A Programmer's Guide to R

Serafin Schoch

June 16, 2024

Introduction

The programming language R can be both surprisingly convenient and irritating. This guide will highlight some of R's unconventional behavior and explain it. Additionally, I'll cover basic concepts from other programming languages and either show how to do it in Base R or reference a package that covers it.

R is a rather old language first released in 2000 and built upon the even older language S, both primarily used in academic contexts (see History of R). This means Base R is old, and many of beloved concepts don't exist in plain R or behave in unexpected ways. But don't worry, you don't need to relearn everything or miss out on these more modern functionalities. R comes with an ecosystem of well-crafted packages. Especially the "tidyverse" package — a collection of several smaller packages — fills the gaps between Base R and modern programming languages. These packages consolidate recent developments in design of programming languages into neat wrappers and functions. Seriously, if you don't use R with the appropriate packages, you're just using legacy code (see Best Practices for R).

This guide won't cover the R basics. I'll assume you know how to install packages and use them, or at least you know how to figure it out yourself. I'll try to build a solid fundamental understanding of R that one would not easily find by googling or using other search engines. Certainly, you can get an even better understanding by reading documentations. In contrast to the documentations, I'll provide you with an opinionated and concise selection of fundamentals.

Outline

I. Vectors and Piping: In this section, we explore why R, a functional programming language, lacks a map() function similar to Rust, Haskell, or Java, and why the length of "Hello World" is 1. We then demonstrate how the pipe operator in R can simplify function application and enhance code readability.

II. Subsetting and Indexing: Here we explore why accessing elements of a vector and a list are the same, although they look different on the surface. In the same go will look at accessing values by names and how to use "dictionaries" in R.

III. Dataframes: We'll explore the use of dataframes in R and the advantages of using the dplyr package for data manipulation. By comparing base R and dplyr syntax, we demonstrate how dplyr simplifies and enhances data operations.

IV. Functions and Methods: In this section, we explore the versatility of functions as objects, the utility of assertions for error handling within functions, and the implementation of class-specific methods using R's object-oriented features.

V. R Package and Rust: We'll have a look at creating R packages with a Rust backend. Rust integration is streamlined with rextendr. We'll use devtools to create and publish a package.

Some resources: CheatSheet collection, the same on GitHub and R Language Definition.

I. Vectors and Piping

First, we address two questions: Why doesn't a functional programming language like R have a 'map()' function like Rust, Haskell, or even Java? And why is the length of "Hello World" 1?

```
length("Hello World")

## [1] 1
```

Vectors

To answer the second question, we must understand that in R, everything is a vector. Basic types like strings, numerics, and booleans are vectors. There is no single string or boolean, only vectors of length 1 containing a string or boolean (see R Language Definition). Hence, the length of "Hello World" is 1 because it is a vector of size 1 containing the string "Hello World". We can get the number of characters in the string using the 'nchar()' function.

```
nchar("Hello World")

## [1] 11
```

To create a vector of several strings, we combine them with 'c()'. Knowing that the "Hello World" string is a vector and that we can apply 'nchar()' to it, it should be possible to apply 'nchar()' to this vector as well.

```
length(c("The","words","are"))

## [1] 3

nchar(c("The","words","are"))

## [1] 3 5 3
```

As expected, the length of the vector is 3, and the result of 'nchar()' is a vector of numerics containing the number of characters for each string.

Piping

Applying several functions in sequence can be cumbersome with base R syntax.

```
abs(nchar(c("The","words","are")) - 10)

## [1] 7 5 7
```

We can use the pipe operator from the 'magrittr' package, which is also loaded when using the 'dplyr' package, to make this more readable:

```
# library(dplyr)
c("The","words","are") %>%
    nchar(.) %>%
    {. - 10} %>%
    abs(.)

## [1] 7 5 7
```

Even though the code is longer, the order in which the functions and operations are applied is easier to understand. Notice how the syntax looks similar to 'c(...).map(nchar).map(...)'. This is because 'nchar', '-', and 'abs' are vectorized. Note that you could also use '%>% slice()' with the pipe operator.

