Advanced R Exercises

Indrajeet Patil

2022-02-07

Contents

About

My solutions to exercises from Hadley Wickham's *Advanced R* book:
https://adv-r.hadley.nz/
For the official solution manual, see:
https://advanced-r-solutions.rbind.io/

Part I

Foundations

Chapter 1

Names and values

1.1 2.2.2 Exercises

Q1. Explain the relationship

```
a <- 1:10
b <- a
c <- b
d <- 1:10
```

All of these variable names are actively bound to the same value.

```
library(lobstr)

obj_addr(a)
#> [1] "0x14bc0998"
obj_addr(b)
#> [1] "0x14bc0998"
obj_addr(c)
#> [1] "0x14bc0998"
obj_addr(d)
#> [1] "0x16c543b8"
```

Q2. Function object address

Following code verifies that indeed these calls all point to the same underlying function object.

```
obj_addr(mean)
#> [1] "0x1653eaf0"
obj_addr(base::mean)
```

```
#> [1] "0x1653eaf0"
obj_addr(get("mean"))
#> [1] "0x1653eaf0"
obj_addr(evalq(mean))
#> [1] "0x1653eaf0"
obj_addr(match.fun("mean"))
#> [1] "0x1653eaf0"
```

Q3. Converting non-syntactic names

The conversion of non-syntactic names to syntactic ones can sometimes corrupt the data. Some datasets may require non-syntactic names.

To suppress this behavior, one can set `check.names = FALSE`.

Q4. Behavior of `make.names()`

It just prepends `X` in non-syntactic names and invalid characters (like `@`) are translated to `.`.

```
make.names(c("123abc", "@me", "_yu", "  gh", "else"))
#> [1] "X123abc" "X.me"    "X_yu"    "X..gh"   "else."
```

Q5. Why is `.123e1` not a syntactic name?

Because it is parsed as a number.

```
.123e1 < 1
#> [1] FALSE
```

1.2 2.3.6 Exercises

Q1. Usefulness of `tracemem()`

`tracemem()` traces copying of objects in R, but since the object created here is not assigned a name, there is nothing to trace.

```
tracemem(1:10)
#> [1] "<0000000017B2C718>"
```

Q2. Why two copies when you run this code?

Were it not for `4` being a double - and not an integer (`4L`) - this would have been modified in place.

```
x <- c(1L, 2L, 3L)
tracemem(x)
```

```
#> [1] "<000000001574ED28>"

x[[3]] <- 4
#> tracemem[0x000000001574ed28 -> 0x0000000014b0a280]: eval eval withVisible withCallingHandlers
#> tracemem[0x0000000014b0a280 -> 0x00000000158f3010]: eval eval withVisible withCallingHandlers
```

Try with integer:

```
x <- c(1L, 2L, 3L)
tracemem(x)
#> [1] "<0000000017F0F508>"

x[[3]] <- 4L
#> tracemem[0x0000000017f0f508 -> 0x0000000014988af8]: eval eval withVisible withCallingHandlers
```

As for why this still produces a copy, this is from Solutions manual:

> Please be aware that running this code in RStudio will result in additional copies because of the reference from the environment pane.

Q3. Study relationship

```
a <- 1:10
b <- list(a, a)
c <- list(b, a, 1:10)

ref(a)
#> [1:0x1773a8d8] <int>

ref(b)
#> o [1:0x1dccc878] <list>
#> +-[2:0x1773a8d8] <int>
#> \-[2:0x1773a8d8]

ref(c)
#> o [1:0x1c0c7a98] <list>
#> +-o [2:0x1dccc878] <list>
#> | +-[3:0x1773a8d8] <int>
#> | \-[3:0x1773a8d8]
#> +-[3:0x1773a8d8]
#> \-[4:0x17a33dc0] <int>
```

Q4. List inside another list

```
x <- list(1:10)
x
```

```
#> [[1]]
#>  [1]  1  2  3  4  5  6  7  8  9 10
obj_addr(x)
#> [1] "0x226a7e20"

x[[2]] <- x
x
#> [[1]]
#>  [1]  1  2  3  4  5  6  7  8  9 10
#>
#> [[2]]
#> [[2]][[1]]
#>  [1]  1  2  3  4  5  6  7  8  9 10
obj_addr(x)
#> [1] "0x21d43c38"

ref(x)
#> o [1:0x21d43c38] <list>
#> +-[2:0x16c1c4c8] <int>
#> \-o [3:0x226a7e20] <list>
#>   \-[2:0x16c1c4c8]
```

Figure here: https://advanced-r-solutions.rbind.io/images/names_values/copy_on_modify_fig2.png

1.3 2.4.1 Exercises

Q1. Object size difference between {base} and {lobstr}

This function...does not detect if elements of a list are shared.

```
y <- rep(list(runif(1e4)), 100)

object.size(y)
#> 8005648 bytes

obj_size(y)
#> 80,896 B
```

Q2. Misleading object size

These functions are not externally created objects in R, but are always available, so doesn't make much sense to measure their size.

```
funs <- list(mean, sd, var)
obj_size(funs)
```

```
#> 17,608 B
```

Nevertheless, it's still interesting that the addition is not the same as size of list of those objects.

```
obj_size(mean)
#> 1,184 B
obj_size(sd)
#> 4,480 B
obj_size(var)
#> 12,472 B

obj_size(mean) + obj_size(sd) + obj_size(var)
#> 18,136 B
```

Q3. Predict object sizes

```
a <- runif(1e6)
obj_size(a)
#> 8,000,048 B

b <- list(a, a)
obj_size(b)
#> 8,000,112 B
obj_size(a, b)
#> 8,000,112 B

b[[1]][[1]] <- 10
obj_size(b)
#> 16,000,160 B
obj_size(a, b)
#> 16,000,160 B

b[[2]][[1]] <- 10
obj_size(b)
#> 16,000,160 B
obj_size(a, b)
#> 24,000,208 B
```

1.4 2.5.3 Exercises

Q1. Why not a circular list?

