chain multiple `if` statements together:

```
if (this) {
  # do that
} else if (that) {
  # do something else
} else {
  #
}
```

But if you end up with a very long series of chained `if` statements, you should consider rewriting. One useful technique is the `switch()` function. It allows you to evaluate selected code based on position or name:

```
#> function(x, y, op) {
#>   switch(op,
#>     plus = x + y,
#>     minus = x - y,
#>     times = x * y,
#>     divide = x / y,
#>     stop("Unknown op!")
#>   )
#> }
```

Another useful function that can often eliminate long chains of `if` statements is `cut()`. It's used to discretize continuous variables.

Code Style

Both `if` and `function` should (almost) always be followed by squiggly brackets (`{}`), and the contents should be indented by two spaces. This makes it easier to see the hierarchy in your code by skimming the lefthand margin.

An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should

always go on its own line, unless it's followed by `else`. Always indent the code inside curly braces:

```
# Good
if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}

# Bad
if (y < 0 && debug)
message("Y is negative")

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

It's OK to drop the curly braces if you have a very short `if` statement that can fit on one line:

```
y <- 10
x <- if (y < 20) "Too low" else "Too high"
```

I recommend this only for very brief `if` statements. Otherwise, the full form is easier to read:

```
if (y < 20) {
  x <- "Too low"
} else {
  x <- "Too high"
}
```

Exercises

1. What's the difference between `if` and `ifelse()`? Carefully read the help and construct three examples that illustrate the key differences.
2. Write a greeting function that says "good morning," "good afternoon," or "good evening," depending on the time of day. (Hint: use a time argument that defaults to `lubridate::now()`. That will make it easier to test your function.)

3. Implement a `fizzbuzz` function. It takes a single number as input. If the number is divisible by three, it returns "fizz". If it's divisible by five it returns "buzz". If it's divisible by three and five, it returns "fizzbuzz". Otherwise, it returns the number. Make sure you first write working code before you create the function.
4. How could you use `cut()` to simplify this set of nested if-else statements?

```
if (temp <= 0) {
  "freezing"
} else if (temp <= 10) {
  "cold"
} else if (temp <= 20) {
  "cool"
} else if (temp <= 30) {
  "warm"
} else {
  "hot"
}
```

   How would you change the call to `cut()` if I'd used `<` instead of `<=`? What is the other chief advantage of `cut()` for this problem? (Hint: what happens if you have many values in `temp`?)
5. What happens if you use `switch()` with numeric values?
6. What does this `switch()` call do? What happens if `x` is "e"?

```
switch(x,
  a = ,
  b = "ab",
  c = ,
  d = "cd"
)
```

   Experiment, then carefully read the documentation.

Function Arguments

The arguments to a function typically fall into two broad sets: one set supplies the *data* to compute on, and the other supplies arguments that control the *details* of the computation. For example:

- In `log()`, the data is `x`, and the detail is the `base` of the logarithm.

- In `mean()`, the data is `x`, and the details are how much data to trim from the ends (`trim`) and how to handle missing values (`na.rm`).
- In `t.test()`, the data are `x` and `y`, and the details of the test are `alternative`, `mu`, `paired`, `var.equal`, and `conf.level`.
- In `str_c()` you can supply any number of strings to `...`, and the details of the concatenation are controlled by `sep` and `collapse`.

Generally, data arguments should come first. Detail arguments should go on the end, and usually should have default values. You specify a default value in the same way you call a function with a named argument:

```
# Compute confidence interval around
# mean using normal approximation
mean_ci <- function(x, conf = 0.95) {
  se <- sd(x) / sqrt(length(x))
  alpha <- 1 - conf
  mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
}

x <- runif(100)
mean_ci(x)
#> [1] 0.498 0.610
mean_ci(x, conf = 0.99)
#> [1] 0.480 0.628
```

The default value should almost always be the most common value. The few exceptions to this rule have to do with safety. For example, it makes sense for `na.rm` to default to `FALSE` because missing values are important. Even though `na.rm = TRUE` is what you usually put in your code, it's a bad idea to silently ignore missing values by default.

When you call a function, you typically omit the names of the data arguments, because they are used so commonly. If you override the default value of a detail argument, you should use the full name:

```
# Good
mean(1:10, na.rm = TRUE)

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

You can refer to an argument by its unique prefix (e.g., `mean(x, n = TRUE)`), but this is generally best avoided given the possibilities for confusion.