Conclusion

Since everything in R is a vector, all basic functions are vectorized and work on vectors, making a 'map()' function in most cases unnecessary. Additionally, the pipe operator makes the code resemble the syntax of functional programming languages that use 'map()' and makes it easier to understand the order of applied functions. If you feel that the base R options don't resemble the iterator options of typical functional programming languages enough, have a look at the 'purrr' package.

II. Slicing and Names

Some of the most commonly used data structures are dictionaries and lists, but R doesn't have a dictionary type. Furthermore, lists in R behave peculiarly: Why does calling 'some_list[1]' return the first element wrapped in a list, while 'some_vec[1]' returns just the first element?

```
some_vec <- c("A","B","C")
identical("A", some_vec[1])

## [1] TRUE

some_list <- list("A","B","C")
identical("A", some_list[1])

## [1] FALSE
```

Slicing

To jump straight to the answer, the '[]' brackets are used for slicing. The question might be misleading since slicing a vector also returns a slice and not an element. However, all basic types in R are vectors. In that sense, the sliced vector containing only one string is as close as we get to a single string, meaning that "A" and 'some_vec[1]' are essentially the same. This does not hold for lists.

We can access the elements of a list with the '[[]]' brackets. We can also use this on vectors, but we'll receive just the same as we do by slicing.

```
some_list <- list("A","B","C")
identical("A", some_list[[1]])

## [1] TRUE
```

Slicing is quite versatile in R. Chapter 2.7 from the *R-intro* provides good insights into all possibilities of slicing. We can use it to retrieve a selected part of a vector/list or to update the selected values of a vector/list. Here are some examples in code:

```
vec <- c("A","B","C")
vec[c(TRUE, FALSE, TRUE)] # "A", "C"
vec["B" == vec] # returns "B"
vec[-2] # returns "A", "C"
vec[2:3] <- "X" # vec = "A", "X", "X"
```

Dictionaries

There is no dictionary type since every vector or list can have named entries. These names can be given on creation or later by assigning the attribute names. Names can be used to slice and access elements of vectors/lists.

```
vec <- c(1:3) # assign later
names(vec) <- c("A","B","C")
c(A=1,B=2,C=3) # assign on creation

## A B C
## 1 2 3

vec["B"]

## B
## 2
```

Lists additionally provide the '$name' dollar syntax, which does the same as '[["name"]]'.

```
list <- list(A=1, B=2, C=3)
identical(list[["A"]], list$A)

## [1] TRUE
```

Conclusion

There is no separate dictionary type since vectors and lists can be turned into dictionaries by naming them. The '$' dollar syntax is handy to access values of named lists. Furthermore, the '[]' brackets are used to slice and not to retrieve an element. Confusion can occur when using it on vectors since, in that case, a slice of size 1 is really the same as the element itself.

III. Dataframes

Dataframes are probably the most used data type in R, and this is where the 'dplyr' package becomes invaluable. Under the hood, dataframes are lists of lists (and vectors) with some additional constraints (e.g., all contained lists must have the same length). A tibble is a dataframe with some extra verifications and fewer automatic transformations, reducing unexpected mistakes. Additionally, it is the standard dataframe used throughout the tidyverse package.

Following, we will use the made-up data of some Momo characters to demonstrate filtering, selecting, summarizing, and grouping of dataframes. First, I'll show how to do it in base R, and instead of explaining, I'll just provide the syntax using the 'dplyr' package.

```
characters <- tibble(
  Name = c("Momo", "Beppo", "Gigi"),
  Age = c(12, 52, 27),
  Skill = c("Listening", "Steady Pace",
  "Storytelling"),
  Height = c(120, 180, 150),
)
characters

## # A tibble: 3 x 4
##   Name    Age Skill        Height
##   <chr> <dbl> <chr>         <dbl>
## 1 Momo     12 Listening       120
## 2 Beppo    52 Steady Pace     180
## 3 Gigi     27 Storytelling    150

# filter and select --------------------
tmp <- characters[ # base R
    characters$Age < 50,
    c("Name", "Skill")]
tmp[order(tmp$Name),]

## # A tibble: 2 x 2
##   Name  Skill
##   <chr> <chr>
## 1 Gigi  Storytelling
## 2 Momo  Listening

characters %>% # dplyr
  filter(Age < 50) %>%
  select(Name, Skill) %>%
  arrange(Name)

## # A tibble: 2 x 2
##   Name  Skill
##   <chr> <chr>
## 1 Gigi  Storytelling
## 2 Momo  Listening

# grouping and summarizing ------------
tapply(# base R
  characters$Height,
  cut(characters$Age, c(0, 50, 100)),
  mean
)

##   (0,50] (50,100]
##      135      180

characters %>% # dplyr
  group_by(age_cat =
    cut(Age, c(0,50,100))) %>%
  summarise(avg_height = mean(Height))

## # A tibble: 2 x 2
##   age_cat  avg_height
##   <fct>         <dbl>
## 1 (0,50]          135
## 2 (50,100]        180
```