Copy-on-modify prevents the creation of a circular list.

```
x <- list()

obj_addr(x)
#> [1] "0x23e8d308"

tracemem(x)
#> [1] "<0000000023E8D308>"

x[[1]] <- x
#> tracemem[0x0000000023e8d308 -> 0x0000000023fbe568]: eval eval withVisible withCalli

obj_addr(x[[1]])
#> [1] "0x23e8d308"
```

Q2. Why are loops so slow

```
library(bench)
```

Q3. `tracemem()` on an environment

It doesn't work and the documentation makes it clear as to why:

> It is not useful to trace NULL, environments, promises, weak references, or external pointer objects, as these are not duplicated

```
e <- rlang::env(a = 1, b = "3")
tracemem(e)
#> Error in tracemem(e): 'tracemem' is not useful for promise and environment objects
```

Chapter 2

Vectors

2.1 Exercise 3.2.5

Q1. Create raw and complex scalars

The raw type holds raw bytes. For example,

```
x <- "A string"

(y <- charToRaw(x))
#> [1] 41 20 73 74 72 69 6e 67

typeof(y)
#> [1] "raw"
```

You can use it to also figure out some encoding issues (both of these are scalars):

```
charToRaw("\"")
#> [1] 22
charToRaw("'")
#> [1] 94
```

Complex vectors can be used to represent (surprise!) complex numbers.

Example of a complex scalar:

```
(x <- complex(length.out = 1, real = 1, imaginary = 8))
#> [1] 1+8i

typeof(x)
#> [1] "complex"
```

Q2. Vector coercion rules

Usually, the more *general* type would take precedence.

```
c(1, FALSE)
#> [1] 1 0

c("a", 1)
#> [1] "a" "1"

c(TRUE, 1L)
#> [1] 1 1
```

Let's try some more examples.

```
c(1.0, 1L)
#> [1] 1 1

c(1.0, "1.0")
#> [1] "1"   "1.0"

c(TRUE, "1.0")
#> [1] "TRUE" "1.0"
```

Q3. Comparisons between different types

The coercion in vectors reveal why some of these comparisons return the results that they do.

```
1 == "1"
#> [1] TRUE

c(1, "1")
#> [1] "1" "1"
```

```
-1 < FALSE
#> [1] TRUE

c(-1, FALSE)
#> [1] -1  0
```

```
"one" < 2
#> [1] FALSE

c("one", 2)
#> [1] "one" "2"

sort(c("one", 2))
#> [1] "2"   "one"
```

Q4. Why `NA` defaults to `"logical"` type

The `"logical"` type is the lowest in the coercion hierarchy.

So `NA` defaulting to any other type (e.g. `"numeric"`) would mean that any time there is a missing element in a vector, rest of the elements would be converted to a type higher in hierarchy, which would be problematic for types lower in hierarchy.

```
typeof(NA)
#> [1] "logical"

c(FALSE, NA_character_)
#> [1] "FALSE" NA
```

Q5. Misleading variants of `is.*` functions

- `is.atomic()`:
- `is.numeric()`:
- `is.vector()`:

2.2 Exercise 3.3.4

Q1. Reading source code

```
setNames
#> function (object = nm, nm)
#> {
#>     names(object) <- nm
#>     object
#> }
#> <bytecode: 0x0000000001798f520>
#> <environment: namespace:stats>

setNames(c(1, 2), c("a", "b"))
#> a b
#> 1 2
```

```
unname
#> function (obj, force = FALSE)
#> {
#>     if (!is.null(names(obj)))
#>         names(obj) <- NULL
#>     if (!is.null(dimnames(obj)) && (force || !is.data.frame(obj)))
#>         dimnames(obj) <- NULL
#>     obj
#> }
```

```
#> <bytecode: 0x0000000014b9e8e8>
#> <environment: namespace:base>

A <- provideDimnames(N <- array(1:24, dim = 2:4))

unname(A, force = TRUE)
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
#> [2,]    8   10   12
#>
#> , , 3
#>
#>      [,1] [,2] [,3]
#> [1,]   13   15   17
#> [2,]   14   16   18
#>
#> , , 4
#>
#>      [,1] [,2] [,3]
#> [1,]   19   21   23
#> [2,]   20   22   24
```

Q2. 1-dimensional vector

Dimensions for a 1-dimensional vector are `NULL`.

`NROW()` and `NCOL()` are helpful for getting dimensions for 1D vectors by treating them as if they were a data frame vectors.

```
x <- character(0)

dim(x)
#> NULL

nrow(x)
#> NULL
NROW(x)
#> [1] 0
```

```
ncol(x)
#> NULL
NCOL(x)
#> [1] 1
```

Q3. Difference between vectors and arrays

`1:5` is a 1D vector without dimensions, while `x1`, `x2`, and `x3` are one-dimensional arrays.

```
1:5
#> [1] 1 2 3 4 5
(x1 <- array(1:5, c(1, 1, 5)))
#> , , 1
#>
#>      [,1]
#> [1,]    1
#>
#> , , 2
#>
#>      [,1]
#> [1,]    2
#>
#> , , 3
#>
#>      [,1]
#> [1,]    3
#>
#> , , 4
#>
#>      [,1]
#> [1,]    4
#>
#> , , 5
#>
#>      [,1]
#> [1,]    5
(x2 <- array(1:5, c(1, 5, 1)))
#> , , 1
#>
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    2    3    4    5
(x3 <- array(1:5, c(5, 1, 1)))
#> , , 1
#>
```

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

Q4. About `structure()`

From `?attributes` (emphasis mine):

> Note that some attributes (namely class, **comment**, dim, dimnames, names, row.names and tsp) are treated specially and have restrictions on the values which can be set.

```
structure(1:5, x = "my attribute")
#> [1] 1 2 3 4 5
#> attr(,"x")
#> [1] "my attribute"

structure(1:5, comment = "my attribute")
#> [1] 1 2 3 4 5
```

