Functional Programming in R 4

Advanced Statistical Programming for Data Science, Analysis, and Finance

—

Second Edition

—

Thomas Mailund

Apress®

Functional Programming in R 4

Advanced Statistical Programming for Data Science, Analysis, and Finance

Second Edition

Thomas Mailund

Apress®

Functional Programming in R 4: Advanced Statistical Programming for Data Science, Analysis, and Finance

Thomas Mailund
Aarhus N, Denmark

ISBN-13 (pbk): 978-1-4842-9486-4 ISBN-13 (electronic): 978-1-4842-9487-1
https://doi.org/10.1007/978-1-4842-9487-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Melissa Duffy
Development Editor: James Markham
Editorial Project Manager: Mark Powers

Cover designed by eStudioCalamar

Cover image by Max van den Oetelaar on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Thomas Mailund is Senior Software Architect at Kvantify, a quantum computing company from Denmark. He has a background in math and computer science. He now works on developing algorithms for computational problems applicable for quantum computing. He previously worked at the Bioinformatics Research Centre, Aarhus University, on genetics and evolutionary studies, particularly comparative genomics, speciation, and gene flow between emerging species. He has published *Beginning Data Science in R* with Apress, as well as other books out there.

About the Technical Reviewer

Megan J. Hirni is currently pursuing her PhD at the University of Missouri-Columbia with a focus on applied statistics research. In addition to her love for R coding, Megan loves meeting new people and learning new topics in multifaceted fields.

Acknowledgments

I would like to thank Duncan Murdoch and the people on the R-help mailing list for helping me work out a kink in lazy evaluation in the trampoline example.

CHAPTER 1

Introduction

Welcome to *Functional Programming in R 4*. I wrote this book to have teaching material beyond the typical introductory level most textbooks on R have, where functions are simple constructions for wrapping up some reusable instructions that you can then call when you need those instructions run. In languages such as R, functions are more than this. They are objects in their own right that you can also treat as data, create and manipulate and pass around like other objects, and learning how to do this will make you a far more effective R programmer.

The R language is a multiparadigm language with elements from procedural programming, object-oriented programming, and functional programming. Procedural programming focuses on the instructions you want the computer to execute—add these numbers, put the result in this variable, loop through this list, etc. Object-oriented programming, on the other hand, focuses on what kind of data you manipulate, which operations you can perform on them, and how they change when you manipulate them. If you are interested in these aspects of the R language, I have written another book, *Advanced Object-Oriented Programming in R*, also by Apress, that you might be interested in.

Functional programming is the third style of programming, where the focus is on transformations. Functions transform data from input to output, and by composing transformations, you construct programs from simpler functions to more involved pipelines for your data. In functional programming, functions themselves are considered data, and just as with other data, you can write transformations that take functions as input and

T. Mailund, *Functional Programming in R 4*, https://doi.org/10.1007/978-1-4842-9487-1_1

produce (other) functions as output. You can thus write simple functions, then adapt them (using other functions to modify them), and combine them in various ways to construct complete programs.

The R programming language supports procedural programming, object-oriented programming, and functional programming, but it is mainly a functional language. It is not a "pure" functional language. Pure functional languages will not allow you to modify the state of the program by changing values parameters hold and will not allow functions to have side effects (and need various tricks to deal with program input and output because of it).

R is somewhat close to "pure" functional languages. In general, data is immutable, so changes to data inside a function do ordinarily not alter the state of data outside that function. But R does allow side effects, such as printing data or making plots, and, of course, allows variables to change values.

Pure functions have no side effects, so a function called with the same input will always return the same output. Pure functions are easier to debug and to reason about because of this. They can be reasoned about in isolation and will not depend on the context in which they are called. The R language does not guarantee that the functions you write are pure, but you can write most of your programs using only pure functions. By keeping your code mostly purely functional, you will write more robust code and code that is easier to modify when the need arises.

You will want to move the impure functions to a small subset of your program. These functions are typically those that need to sample random data or that produce output (either text or plots). If you know where your impure functions are, you know when to be extra careful with modifying code.

The next chapter contains a short introduction to functions in R. Some parts you might already know, and so feel free to skip ahead, but I give a detailed description of how functions are defined and used to ensure that we are all on the same page. The following chapters then move on to more complex issues.

CHAPTER 2

Functions in R

In this chapter, we cover how to write functions in R. If you already know much of what is covered, feel free to skip ahead. We will discuss the way parameters are passed to functions as "promises," a way of passing parameters known as lazy evaluation. If you are not familiar with that but know how to write functions, you can jump forward to that section. We will also cover how to write infix operators and replacement functions, so if you do not know what those are, and how to write them, you can skip ahead to those sections. If you are new to R functions, continue reading from here.

Writing Functions in R

You create an R function using the `function` keyword or, since R 4.1, the `\()` syntax. For example, we can write a function that squares numbers like this:

```
square <- function(x) x**2
```

or like this:

```
square <- \(x) x**2
```

and then use it like this:

```
square(1:5)
```

```
## [1]  1  4  9  16  25
```


T. Mailund, *Functional Programming in R 4*, https://doi.org/10.1007/978-1-4842-9487-1_2

The shorter syntax, `\(x) x**2`, is intended for so-called "lambda expressions," and the backslash notation is supposed to look like the Greek letter lambda, λ. Lambda expressions are useful when we need to provide short functions as arguments to other functions, which is something we return to in later chapters. Usually, we use the `function()` syntax when defining reusable functions, and I will stick to this notation in every case where we define and name a function the way we did for `square` earlier.