Conclusion

Base R can do most of the things 'dplyr' can, but 'dplyr' syntax seems to explain itself. Moreover, the syntax facilitates thinking of more complex transformations. Imagine having data from different weather stations, and you want the newest measurement of each station. How would you do it in base R? With 'dplyr', you'll group by stations, arrange by date, and slice to yield the first element of each group. (Cheat Sheets: dplyr and check tidyr for pivot longer/wider)

IV. Functions and Methods

R, being a functional programming language, handles functions as objects and can even process them into lists and vice versa. While the usage of functions as objects is very useful, the conversion to lists is a legacy feature that can be happily ignored. Functions are one of the main ways to build and maintain larger coding projects. Usually, the most common functions already exist in Base R, and almost everything one would want is available in a package, reducing the need to write your own functions. Reusing functions can reduce bugs simply by using code that has already been tested by potentially hundreds of other people. Even if you build your own functions, they'll be tested by you, and after testing, you'll know they work. But imagine rewriting the code over and over again. The potential for a typo, misremembering a function name, or assigning a wrong variable is always there. Furthermore, well-designed functions make the code readable such that it no longer takes a lot of brain power to understand what is happening, even if the underlying procedure is very complex. Enough of the ramble about the benefits of clean code.

Functions

Good error messages are invaluable. They save potentially hours of debugging. I like to use assertions at the beginning of functions to verify whether the input is acceptable. This helps to catch errors as early as possible, and additionally, you can add messages that help to understand what should be different. For the sake of this section, we will write our own assertion function (similar to 'stopifnot()').

```
assert <- function(condition, message){
  if(exists("assert_off")&&assert_off)
    return("assertions off")
  if (!condition) {
    stop(message, call. = FALSE)
  }}
```

```
assert(FALSE, "something failed")

## Error:  something failed

assert_off <- TRUE
assert(FALSE, "something failed")

## [1] "assertions off"

rm(assert_off)
```

Here a lot is happening. First, we define the function 'assert', taking a condition and a message. The condition has to evaluate to true in order to pass the assertion. If not, the message is thrown. Additionally, we have a flag 'assert_off' which can be set to true to disable all assertions. This works because R looks for the variable 'assert_off' in the local environment of the function, but since it can't find it there, R goes up an environment until it finds the variable or the global scope is reached. If it can't find the variable, it would throw an error. Since we don't want that to happen, we check whether the variable 'assert_off' exists. If it doesn't exist, we assume assertions should be turned on. In practice, I would remove the flag, but it's a nice way to discuss environments. Here is how we can apply the 'assert' function.

```
# fromJSON has a nasty error message

readJSON <- function(path) {
  assert(
    file.exists(path),
    paste0("can't find \"", path,"\""))
  fromJSON(path)
}

readJSON("some_file.json")

## Error:  can't find "some_file.json"
```