2.3 Exercise 3.4.5

Q1. `table()` function

`table()` returns an array with integer type and its dimensions scale with the number of variables present.

```
(x <- table(mtcars$am))
#>
#>  0  1
#> 19 13
(y <- table(mtcars$am, mtcars$cyl))
#>
#>      4  6  8
#>   0  3  4 12
#>   1  8  3  2
(z <- table(mtcars$am, mtcars$cyl, mtcars$vs))
#> , ,  = 0
#>
#>
#>      4  6  8
#>   0  0  0 12
#>   1  1  3  2
```

```
#>
#> , ,  = 1
#>
#>
#>      4  6  8
#>   0  3  4  0
#>   1  7  0  0

# type
purrr::map(list(x, y, z), typeof)
#> [[1]]
#> [1] "integer"
#>
#> [[2]]
#> [1] "integer"
#>
#> [[3]]
#> [1] "integer"

# attributes
purrr::map(list(x, y, z), attributes)
#> [[1]]
#> [[1]]$dim
#> [1] 2
#>
#> [[1]]$dimnames
#> [[1]]$dimnames[[1]]
#> [1] "0" "1"
#>
#>
#> [[1]]$class
#> [1] "table"
#>
#>
#> [[2]]
#> [[2]]$dim
#> [1] 2 3
#>
#> [[2]]$dimnames
#> [[2]]$dimnames[[1]]
#> [1] "0" "1"
#>
#> [[2]]$dimnames[[2]]
#> [1] "4" "6" "8"
#>
```

```
#>
#> [[2]]$class
#> [1] "table"
#>
#>
#> [[3]]
#> [[3]]$dim
#> [1] 2 3 2
#>
#> [[3]]$dimnames
#> [[3]]$dimnames[[1]]
#> [1] "0" "1"
#>
#> [[3]]$dimnames[[2]]
#> [1] "4" "6" "8"
#>
#> [[3]]$dimnames[[3]]
#> [1] "0" "1"
#>
#>
#> [[3]]$class
#> [1] "table"
```

Q2. Factor reversal

Its levels changes but the underlying integer values remain the same.

```
f1 <- factor(letters)
f1
#>  [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
#> 26 Levels: a b c d e f g h i j k l m n o p q r s t u ... z
as.integer(f1)
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
#> [19] 19 20 21 22 23 24 25 26

levels(f1) <- rev(levels(f1))
f1
#>  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
#> 26 Levels: z y x w v u t s r q p o n m l k j i h g f ... a
as.integer(f1)
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
#> [19] 19 20 21 22 23 24 25 26
```

Q3. Factor reversal-2

`f2`: Only the underlying integers are reversed, but levels remain unchanged. `f3`: Both the levels and the underlying integers are reversed.

```
f2 <- rev(factor(letters))
f2
#>  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
#> 26 Levels: a b c d e f g h i j k l m n o p q r s t u ... z
as.integer(f2)
#>  [1] 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9
#> [19]  8  7  6  5  4  3  2  1

f3 <- factor(letters, levels = rev(letters))
f3
#>  [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
#> 26 Levels: z y x w v u t s r q p o n m l k j i h g f ... a
as.integer(f3)
#>  [1] 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9
#> [19]  8  7  6  5  4  3  2  1
```

2.4 Exercise 3.5.4

Q1. Differences between list and atomic vector

feature	atomic vector	list (aka generic vector)
element type	unique	mixed[1]
recursive?	no	yes[2]
return for out-of-bounds index	`NA`[3]	`NULL`[4]
memory address	single memory reference[5]	reference per list element[6]

Q2. Converting a list to an atomic vector

List already *is* a vector, so `as.vector` is not going to change anything, and there is no `as.atomic.vector`. Thus the need to use `unlist()`.

```
x <- list(a = 1, b = 2)

is.vector(x)
#> [1] TRUE
is.atomic(x)
#> [1] FALSE
```

```
as.vector(x)
#> $a
#> [1] 1
#>
#> $b
#> [1] 2

unlist(x)
#> a b
#> 1 2
```

Q3. Comparing `c()` and `unlist()` for date and datetime

```
# creating a date and datetime
date <- as.Date("1947-08-15")
datetime <- as.POSIXct("1950-01-26 00:01", tz = "UTC")

# check attributes
attributes(date)
#> $class
#> [1] "Date"
attributes(datetime)
#> $class
#> [1] "POSIXct" "POSIXt"
#>
#> $tzone
#> [1] "UTC"

# check their underlying double representation
as.double(date) # number of days since the Unix epoch 1970-01-01
#> [1] -8175
as.double(datetime) # number of seconds since then
#> [1] -628991940
```

Behavior with `c()`: Works as expected. Only odd thing is that it strips the `tzone` attribute.

```
c(date, datetime)
#> [1] "1947-08-15" "1950-01-26"

attributes(c(date, datetime))
#> $class
#> [1] "Date"

c(datetime, date)
```

```
#> [1] "1950-01-26 01:01:00 CET"  "1947-08-15 02:00:00 CEST"

attributes(c(datetime, date))
#> $class
#> [1] "POSIXct" "POSIXt"
```

Behavior with `unlist()`: Removes all attributes and we are left only with the underlying double representations of these objects.

```
unlist(list(date, datetime))
#> [1]      -8175 -628991940

unlist(list(datetime, date))
#> [1] -628991940      -8175
```

2.5 Exercise 3.6.8

Q1. Data frame with 0 dimensions

Data frame with 0 rows is possible. This is basically a list with a vector of length 0.

```
data.frame(x = numeric(0))
#> [1] x
#> <0 rows> (or 0-length row.names)
```

Data frame with 0 columns is possible. This will be an empty list.

```
data.frame(row.names = 1)
#> data frame with 0 columns and 1 row
```

Both in one go:

```
data.frame()
#> data frame with 0 columns and 0 rows

dim(data.frame())
#> [1] 0 0
```