Notice that when you call a function, you should place a space around `=` in function calls, and always put a space after a comma, not before (just like in regular English). Using whitespace makes it easier to skim the function for the important components:

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

Choosing Names

The names of the arguments are also important. R doesn't care, but the readers of your code (including future-you!) will. Generally you should prefer longer, more descriptive names, but there are a handful of very common, very short names. It's worth memorizing these:

- `x`, `y`, `z`: vectors.
- `w`: a vector of weights.
- `df`: a data frame.
- `i`, `j`: numeric indices (typically rows and columns).
- `n`: length, or number of rows.
- `p`: number of columns.

Otherwise, consider matching names of arguments in existing R functions. For example, use `na.rm` to determine if missing values should be removed.

Checking Values

As you start to write more functions, you'll eventually get to the point where you don't remember exactly how your function works. At this point it's easy to call your function with invalid inputs. To avoid this problem, it's often useful to make constraints explicit. For example, imagine you've written some functions for computing weighted summary statistics:

```
wt_mean <- function(x, w) {
  sum(x * w) / sum(x)
}
wt_var <- function(x, w) {
  mu <- wt_mean(x, w)
  sum(w * (x - mu) ^ 2) / sum(w)
}
wt_sd <- function(x, w) {
  sqrt(wt_var(x, w))
}
```

What happens if `x` and `w` are not the same length?

```
wt_mean(1:6, 1:3)
#> [1] 2.19
```

In this case, because of R's vector recycling rules, we don't get an error.

It's good practice to check important preconditions, and throw an error (with `stop()`) if they are not true:

```
wt_mean <- function(x, w) {
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }
  sum(w * x) / sum(x)
}
```

Be careful not to take this too far. There's a trade-off between how much time you spend making your function robust, versus how long you spend writing it. For example, if you also added a `na.rm` argument, I probably wouldn't check it carefully:

```
wt_mean <- function(x, w, na.rm = FALSE) {
  if (!is.logical(na.rm)) {
    stop("`na.rm` must be logical")
  }
  if (length(na.rm) != 1) {
    stop("`na.rm` must be length 1")
  }
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }

  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(x)
}
```

This is a lot of extra work for little additional gain. A useful compromise is the built-in `stopifnot()`; it checks that each argument is `TRUE`, and produces a generic error message if not:

```
wt_mean <- function(x, w, na.rm = FALSE) {
  stopifnot(is.logical(na.rm), length(na.rm) == 1)
  stopifnot(length(x) == length(w))

  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(x)
}
wt_mean(1:6, 6:1, na.rm = "foo")
#> Error: is.logical(na.rm) is not TRUE
```

Note that when using `stopifnot()` you assert what should be true rather than checking for what might be wrong.

Dot-Dot-Dot (...)

Many functions in R take an arbitrary number of inputs:

```
sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
#> [1] 55
stringr::str_c("a", "b", "c", "d", "e", "f")
#> [1] "abcdef"
```

How do these functions work? They rely on a special argument: `...` (pronounced dot-dot-dot). This special argument captures any number of arguments that aren't otherwise matched.

It's useful because you can then send those `...` on to another function. This is a useful catch-all if your function primarily wraps another function. For example, I commonly create these helper functions that wrap around `str_c()`:

```
commas <- function(...) stringr::str_c(..., collapse = ", ")
commas(letters[1:10])
#> [1] "a, b, c, d, e, f, g, h, i, j"

rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  cat(title, " ", stringr::str_dup(pad, width), "\n", sep = "")
}
rule("Important output")
#> Important output ----------------------------------------
```

Here `...` lets me forward on any arguments that I don't want to deal with to `str_c()`. It's a very convenient technique. But it does come at a price: any misspelled arguments will not raise an error. This makes it easy for typos to go unnoticed:

```
x <- c(1, 2)
sum(x, na.mr = TRUE)
#> [1] 4
```

If you just want to capture the values of the `...`, use `list(...)`.

Lazy Evaluation

Arguments in R are lazily evaluated: they're not computed until they're needed. That means if they're never used, they're never called. This is an important property of R as a programming language, but is generally not important when you're writing your own functions for data analysis. You can read more about lazy evaluation at *http://adv-r.had.co.nz/Functions.html#lazy-evaluation*.

Exercises

1. What does `commas(letters, collapse = "-")` do? Why?
2. It'd be nice if you could supply multiple characters to the `pad` argument, e.g., `rule("Title", pad = "-+")`. Why doesn't this currently work? How could you fix it?
3. What does the `trim` argument to `mean()` do? When might you use it?
4. The default value for the `method` argument to `cor()` is `c("pearson", "kendall", "spearman")`. What does that mean? What value is used by default?