Methods

In addition to functions, R has generics (methods) that have different implementations depending on the class of the first argument provided. Every R object has a class attribute (a vector of strings). You can manually modify the class attribute, potentially breaking the behavior of some functions. Classes provide the backbone for various operations, such as correctly handling additions for both normal and complex numbers, and even for adding visual elements when building a ggplot. Let's build our own class and generics.

```
worker <- 4
class(worker)

## [1] "numeric"

class(worker) <- "proletariat"
class(worker)

## [1] "proletariat"

mean(worker)

## [1] 4

mean.proletariat <- function(x)
  "doesn't work"
mean(worker)

## [1] "doesn't work"

"+.proletariat" <- function(a, b) {
  if (sum(a, b) >= 10) "protest" else
    sum(a, b) * 0.75
}
worker + worker

## [1] 6

worker + worker + worker

## [1] "protest"

future <- function(x)
  UseMethod("future")
future.proletariat <- function(x)
  "irrelevant"

future(worker)

## [1] "irrelevant"

future(4)

## Error in UseMethod("future"):
nicht anwendbare Methode für 'future'
auf Objekt der Klasse "c('double',
'numeric')" angewendet
```

Here, we first create a variable 'worker' which is a numeric vector (4). Then we assign the class 'proletariat' to the worker to define its class. We see 'worker' is now aware of its class. However, since we did not implement any methods for this class, it still behaves like a number. Next, we show how a worker should behave when we calculate the mean. We do this by adding the method 'mean.proletariat'. Now, whenever 'mean' is called on an object with the class 'proletariat', the string "doesn't work" is returned. Yay, the worker has learned: 'mean(worker)' doesn't work. We can even change the behavior of the '+' function. Notice we have to use '"+"' since '+' is a special symbol. This way, we can teach the proletariat what to do when multiple of them meet up — protest. For now, we added methods for already existing functions. If we want to create a new function that acts as a method, we use the 'UseMethod' function. This way, we create the function 'future' and define its behavior for the proletariat. Now we can also check what the future of a worker is: irrelevance. Since no default method is implemented for 'future' ('future.default <- ...'), the call to 'future(4)' throws an error.

Conclusions

We have learned how to throw errors and how environments resolve variables. Additionally, we explored class-specific functionalities and adapting basic functions like addition. These are valuable tools for creating more complex code. For further reading, refer to the R Language Definition.

V. R Package and Rust

A rule of thumb in programming is that if you want to do something, someone else has probably already done it better and documented it well. R does not natively support Rust, but it is fairly easy to build and run R packages with Rust code, or even build Rust functions in R and then run them in the R environment.

Rust from Within

The 'extendr' crate for Rust and the 'rextendr' package make it easy to run Rust code from within R. Check the rextendr documentation for installation and further examples. After installing all dependencies, mainly Rust, R, and the 'rextendr' package, you can run the following lines:

```
suppressMessages(
rextendr::rust_function("
fn add_from_rust(a:f64, b:f64) -> f64 {
    a + b
}"))

add_from_rust(1.2, 1.7)

## [1] 2.9
```

This is neat but not ideal, since calling 'rust_function()' compiles the provided Rust string. This takes a lot of time, and debugging the Rust code this way is not ideal. By building our own package, we can avoid this. Compiling and debugging will take place in a proper development environment.

Packages

Luckily, creating your own R package is supported by an amazing toolset. No wonder so many well-crafted packages exist. Check out the extendr and devtools documentation for detailed information.

After installing 'devtools' and 'rextendr', these commands will initialize an R package using Rust:

```
require(devtools)
create_package("path")
# Change directory to the package
use_readme_rmd() # Creating Rmd
rextendr::use_extendr() # Init
rextendr::document() # Update
```

The devtools cheat sheet is an excellent summary of how to create and publish a package to GitHub. From here on, everything should be smooth sailing.

I used the commands above to initialize my own package. Initially, it did not immediately work, but after running many commands in the devtools cheat sheet and restarting my R sessions, the following commands worked:

```
load_all()
check()
rextendr::document() # Update
# Push to GitHub
```

I have published my package on GitHub. Take a look at it here. You can install the package directly from GitHub with:

```
# devtools::install_github(
#     "S3r4f1n/rpackageUsingRust")
suppressMessages(library(
    rpackageUsingRust))
greeting_n_times(2)

## [1] "Hello world!\nHello world!\n"
```

Conclusion

Building an R package is not trivial, but the tooling (devtools) is quite easy to use and well documented. With the 'extendr' crate and 'rextendr' package, the interoperability between Rust and R is well supported, enabling the use of Rust as a backend for R packages. If you want to create your own R package using Rust, check the dedicated articles on extendr and devtools.