Q2. Non-unique rownames

If you attempt to set rownames that are not unique, it will not work.

```
data.frame(row.names = c(1, 1))
#> Error in data.frame(row.names = c(1, 1)): duplicate row.names: 1
```

Q3. Transposing dataframes

Transposing a dataframe transforms it into a matrix and coerces all its elements to be of the same type.

```
# original
(df <- head(iris))
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa

# transpose
t(df)
#>              1       2       3       4       5
#> Sepal.Length "5.1"   "4.9"   "4.7"   "4.6"   "5.0"
#> Sepal.Width  "3.5"   "3.0"   "3.2"   "3.1"   "3.6"
#> Petal.Length "1.4"   "1.4"   "1.3"   "1.5"   "1.4"
#> Petal.Width  "0.2"   "0.2"   "0.2"   "0.2"   "0.2"
#> Species      "setosa" "setosa" "setosa" "setosa" "setosa"
#>              6
#> Sepal.Length "5.4"
#> Sepal.Width  "3.9"
#> Petal.Length "1.7"
#> Petal.Width  "0.4"
#> Species      "setosa"

# transpose of a transpose
t(t(df))
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width
#> 1 "5.1"        "3.5"       "1.4"        "0.2"
#> 2 "4.9"        "3.0"       "1.4"        "0.2"
#> 3 "4.7"        "3.2"       "1.3"        "0.2"
#> 4 "4.6"        "3.1"       "1.5"        "0.2"
#> 5 "5.0"        "3.6"       "1.4"        "0.2"
#> 6 "5.4"        "3.9"       "1.7"        "0.4"
#>   Species
#> 1 "setosa"
#> 2 "setosa"
#> 3 "setosa"
#> 4 "setosa"
#> 5 "setosa"
#> 6 "setosa"
```

```
# is it a dataframe?
is.data.frame(df)
#> [1] TRUE
is.data.frame(t(df))
#> [1] FALSE
is.data.frame(t(t(df)))
#> [1] FALSE

# check type
typeof(df)
#> [1] "list"
typeof(t(df))
#> [1] "character"
typeof(t(t(df)))
#> [1] "character"

# check dimensions
dim(df)
#> [1] 6 5
dim(t(df))
#> [1] 5 6
dim(t(t(df)))
#> [1] 6 5
```

Q4. `as.matrix()` and dataframe

The return type of `as.matrix()` depends on dataframe column types.

```
# example with mixed types (coerced to character)
(df <- head(iris))
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa

as.matrix(df)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width
#> 1 "5.1"        "3.5"       "1.4"        "0.2"
#> 2 "4.9"        "3.0"       "1.4"        "0.2"
#> 3 "4.7"        "3.2"       "1.3"        "0.2"
#> 4 "4.6"        "3.1"       "1.5"        "0.2"
#> 5 "5.0"        "3.6"       "1.4"        "0.2"
```

```
#> 6 "5.4"        "3.9"       "1.7"        "0.4"
#>   Species
#> 1 "setosa"
#> 2 "setosa"
#> 3 "setosa"
#> 4 "setosa"
#> 5 "setosa"
#> 6 "setosa"

str(as.matrix(df))
#>  chr [1:6, 1:5] "5.1" "4.9" "4.7" "4.6" "5.0" "5.4" ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : chr [1:6] "1" "2" "3" "4" ...
#>   ..$ : chr [1:5] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" ...

# another example (no such coercion)
BOD
#>   Time demand
#> 1    1    8.3
#> 2    2   10.3
#> 3    3   19.0
#> 4    4   16.0
#> 5    5   15.6
#> 6    7   19.8

as.matrix(BOD)
#>      Time demand
#> [1,]    1    8.3
#> [2,]    2   10.3
#> [3,]    3   19.0
#> [4,]    4   16.0
#> [5,]    5   15.6
#> [6,]    7   19.8
```

From documentation of `data.matrix()`:

> Return the matrix obtained by converting all the variables in a data frame to numeric mode and then binding them together as the columns of a matrix.

So `data.matrix()` always returns a numeric matrix:

```
data.matrix(df)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2       1
#> 2          4.9         3.0          1.4         0.2       1
#> 3          4.7         3.2          1.3         0.2       1
```

```
#> 4          4.6         3.1          1.5         0.2       1
#> 5          5.0         3.6          1.4         0.2       1
#> 6          5.4         3.9          1.7         0.4       1

str(data.matrix(df))
#>  num [1:6, 1:5] 5.1 4.9 4.7 4.6 5 5.4 3.5 3 3.2 3.1 ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : chr [1:6] "1" "2" "3" "4" ...
#>   ..$ : chr [1:5] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" ...
```

Chapter 3

Control flow

Exercise 5.2.4

Q1. `ifelse()` return type

It's type unstable, i.e. the type of return will depend on the type of each condition (`yes` and `no`, i.e.), and works only for cases where `test` argument evaluated to a `logical` type.

```
ifelse(TRUE, 1, "no") # numeric returned
#> [1] 1
ifelse(FALSE, 1, "no") # character returned
#> [1] "no"
ifelse(NA, 1, "no")
#> [1] NA
```

Additionally, if the test doesn't resolve to `logical` type, it will try to coerce whatever the resulting type to `logical`:

```
# will work
ifelse("TRUE", 1, "no")
#> [1] 1
ifelse("true", 1, "no")
#> [1] 1

# won't work
ifelse("tRuE", 1, "no")
#> [1] NA
ifelse(NaN, 1, "no")
#> [1] NA
```

Q2. Why does the following code work?

As can be seen, the code works because the tests are successfully coerced to a logical type.

```
x <- 1:10
as.logical(length(1:10))
#> [1] TRUE
if (length(x)) "not empty" else "empty"
#> [1] "not empty"

x <- numeric()
as.logical(length(numeric()))
#> [1] FALSE
if (length(x)) "not empty" else "empty"
#> [1] "empty"
```

Exercise 5.3.3

Q1. Why does this work?

