Advanced R Exercises

Indrajeet Patil

2022-03-21

Contents

About

My solutions to exercises from the *Advanced R* (2nd Edition) book.

Note that you **should be** reading the official solutions manual, which has more detailed explanations and are guaranteed to have correct solutions as the original author was also involved in writing them. I provide no such guarantees.

- My solutions
- Official solutions

Chapter 1

Introduction

No exercises.

Chapter 2

Names and values

2.1 2.2.2 Exercises

Q1. Explain the relationship between `a`, `b`, `c` and `d` in the following code:

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

A1.

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

The names (`a`, `b`, and `c`) are references to the same object in memory, as can be seen by the identical memory address:

```
library(lobstr)

obj_addrs(list(a, b, c))
#> [1] "0x11f138070" "0x11f138070" "0x11f138070"
```

Except `d`, which is a different object, even if it has the same value:

```
obj_addr(d)
#> [1] "0x11f456310"
```

Q2. 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")
```

A2.

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

```
obj_addr(mean)
#> [1] "0x10eb66338"
obj_addr(base::mean)
#> [1] "0x10eb66338"
obj_addr(get("mean"))
#> [1] "0x10eb66338"
obj_addr(evalq(mean))
#> [1] "0x10eb66338"
obj_addr(match.fun("mean"))
#> [1] "0x10eb66338"
```

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?

A3.

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

A4.

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

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

Q5. I slightly simplified the rules that govern syntactic names. Why is `.123e1` not a syntactic name? Read `?make.names` for the full details.

A5.

Because it is parsed as a number.

```
typeof(.123e1)
#> [1] "double"
```

And as the docs mention (emphasis mine):

> A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or **the dot not followed by a number**.

Here is how `make.names()` will make it syntactic:

```
make.names(.123e1)
#> [1] "X1.23"
```

2.2 2.3.6 Exercises

Q1. Why is `tracemem(1:10)` not useful?

A1.

`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] "<0x10f281738>"
```

Q2. Explain why `tracemem()` shows two copies when you run this code. Hint: carefully look at the difference between this code and the code shown earlier in the section.

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

x[[3]] <- 4
```

A2.

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] "<0x11a2e0fc8>"

x[[3]] <- 4
#> tracemem[0x11a2e0fc8 -> 0x10f6678c8]: eval eval eval_with_user_handlers withVisible
#> tracemem[0x10f6678c8 -> 0x119daf008]: eval eval eval_with_user_handlers withVisible
```

Try with integer:

```
x <- c(1L, 2L, 3L)
tracemem(x)
#> [1] "<0x119d8e908>"

x[[3]] <- 4L
#> tracemem[0x119d8e908 -> 0x119b244c8]: eval eval eval_with_user_handlers withVisible
```

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. Sketch out the relationship between the following objects:

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

A3.

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

ref(a)
#> [1:0x118a941e8] <int>

ref(b)
#>   [1:0x119f78488] <list>
```

```
#>  [2:0x118a941e8] <int>
#>  [2:0x118a941e8]

ref(c)
#>  [1:0x118ec5958] <list>
#>    [2:0x119f78488] <list>
#>    [3:0x118a941e8] <int>
#>    [3:0x118a941e8]
#>  [3:0x118a941e8]
#>  [4:0x118c1f1d0] <int>
```

Q4. What happens when you run this code?

```
x <- list(1:10)
x[[2]] <- x
```

Draw a picture.

A4.

```
x <- list(1:10)
x
#> [[1]]
#>  [1]  1  2  3  4  5  6  7  8  9 10
obj_addr(x)
#> [1] "0x10ff41388"

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] "0x11a2a6ac8"

ref(x)
#>  [1:0x11a2a6ac8] <list>
#>  [2:0x11f79c1c0] <int>
#>    [3:0x10ff41388] <list>
#>    [2:0x11f79c1c0]
```

Figure from the official solution manual can be found here: https://advanced-r-solutions.rbind.io/images/names_values/copy_on_modify_fig2.png

2.3 2.4.1 Exercises

Q1. In the following example, why are `object.size(y)` and `obj_size(y)` so radically different? Consult the documentation of `object.size()`.

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

object.size(y)
obj_size(y)
```

A1.

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. Take the following list. Why is its size somewhat misleading?

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

A2.

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 the output of the following code:

```
a <- runif(1e6)
obj_size(a)

b <- list(a, a)
obj_size(b)
obj_size(a, b)

b[[1]][[1]] <- 10
obj_size(b)
obj_size(a, b)

b[[2]][[1]] <- 10
obj_size(b)
obj_size(a, b)
```

A3. Correctly predicted

```
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
```

2.4 2.5.3 Exercises

Q1. Explain why the following code doesn't create a circular list.

```
x <- list()
x[[1]] <- x
```

A1.

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

```
x <- list()

obj_addr(x)
#> [1] "0x1189fcd88"

tracemem(x)
#> [1] "<0x1189fcd88>"

x[[1]] <- x
#> tracemem[0x1189fcd88 -> 0x118bac2b8]: eval eval eval_with_user_handlers withVisible

obj_addr(x[[1]])
#> [1] "0x1189fcd88"
```

Q2. Wrap the two methods for subtracting medians into two functions, then use the 'bench' package to carefully compare their speeds. How does performance change as the number of columns increase?

A2.

Let's first microbenchmark functions that do and do not create copies for varying lengths of number of columns.

