Advanced R (2ed): solutions manual

Martin Frigaard

2023-06-09

Table of contents

Why? Helper functions	. 4
I Foundations	7
Names & Values	8
Q:1 Q:2	
Vectors	14
Subsetting	15
Control flow	16
Choices Q1 Q2	
Q1	. 20
Functions	22
Environments	23
Conditions	24
II Functional programming	25
Functionals	26

Function factories	27
Function operators	28
III Object-oriented programming	29
Base types	30
S3	31
R6	32
S4	33
Trade-offs	34
IV Metaprogramming	35
Big picture	36
Expressions	37
Quasiquotation	38
Evalutation	39
Translating R code	40
Techniques	41
	40
V Debugging	42
Debugging	43
Measuring performance	44
Improving performance	45
Rewriting R code in C++	46

Why?

This is yet *another* version of the solutions to Advanced R, 2nd edition (advR2). I've returned to advR2 more than any other resource for my day-to-day activities as an R developer. After working through each of the exercises, then reading how others had solved them, I decided some the approaches I took to solving the exercises differed enough to warrant putting them into a book.

I highly recommend the Advanced R Solutions by Malte Grosser, Henning Bumann, and Hadley Wickham. I'll refer to these solutions in the following callout box:

```
♦ Answer: advRs 2021
```

Indrajeet Patil also has a solutions manual worth reading for alternative approaches. I'll refer to these solutions in the following callout box

```
⚠ Answer: advRs 2022
```

Helper functions

While reading advR2, I decided to write a few helper functions for returning various characteristics of R objects:

- 1. obj_info() combines base functions for class(), typeof(), attributes() along with functions from lobstr and sloop.
- obj_info() works with functions and function outputs:

```
obj_info(x = Sys.time())
# OBJECT: [Sys.time()]
# class/type: POSIXct/POSIXt/double
# attr name: class
# attr values: POSIXct, POSIXt
# address: 0x7fb4a98c3128
```

```
# function type: not a function
obj_info(x = Sys.time)
# OBJECT: [Sys.time]
# class/type: function/closure
# No attributes
# address: 0x7fb4a3465a60
# function type: internal
```

• obj_info() and replacement, assignment, arithmetic, and matching operators:

```
obj_info(x = [])
# OBJECT: [[]
# class/type: function/special
# No attributes
# address: 0x7fb49f0180f0
# function type: primitive, generic
obj_info(x = `<-`)
# OBJECT: [<-]
# class/type: function/special
# No attributes
# address: 0x7fb49f00ce08
# function type: primitive
obj_info(x = `-`)
# OBJECT: [-]
# class/type: function/builtin
# No attributes
# address: 0x7fb49f01a0b8
# function type: primitive, generic
obj_info(x = \int(x))
# OBJECT: [%in%]
# class/type: function/closure
# No attributes
# address: 0x7fb4a02790e0
# function type: function
```

• obj_info() and vectors:

```
obj_info(1:10)
# OBJECT: [1:10]
# class/type: integer
# No attributes
# address: 0x7fb4a22983a0
```

```
# function type: not a function
obj_info(mtcars$mpg)
# OBJECT: [mtcars$mpg]
# class/type: numeric/double
# No attributes
# address: 0x7fb4aa11e7c0
# function type: not a function
x \leftarrow factor(x = LETTERS[1:3],
         levels = LETTERS[1:3],
         labels = LETTERS[1:3])
obj_info(x = x)
# OBJECT: [x]
# class/type: factor/integer
# attr name: levels, class
# attr values: A, B, C, factor
# address: 0x7fb4a22ec0c8
# function type: not a function
```

Part I Foundations

Names & Values

I'll load the lobstr, waldo, and purrr packages for this chapter.

```
renv::install("lobstr")
renv::install("waldo")
renv::install("purrr")
renv::install("sloop")
library(lobstr)
library(waldo)
library(purrr)
library(sloop)
```

Exercises: Binding basics

Q:1

i Q:1 Explain the relationship between a, b, c and d in the following code: a <- 1:10 b <- a c <- b d <- 1:10

? A:1

a and d are vectors with the same values, but different memory locations. I use waldo::compare() because it has prettier printing for differences:

```
waldo::compare(
  x = lobstr::obj_addr(x = a),
  y = lobstr::obj_addr(x = b))
# v No differences
waldo::compare(
  x = lobstr::obj_addr(x = a),
  y = lobstr::obj_addr(x = c))
# v No differences
waldo::compare(
  x = lobstr::obj_addr(x = b),
  y = lobstr::obj_addr(x = c))
# v No differences
waldo::compare(
  x = lobstr::obj_addr(x = a),
  y = lobstr::obj_addr(x = d))
# `old`: "0x7ff67857dbf8"
# `new`: "0x7ff67b1b3d50"
```

Q:2

i Q:2

The following code accesses the mean function in multiple ways. Do they all point to the same underlying function object? Verify this with lobstr::obj_addr().

```
mean
base::mean
get("mean")
evalq(mean)
match.fun("mean")
```