This works because `1:length(x)` goes both ways; in this case, from 1 to 0. And, since out-of-bound values for atomic vectors is `NA`, all related operations with it also lead to `NA`.

```
x <- numeric()
out <- vector("list", length(x))

for (i in 1:length(x)) {
  print(x[i])
  print(out[i])

  out[i] <- x[i]^2
}
#> [1] NA
#> [[1]]
#> NULL
#>
#> numeric(0)
#> list()

out
#> [[1]]
#> [1] NA
```

A way to do avoid this unintended behavior would be:

```
x <- numeric()
out <- vector("list", length(x))

for (i in 1:seq_along(x)) {
  out[i] <- x[i]^2
}
#> Error in 1:seq_along(x): argument of length 0

out
#> list()
```

Q2. Index vectors

Surprisingly (at least to me), `x` takes all values of the vector `xs`:

```
# xs <- c(1, 2, 3)
xs <- c(4, 5, 6)
for (x in xs) {
  print(x)
  xs <- c(xs, x * 2)
}
#> [1] 4
#> [1] 5
#> [1] 6

xs
#> [1]  4  5  6  8 10 12
```

Q3. Index increase

In a `for` loop - like in a `while` loop - the index is updated in the beginning of each iteration.

```
for (i in 1:3) {
  cat("before: ", i, "\n")
  i <- i * 2
  cat("after: ", i, "\n")
}
#> before:  1
#> after:  2
#> before:  2
#> after:  4
#> before:  3
#> after:  6

i <- 1
```

```
while (i < 3) {
  cat("before: ", i, "\n")
  i <- i * 2
  cat("after: ", i, "\n")
}
#> before:  1
#> after:  2
#> before:  2
#> after:  4
```

Chapter 4

Functionals

4.1 Exercise 9.2.6

Q1. Study `as_mapper()`

```
library(purrr)

# mapping by position ----------------------

x <- list(1, list(2, 3, list(1, 2)))

map(x, 1)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
as_mapper(1)
#> function (x, ...)
#> pluck(x, 1, .default = NULL)
#> <environment: 0x0000000016212c60>

map(x, list(2, 1))
#> [[1]]
#> NULL
#>
#> [[2]]
#> [1] 3
as_mapper(list(2, 1))
#> function (x, ...)
```

```
#> pluck(x, 2, 1, .default = NULL)
#> <environment: 0x0000000015688618>

# mapping by name ------------------------

y <- list(
  list(m = "a", list(1, m = "mo")),
  list(n = "b", list(2, n = "no"))
)

map(y, "m")
#> [[1]]
#> [1] "a"
#>
#> [[2]]
#> NULL
as_mapper("m")
#> function (x, ...)
#> pluck(x, "m", .default = NULL)
#> <environment: 0x0000000014b1dfb8>

# mixing position and name
map(y, list(2, "m"))
#> [[1]]
#> [1] "mo"
#>
#> [[2]]
#> NULL
as_mapper(list(2, "m"))
#> function (x, ...)
#> pluck(x, 2, "m", .default = NULL)
#> <environment: 0x00000000153d1158>

# compact functions ----------------------------

map(y, ~ length(.x))
#> [[1]]
#> [1] 2
#>
#> [[2]]
#> [1] 2
as_mapper(~ length(.x))
#> <lambda>
#> function (..., .x = ..1, .y = ..2, . = ..1)
#> length(.x)
```

```
#> attr(,"class")
#> [1] "rlang_lambda_function" "function"
```

You can extract attributes using `purrr::attr_getter()`:

```
pluck(Titanic, attr_getter("class"))
#> [1] "table"
```

Q2. Properly specifying anonymous functions

As shown by `as_mapper()` outputs below, the second call is not appropriate for generating random numbers because it translates to `pluck()` function where the indices for plucking are taken to be randomly generated numbers.

```
library(purrr)

map(1:3, ~ runif(2))
#> [[1]]
#> [1] 0.08755097 0.40734483
#>
#> [[2]]
#> [1] 0.41967533 0.01231029
#>
#> [[3]]
#> [1] 0.4299795 0.6542232
as_mapper(~ runif(2))
#> <lambda>
#> function (..., .x = ..1, .y = ..2, . = ..1)
#> runif(2)
#> attr(,"class")
#> [1] "rlang_lambda_function" "function"

map(1:3, runif(2))
#> [[1]]
#> NULL
#>
#> [[2]]
#> NULL
#>
#> [[3]]
#> NULL
as_mapper(runif(2))
#> function (x, ...)
#> pluck(x, 0.14191510155797, 0.703806154197082, .default = NULL)
#> <environment: 0x00000000179f50d8>
```

Q3. Use the appropriate `map()` function

Compute the standard deviation of every column in a numeric data frame.

```
map_dbl(mtcars, sd)
#>         mpg         cyl        disp          hp        drat
#>   6.0269481   1.7859216 123.9386938  68.5628685   0.5346787
#>          wt        qsec          vs          am        gear
#>   0.9784574   1.7869432   0.5040161   0.4989909   0.7378041
#>        carb
#>   1.6152000
```

Compute the standard deviation of every numeric column in a mixed data frame.

```
keep(iris, is.numeric) %>%
  map_dbl(sd)
#> Sepal.Length  Sepal.Width Petal.Length  Petal.Width
#>    0.8280661    0.4358663    1.7652982    0.7622377
```

Compute the number of levels for every factor in a data frame.

```
modify_if(dplyr::starwars, is.character, as.factor) %>%
  keep(is.factor) %>%
  map_int(~ length(levels(.)))
#>       name hair_color skin_color  eye_color        sex
#>         87         12         31         15          4
#>     gender  homeworld    species
#>          2         48         37
```