The function we have written takes one argument, `x`, and returns the result `x**2`. The return value of a function is always the last expression evaluated in it. If you write a function with a single expression, you can write it as earlier, but for more complex functions, you will typically need several statements in it. If you do, you can put the function's body in curly brackets like this:

```
square <- function(x) {
    x**2
}
```

The following function needs the curly brackets since it needs three separate statements to compute its return value, one for computing the mean of its input, one for getting the standard deviation, and a final expression that returns the input scaled to be centered on the mean and having one standard deviation.

```
rescale <- function(x) {
    m <- mean(x)
    s <- sd(x)
     (x - m) / s
}
```

The first two statements are just there to define some variables we can use in the final expression. This is typical for writing short functions.

Variables you assign to inside a function will only be visible from inside the function. When the function completes its execution, the variables cease to exist. From inside a function, you can see the so-called *local* variables—the function arguments and the variables you assign to in the function body—and you can see the so-called *global* variables—those assigned to outside of the function. Outside of the function, however, you can only see the global variables. At least that is a good way to think about which variables you can see at any point in a program until we get to the gritty details in Chapter 4. For now, think in terms of global variables and local variables, where anything you write outside a function is in the first category and can be seen by anyone, and where function parameters and variables assigned to inside functions are in the second category; see Figure 2-1. If you have the same name for both a global and a local variable, as in the figure where we have a global variable x and a function parameter x, then the name always refers to the local variable.

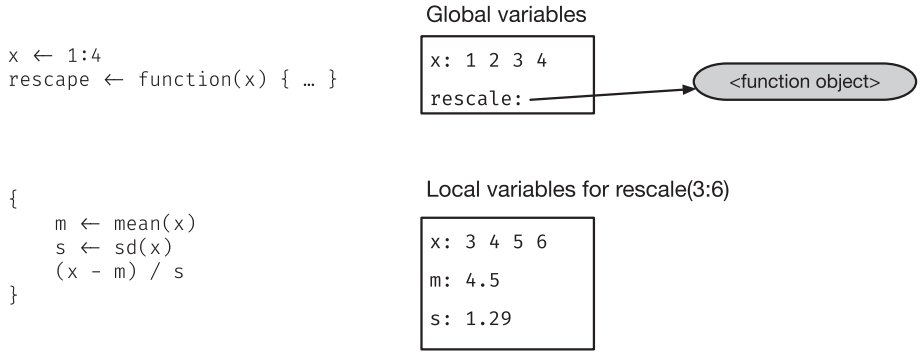

Figure 2-1. *Local and global variables*

Assignments are really also expressions. They return an object, the value that is being assigned; they just do so quietly. R considers some expressions "invisible," and while they do evaluate to some value or other—all expressions do—R does not print the result. Assignments are invisible in this way; they do return to the value on the right-hand side of the

assignment, but R makes the result invisible. You can remove this invisibility by putting an assignment in parentheses. The parentheses make R remove the invisibility of the expression result, so you see the actual value:

```
(x <- 1:5)
```

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

You can also go the other way and make a value invisible. When you evaluate an expression, R will print it:

```
x**2
```

```
## [1] 1 4 9 16 25
```

but if you put the expression in a call to `invisible`, R will not print the result:

```
invisible(x**2)
```

We usually use assignments for their side effect, assigning a name to a value, so you might not think of them as expressions, but everything you do in R is actually an expression. That includes control structures like `if`-statements and `for`-loops. They return values. They are actually functions themselves, and they return values. If you evaluate an `if`-statement, you get the value of the last expression in the branch it takes:

```
if (2 + 2 == 4) "Brave New World" else "1984"
```

```
## [1] "Brave New World"
```

If you evaluate a loop, you get the value `NULL` (and not the last expression in its body):

```
x <- for (i in 1:10) i
x
```

```
## NULL
```

Even parentheses and subscripting are functions. Parentheses evaluate to the value you put inside them (but stripped of invisibility), and subscripting, `[...]` or `[[...]]`, evaluates to some appropriate value on the data structure you are accessing (and you can define how this will work for your own data structures if you want to).

If you want to return a value from a function before its last expression, you can use the `return` function. It might look like a keyword, but it *is* a function, and you need to include the parentheses when you use it. Many languages will let you return a value by writing

```
return expression
```

Not R. In R, you need to write

```
return(expression)
```

If you are trying to return a value, this will not cause you much trouble. R will tell you that you have a syntax error if you use the former and not the latter syntax. Where it can be a problem is if you want to return from a function without providing a value (in which case the function automatically returns `NULL`).

If you write something like this:

```
f <- function(x) {
  if (x < 0) return;
  # Something else happens here...
}
```

you will not return if `x` is negative. The `if`-expression evaluates to the function `return`, which is not what you want. Instead, you must write

```
f <- function(x) {
  if (x < 0) return();
  # Something else happens here...
}
```