Return Values

Figuring out what your function should return is usually straightforward: it's why you created the function in the first place! There are two things you should consider when returning a value:

- Does returning early make your function easier to read?
- Can you make your function pipeable?

Explicit Return Statements

The value returned by the function is usually the last statement it evaluates, but you can choose to return early by using `return()`. I think it's best to save the use of `return()` to signal that you can return early with a simpler solution. A common reason to do this is because the inputs are empty:

```
complicated_function <- function(x, y, z) {
  if (length(x) == 0 || length(y) == 0) {
    return(0)
  }

  # Complicated code here
}
```

Another reason is because you have a `if` statement with one complex block and one simple block. For example, you might write an `if` statement like this:

```
f <- function() {
  if (x) {
    # Do
    # something
    # that
    # takes
    # many
    # lines
    # to
    # express
  } else {
    # return something short
  }
}
```

But if the first block is very long, by the time you get to the `else`, you've forgotten the `condition`. One way to rewrite it is to use an early return for the simple case:

```
f <- function() {
  if (!x) {
    return(something_short)
  }

  # Do
  # something
  # that
  # takes
  # many
  # lines
```

```
  # to
  # express
}
```

This tends to make the code easier to understand, because you don't need quite so much context to understand it.

Writing Pipeable Functions

If you want to write your own pipeable functions, thinking about the return value is important. There are two main types of pipeable functions: transformation and side-effect.

In *transformation* functions, there's a clear "primary" object that is passed in as the first argument, and a modified version is returned by the function. For example, the key objects for **dplyr** and **tidyr** are data frames. If you can identify what the object type is for your domain, you'll find that your functions just work with the pipe.

Side-effect functions are primarily called to perform an action, like drawing a plot or saving a file, not transforming an object. These functions should "invisibly" return the first argument, so they're not printed by default, but can still be used in a pipeline. For example, this simple function prints out the number of missing values in a data frame:

```
show_missings <- function(df) {
  n <- sum(is.na(df))
  cat("Missing values: ", n, "\n", sep = "")

  invisible(df)
}
```

If we call it interactively, the `invisible()` means that the input `df` doesn't get printed out:

```
show_missings(mtcars)
#> Missing values: 0
```

But it's still there, it's just not printed by default:

```
x <- show_missings(mtcars)
#> Missing values: 0
class(x)
#> [1] "data.frame"
dim(x)
#> [1] 32 11
```

And we can still use it in a pipe:

```
mtcars %>%
  show_missings() %>%
  mutate(mpg = ifelse(mpg < 20, NA, mpg)) %>%
  show_missings()
#> Missing values: 0
#> Missing values: 18
```

Environment

The last component of a function is its environment. This is not something you need to understand deeply when you first start writing functions. However, it's important to know a little bit about environments because they are crucial to how functions work. The environment of a function controls how R finds the value associated with a name. For example, take this function:

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

In many programming languages, this would be an error, because `y` is not defined inside the function. In R, this is valid code because R uses rules called *lexical scoping* to find the value associated with a name. Since `y` is not defined inside the function, R will look in the *environment* where the function was defined:

```
y <- 100
f(10)
#> [1] 110

y <- 1000
f(10)
#> [1] 1010
```

This behavior seems like a recipe for bugs, and indeed you should avoid creating functions like this deliberately, but by and large it doesn't cause too many problems (especially if you regularly restart R to get to a clean slate).

The advantage of this behavior is that from a language standpoint it allows R to be very consistent. Every name is looked up using the same set of rules. For `f()` that includes the behavior of two things that you might not expect: `{` and `+`. This allows you to do devious things like:

```
`+` <- function(x, y) {
  if (runif(1) < 0.1) {
    sum(x, y)
  } else {
    sum(x, y) * 1.1
  }
}
table(replicate(1000, 1 + 2))
#>
#>   3 3.3
#> 100 900
rm(`+`)
```

This is a common phenomenon in R. R places few limits on your power. You can do many things that you can't do in other programming languages. You can do many things that 99% of the time are extremely ill-advised (like overriding how addition works!). But this power and flexibility is what makes tools like **ggplot2** and **dplyr** possible. Learning how to make best use of this flexibility is beyond the scope of this book, but you can read about in *Advanced R*.