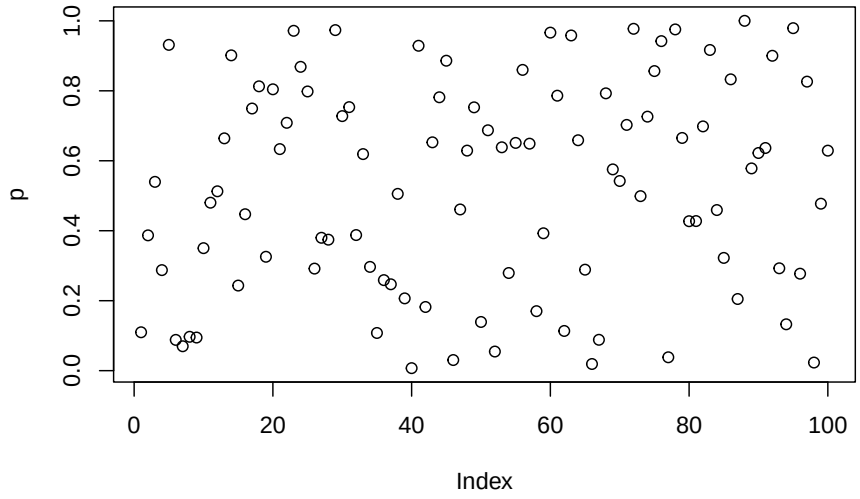
Q4. Statistics and visualization with {purrr}

Extract the p-value from each test, then visualise.

```
trials <- map(1:100, ~ t.test(rpois(10, 10), rpois(7, 10)))

p <- map_dbl(trials, "p.value")

plot(p)
```



Q5. Fixing non-functioning code

```
x <- list(
  list(1, c(3, 9)),
  list(c(3, 6), 7, c(4, 7, 6))
)

triple <- function(x) x * 3
map(x, .f = ~ map(., ~ triple(.)))
#> [[1]]
#> [[1]][[1]]
#> [1] 3
#>
#> [[1]][[2]]
#> [1]  9 27
#>
#>
#> [[2]]
#> [[2]][[1]]
#> [1]  9 18
#>
#> [[2]][[2]]
#> [1] 21
#>
```

```
#> [[2]][[3]]
#> [1] 12 21 18
```

Q6. Use `map()` to fit linear models to the `mtcars` dataset

```
formulas <- list(
  mpg ~ disp,
  mpg ~ I(1 / disp),
  mpg ~ disp + wt,
  mpg ~ I(1 / disp) + wt
)

map(formulas, ~ lm(formula = ., data = mtcars))
#> [[1]]
#>
#> Call:
#> lm(formula = ., data = mtcars)
#>
#> Coefficients:
#> (Intercept)         disp
#>    29.59985     -0.04122
#>
#>
#> [[2]]
#>
#> Call:
#> lm(formula = ., data = mtcars)
#>
#> Coefficients:
#> (Intercept)    I(1/disp)
#>       10.75      1557.67
#>
#>
#> [[3]]
#>
#> Call:
#> lm(formula = ., data = mtcars)
#>
#> Coefficients:
#> (Intercept)         disp           wt
#>    34.96055     -0.01772     -3.35083
#>
#>
#> [[4]]
```

```
#>
#> Call:
#> lm(formula = ., data = mtcars)
#>
#> Coefficients:
#> (Intercept)    I(1/disp)           wt
#>      19.024     1142.560       -1.798
```

Q7. Computing R-squared

```
bootstrap <- function(df) {
  df[sample(nrow(df), replace = TRUE), , drop = FALSE]
}

bootstraps <- map(1:10, ~ bootstrap(mtcars))

map_dbl(
  bootstraps,
  ~ summary(lm(formula = mpg ~ disp, data = .))$r.squared
)
#>  [1] 0.7467027 0.7839514 0.7263799 0.7319638 0.7652060
#>  [6] 0.7377944 0.7204561 0.7430256 0.8514938 0.7326552
```

4.2 Exercise 9.4.6

Q1. Explain the results

`modify()` returns the object of type same as the input. Since the input here is a dataframe of certain dimensions and `.f = 1` translates to plucking the first element in each column, it returns a dataframes with the same dimensions with the plucked element recycled across rows.

```
head(modify(mtcars, 1))
#>                   mpg cyl disp  hp drat   wt  qsec vs am
#> Mazda RX4          21   6  160 110  3.9 2.62 16.46  0  1
#> Mazda RX4 Wag      21   6  160 110  3.9 2.62 16.46  0  1
#> Datsun 710         21   6  160 110  3.9 2.62 16.46  0  1
#> Hornet 4 Drive     21   6  160 110  3.9 2.62 16.46  0  1
#> Hornet Sportabout  21   6  160 110  3.9 2.62 16.46  0  1
#> Valiant            21   6  160 110  3.9 2.62 16.46  0  1
#>                   gear carb
#> Mazda RX4            4    4
#> Mazda RX4 Wag        4    4
#> Datsun 710           4    4
```

```
#> Hornet 4 Drive       4    4
#> Hornet Sportabout    4    4
#> Valiant              4    4
```

Q2. Use `iwalk()` instead of `walk2()`

```
# with walk2() -----------------------

cyls <- split(mtcars, mtcars$cyl)
paths <- file.path(temp, paste0("cyl-", names(cyls), ".csv"))
walk2(.x = cyls, .y = paths, .f = write.csv)

# with iwalk --------------------------

cyls <- split(mtcars, mtcars$cyl)
names(cyls) <- file.path(temp, paste0("cyl-", names(cyls), ".csv"))
iwalk(cyls, ~ write.csv(.x, .y))
```

Q3. Explain the code

`map2()` supplies the functions defined in `.x = trans` as `f` in the anonymous functions, while the names of the columns defined in `.y = mtcars[nm]` are picked up by `var` in the anonymous function. Note that the function is iterating over indices for vectors of transformations and column names.

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) factor(x, labels = c("auto", "manual"))
)

nm <- names(trans)
mtcars[nm] <- map2(trans, mtcars[nm], function(f, var) f(var))
```

In the `map` approach, the function is iterating over indices for vectors of column names.