9 A2

I used my get_info() function to verify these are all S3 generic functions from the base package, so they all share the same underlying function object.

```
obj_info(mean)
# OBJECT: [mean]
# class/type: function/closure
# No attributes
# address: 0x7ff68c5e4448
# function type: S3, generic
obj_info(base::mean)
# OBJECT: [base::mean]
# class/type: function/closure
# No attributes
# address: 0x7ff68c5e4448
# function type: S3, generic
obj_info(get("mean"))
# OBJECT: [get("mean")]
# class/type: function/closure
# No attributes
# address: 0x7ff68c5e4448
# function type: S3, generic
obj_info(evalq(mean))
# OBJECT: [evalq(mean)]
# class/type: function/closure
# No attributes
# address: 0x7ff68c5e4448
# function type: S3, generic
obj_info(match.fun("mean"))
# OBJECT: [match.fun("mean")]
# class/type: function/closure
# No attributes
# address: 0x7ff68c5e4448
# function type: S3, generic
```

Both solutions manuals solve this problem the same way (with list() and unique()):

\triangle Answer advRs 2021:

Yes, they point to the same object. We confirm this by inspecting the address of the underlying function object.

```
mean_functions <- list(</pre>
  mean,
  base::mean,
  get("mean"),
  evalq(mean),
  match.fun("mean")
unique(obj_addrs(mean_functions))
# [1] "0x7ff68c5e4448"
```

△ Answer advRs 2022 A:2

All listed function calls point to the same underlying function object in memory, as shown by this object's memory address:

```
obj_addrs <- obj_addrs(list(</pre>
  mean,
  base::mean,
  get("mean"),
  evalq(mean),
  match.fun("mean")
))
unique(obj_addrs)
# [1] "0x7ff68c5e4448"
```

i Q3

By default, base R data import functions, like read.csv(), will automatically convert non-syntactic names to syntactic ones. Why might this be problematic? What option allows you to suppress this behaviour?

i Q4

What rules does make.names() use to convert non-syntactic names into syntactic ones?

i Q5

I slightly simplified the rules that govern syntactic names. Why is .123e1 not a syntactic name? Read ?make.names for the full details.

Vectors

Subsetting

Control flow

I'll load the lobstr, waldo, and purrr packages for this chapter.

```
renv::install("lobstr")
renv::install("waldo")
renv::install("purrr")
renv::install("sloop")
library(lobstr)
library(waldo)
library(purrr)
library(sloop)
```

Choices

Q1

```
i Q:1

What type of vector does each of the following calls to ifelse() return?

ifelse(TRUE, 1, "no")
# [1] 1
ifelse(FALSE, 1, "no")
# [1] "no"
ifelse(NA, 1, "no")
# [1] NA
```

9 A1

ifelse() is strict when it comes to the type (or mode) in the results of tests, because it returns a "value with the same shape as test"

```
ifelse(test = TRUE, yes = 1, no = "no")
# [1] 1
mode(TRUE)
# [1] "logical"
```

This is essentially saying the test result is FALSE, so it evaluates to 'no'

```
# the 'shape' of the test here is FALSE, so 'no' is returned
ifelse(FALSE, 1, "no")
# [1] "no"
```

I can confirm this with isFALSE(FALSE):

```
# no = return values for false elements of test
ifelse(test = isFALSE(FALSE), yes = 1, no = "no")
# [1] 1

"Missing values in test give missing values in the result."

# the 'shape' of this test is missing, so it returns missing
ifelse(test = NA, 1, "no")
# [1] NA
# but this will work!
ifelse(test = is.na(NA), 1, "no")
# [1] 1
```

Q2

```
i Q:2

Why does the following code work?

x <- 1:10
  if (length(x)) "not empty" else "empty"

x <- numeric()
  if (length(x)) "not empty" else "empty"</pre>
```

```
A2
```

I've rewritten this to make the conditions a little easier to see

```
x <- 1:10
if (length(x)) {
    "not empty"
    } else {
    "empty"
    }
# [1] "not empty"</pre>
```

```
x <- numeric()
if (length(x)) {
    "not empty"
    } else {
    "empty"
    }
# [1] "empty"</pre>
```

The answer can be found by passing both statements to length(x) == 0, because at bottom, this is what logical statements contain:

```
sum(TRUE)
# [1] 1
sum(FALSE)
# [1] 0

x <- 1:10
length(x) == 0
# [1] FALSE

x <- numeric()
length(x) == 0
# [1] TRUE</pre>
```

So if the length of an empty numeric vector equals 0, it's not empty

Loops

Q1

```
i Q:1

Why does this code succeed without errors or warnings?

x <- numeric()
out <- vector("list", length(x))
for (i in 1:length(x)) {
  out[i] <- x[i]^2
}
out
# [[1]]
# [1] NA</pre>
```

Q2

i Q:2

When the following code is evaluated, what can you say about the vector being iterated?

```
xs <- c(1, 2, 3)
for (x in xs) {
   xs <- c(xs, x * 2)
}
xs
# [1] 1 2 3 2 4 6</pre>
```

Q3

i Q:3

 $What \ does \ the \ following \ code \ tell \ you \ about \ when \ the \ index \ is \ updated?$

Functions

Environments

Conditions

Part II Functional programming

Functionals

Function factories

Function operators

Part III Object-oriented programming

Base types

R6

S4

Trade-offs

Part IV Metaprogramming

Big picture

Expressions

Quasiquotation

Evalutation

Translating R code

Techniques

Part V Debugging

Debugging

Measuring performance

Improving performance

Rewriting R code in C++