```
mtcars[nm] <- map(nm, ~ trans[[.x]](mtcars[[.x]]))
```

Q4. Difference between `map2()` and `walk2()`

If we use `map2()`, it will work, but it will print `NULL` to the terminal for every element of the list.

```
bods <- split(BOD, BOD$Time)
nm <- names(bods)
map2(bods, nm, write.csv)
```

4.3 Exercise 9.6.3

Q1. Predicate functions

> A predicate is a function that returns a **single** `TRUE` or `FALSE`.

The `is.na()` function does not return a single value, but instead returns a vector and thus isn't a predicate function.

```
# contrast the following behavior of predicate functions
is.character(c("x", 2))
#> [1] TRUE
is.null(c(3, NULL))
#> [1] FALSE

# with this behavior
is.na(c(NA, 1))
#> [1]  TRUE FALSE
```

The closest equivalent of a predicate function in base-R is `anyNA()` function.

```
anyNA(c(NA, 1))
#> [1] TRUE
```

Q2. Fix `simple_reduce`

Supplied function:

```
simple_reduce <- function(x, f) {
  out <- x[[1]]
  for (i in seq(2, length(x))) {
    out <- f(out, x[[i]])
  }
  out
}
```

Struggles with inputs of length 0 and 1 because function tries to access out-of-bound values.

```
simple_reduce(numeric(), sum)
#> Error in x[[1]]: subscript out of bounds
simple_reduce(1, sum)
#> Error in x[[i]]: subscript out of bounds
simple_reduce(1:3, sum)
#> [1] 6
```

This problem can be solved by adding `init` argument, which supplies the default or initial value for the function to operate on:

```
simple_reduce2 <- function(x, f, init = 0) {
  # initializer will become the first value
  if (length(x) == 0L) {
    return(init)
  }
  if (length(x) == 1L) {
    return(x[[1L]])
  }

  out <- x[[1]]

  for (i in seq(2, length(x))) {
    out <- f(out, x[[i]])
  }
  out
}
```

Let's try it out:

```
simple_reduce2(numeric(), sum)
#> [1] 0
simple_reduce2(1, sum)
#> [1] 1
simple_reduce2(1:3, sum)
#> [1] 6
```

With a different kind of function:

```
simple_reduce2(numeric(), `*`, init = 1)
#> [1] 1
simple_reduce2(1, `*`, init = 1)
#> [1] 1
simple_reduce2(1:3, `*`, init = 1)
#> [1] 6
```

And another one:

```
simple_reduce2(numeric(), `%/%`)
#> [1] 0
simple_reduce2(1, `%/%`)
#> [1] 1
simple_reduce2(1:3, `%/%`)
#> [1] 0
```

4.4 Exercise 9.7.3

Q1.

Q2. `eapply()` and `rapply()`

`eapply()` applies FUN to the named values from an environment and returns the results as a list.

`rapply()` is a recursive version of lapply with flexibility in how the result is structured (how = "..").

Chapter 5

S3

5.1 Exercise 13.2.1

Q1. Differences between `t.test` and `t.data.frame`

```
library(sloop)

# function type
ftype(t.test)
#> [1] "S3"      "generic"
ftype(t.data.frame)
#> [1] "S3"     "method"
```

- `t.test()` is a **generic** function to perform t-test.
- `t.data.frame` is a **method** for generic `t()` (a matrix transform function) and will be dispatched for `data.frame` objects that need to be transformed.

Q2. base-R function with `.`

- `all.equal()`
- Most of `as.*` functions like `as.data.frame()`
- `install.packages()` etc.

For example,

```
ftype(as.data.frame)
#> [1] "S3"      "generic"
```

Q3. What does `as.data.frame.data.frame()` do?

It's a **method** for generic `as.data.frame()`.

Less confusing: `asDataFrame.DataFrame()`.

Q4. Difference in behavior

Before unclassing, the S3 dispatches `.Date` method, while after `.numeric` method.

Before

```
some_days <- as.Date("2017-01-31") + sample(10, 5)

some_days
#> [1] "2017-02-09" "2017-02-03" "2017-02-04" "2017-02-01"
#> [5] "2017-02-10"

s3_dispatch(mean(some_days))
#> => mean.Date
#>  * mean.default

mean(some_days)
#> [1] "2017-02-05"
```

After

```
unclass(some_days)
#> [1] 17206 17200 17201 17198 17207

mean(unclass(some_days))
#> [1] 17202.4

s3_dispatch(mean(unclass(some_days)))
#>    mean.double
#>    mean.numeric
#> => mean.default
```

Q5. Object properties

```
x <- ecdf(rpois(100, 10))
x
#> Empirical CDF
#> Call: ecdf(rpois(100, 10))
#>  x[1:14] =      4,      5,      6,  ...,     16,     17

otype(x)
```

```
#> [1] "S3"

attributes(x)
#> $class
#> [1] "ecdf"     "stepfun"  "function"
#>
#> $call
#> ecdf(rpois(100, 10))

s3_class(x)
#> [1] "ecdf"     "stepfun"  "function"
```

Q6. Object properties

```
x <- table(rpois(100, 5))
x
#>
#>  0  1  2  3  4  5  6  7  8  9 10 11 12
#>  1  4 10 12 16 22  9 15  4  2  3  1  1

otype(x)
#> [1] "S3"

attributes(x)
#> $dim
#> [1] 13
#>
#> $dimnames
#> $dimnames[[1]]
#>  [1] "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
#> [12] "11" "12"
#>
#>
#> $class
#> [1] "table"

s3_class(x)
#> [1] "table"
```

Chapter 6

R6

6.1 Exercise 14.2.6

Q1. R6 class for bank account

Create the superclass and make sure it works as expected.

```
library(R6)

# define the needed class
bankAccount <- R6::R6Class(
  "bankAccount",
  public = list(
    # fields -----------------------
    balance = NA,
    name = NA,

    # methods ----------------------
    initialize = function(name = NULL, balance) {
      self$validate(balance)

      self$name <- name
      self$balance <- balance
    },
    deposit = function(amount) {
      self$validate(amount)
      cat("Current balance is: ", self$balance, "\n", sep = "")
      cat("And you are depositing: ", amount)
      self$balance <- self$balance + amount
      invisible(self)
    },
```

```
    withdraw = function(amount) {
      self$validate(amount)
      cat("Current balance is: ", self$balance, "\n", sep = "")
      cat("And you are withdrawing: ", amount, "\n", sep = "")
      self$balance <- self$balance - amount
      invisible(self)
    },
    validate = function(amount) {
      stopifnot(is.numeric(amount), amount >= 0)
    },
    print = function() {
      cat("Dear ", self$name, ", your balance is: ", self$balance, sep = "")
      invisible(self)
    }
  )
)

# create an instance of an object
indra <- bankAccount$new(name = "Indra", balance = 100)

indra
#> Dear Indra, your balance is: 100

# do deposits and withdrawals to see if the balance changes
indra$deposit(20)
#> Current balance is: 100
#> And you are depositing:  20

indra
#> Dear Indra, your balance is: 120

indra$withdraw(10)
#> Current balance is: 120
#> And you are withdrawing: 10

indra
#> Dear Indra, your balance is: 110

# make sure input validation checks work
indra$deposit(-20)
#> Error in self$validate(amount): amount >= 0 is not TRUE
indra$deposit("pizza")
#> Error in self$validate(amount): is.numeric(amount) is not TRUE
indra$withdraw(-54)
#> Error in self$validate(amount): amount >= 0 is not TRUE
```

```
Anne <- bankAccount$new(name = "Anne", balance = -45)
#> Error in self$validate(balance): amount >= 0 is not TRUE
```

Create a subclass that errors if you attempt to overdraw

```
bankAccountStrict <- R6::R6Class(
  "bankAccountStrict",
  inherit = bankAccount,
  public = list(
    withdraw = function(amount) {
      # use method from superclass
      super$withdraw(amount)

      if (self$balance < 0) {
        invisible(self)
        stop(
          cat("\nYou are trying to withdraw more that your balance.\n"),
          cat("I'm sorry, ", self$name, ", I'm afraid I can't do that.", sep = ""),
          call. = FALSE
        )
      }
    }
  )
)

# create an instance of an object
Pritesh <- bankAccountStrict$new(name = "Pritesh", balance = 100)

Pritesh
#> Dear Pritesh, your balance is: 100

# do deposits and withdrawals to see if the balance changes
Pritesh$deposit(20)
#> Current balance is: 100
#> And you are depositing:  20

Pritesh
#> Dear Pritesh, your balance is: 120

Pritesh$withdraw(150)
#> Current balance is: 120
#> And you are withdrawing: 150
#>
#> You are trying to withdraw more that your balance.
#> I'm sorry, Pritesh, I'm afraid I can't do that.
#> Error:
```

```
Pritesh
#> Dear Pritesh, your balance is: -30

# make sure input validation checks work
Pritesh$deposit(-20)
#> Error in self$validate(amount): amount >= 0 is not TRUE
Pritesh$deposit("pizza")
#> Error in self$validate(amount): is.numeric(amount) is not TRUE
Pritesh$withdraw(-54)
#> Error in self$validate(amount): amount >= 0 is not TRUE
Pritesh <- bankAccountStrict$new(name = "Pritesh", balance = -45)
#> Error in self$validate(balance): amount >= 0 is not TRUE
```

Create a subclass that charges a fee if overdraw

```
bankAccountFee <- R6::R6Class(
  "bankAccountFee",
  inherit = bankAccount,
  public = list(
    withdraw = function(amount) {
      # use method from superclass
      super$withdraw(amount)

      if (self$balance < 0) {
        cat("\nI am charging you 10 euros for overdrawing.\n")
        self$balance <- self$balance - 10
        invisible(self)
      }
    }
  )
)

# create an instance of an object
Mangesh <- bankAccountFee$new(name = "Mangesh", balance = 100)

Mangesh
#> Dear Mangesh, your balance is: 100

# do deposits and withdrawals to see if the balance changes
Mangesh$deposit(20)
#> Current balance is: 100
#> And you are depositing:  20

Mangesh
#> Dear Mangesh, your balance is: 120
```

```
Mangesh$withdraw(150)
#> Current balance is: 120
#> And you are withdrawing: 150
#>
#> I am charging you 10 euros for overdrawing.

Mangesh
#> Dear Mangesh, your balance is: -40

# make sure input validation checks work
Mangesh$deposit(-20)
#> Error in self$validate(amount): amount >= 0 is not TRUE
Mangesh$deposit("pizza")
#> Error in self$validate(amount): is.numeric(amount) is not TRUE
Mangesh$withdraw(-54)
#> Error in self$validate(amount): amount >= 0 is not TRUE
Mangesh <- bankAccountFee$new(name = "Mangesh", balance = -45)
#> Error in self$validate(balance): amount >= 0 is not TRUE
```

Q2. R6 class for carddeck

```
suit <- c("SPADE", "HEARTS", "DIAMOND", "CLUB") # sigh, Windows encoding issues
value <- c("A", 2:10, "J", "Q", "K")
cards <- paste(rep(value, 4), suit)

deck <- R6::R6Class(
  "deck",
  public = list(
    # fields -----------------------

    # methods ----------------------
    draw = function(n) {
      sample(self$cards, n)
    },
    reshuffle = function() {
      sample(self$cards)
      invisible(self)
    },
    print = function() {
      "Drawn cards are:"
      "Number of remaining cards:"
    }
  )
)
```

```
# create a new instance of this object
mydeck <- deck$new()

# draw cards
mydeck$draw(4)

# reshuffle
```

Chapter 7

Big Picture

No exercises.