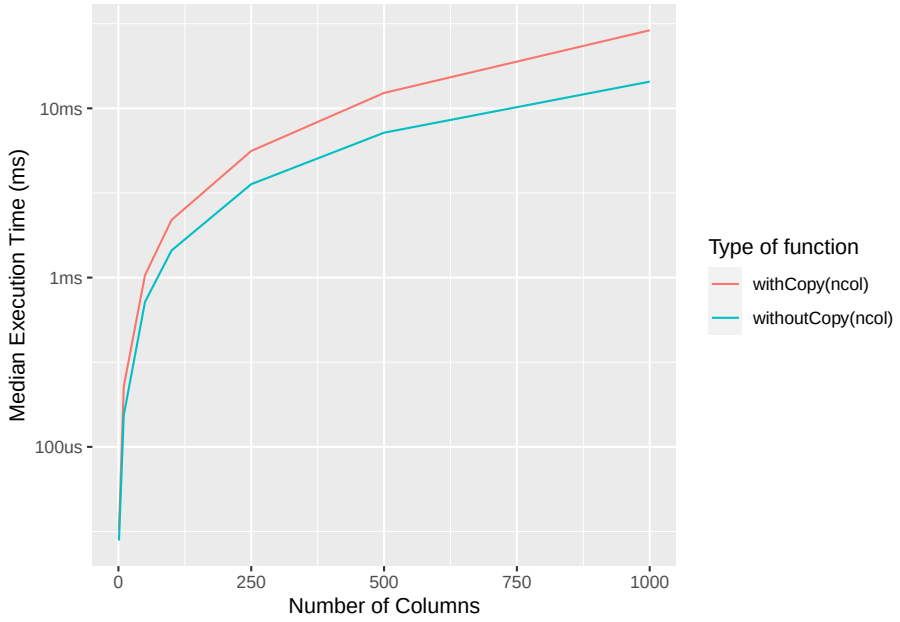
```
library(bench)
library(tidyverse)

generateDataFrame <- function(ncol) {
  as.data.frame(matrix(runif(100 * ncol), nrow = 100))
}
```

```
withCopy <- function(ncol) {
  x <- generateDataFrame(ncol)
  medians <- vapply(x, median, numeric(1))

  for (i in seq_along(medians)) {
    x[[i]] <- x[[i]] - medians[[i]]
  }

  return(x)
}

withoutCopy <- function(ncol) {
  x <- generateDataFrame(ncol)
  medians <- vapply(x, median, numeric(1))

  y <- as.list(x)

  for (i in seq_along(medians)) {
    y[[i]] <- y[[i]] - medians[[i]]
  }

  return(y)
}

benchComparison <- function(ncol) {
  bench::mark(
    withCopy(ncol),
    withoutCopy(ncol),
    iterations = 100,
    check = FALSE
  ) %>%
    dplyr::select(expression:total_time)
}

nColList <- list(1, 10, 50, 100, 250, 500, 1000)

names(nColList) <- as.character(nColList)

benchDf <- purrr::map_dfr(
  .x = nColList,
  .f = benchComparison,
  .id = "nColumns"
)
```

Plotting these benchmarks reveals how the performance gets increasingly worse

as the number of dataframes increases:

```
ggplot(
  benchDf,
  aes(
    x = as.numeric(nColumns),
    y = median,
    group = as.character(expression),
    color = as.character(expression)
  )
) +
  geom_line() +
  labs(
    x = "Number of Columns",
    y = "Median Execution Time (ms)",
    colour = "Type of function"
  )
```



Q3. What happens if you attempt to use `tracemem()` on an environment?

A3.

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 3

Vectors

3.1 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 differences in similar characters:

```
charToRaw("–") # en-dash
#> [1] e2 80 93
charToRaw("—") # em-dash
#> [1] e2 80 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()`

This functions checks if the object is of atomic *type* (or `NULL`), and not if it is an atomic *vector*.

Quoting docs:

> `is.atomic` is true for the atomic types ("logical", "integer", "numeric", "complex", "character" and "raw") and `NULL`.

```
is.atomic(NULL)
#> [1] TRUE

is.vector(NULL)
#> [1] FALSE
```

- `is.numeric()`

Its documentation says:

> `is.numeric` should only return true if the base type of the class is `double` or `integer` and values can reasonably be regarded as `numeric`

Therefore, this function only checks for `double` and `integer` base types and not other types based on top of these types (`factor`, `Date`, `POSIXt`, or `difftime`).

```
x <- factor(c(1L, 2L))

is.numeric(x)
#> [1] FALSE
```

- `is.vector()`

As the documentation for this function reveals:

> `is.vector` returns `TRUE` if `x` is a vector of the specified mode having no attributes *other than names*. It returns `FALSE` otherwise.

Thus, the function can be incorrect in presence if the object has attributes other than `names`.

```
x <- c("x" = 1, "y" = 2)

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

attr(x, "m") <- "abcdef"

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

A better way to check for a vector:

```
is.null(dim(x))
#> [1] TRUE
```

3.2 Exercise 3.3.4

Q1. Reading source code

```
setNames
#> function (object = nm, nm)
#> {
#>     names(object) <- nm
#>     object
```

```
#> }
#> <bytecode: 0x10688f4f8>
#> <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: 0x1163436f0>
#> <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 matrices or dataframes.

```
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 dimensionless **vector**
- `x1`, `x2`, and `x3` are one-dimensional **array**

```
(x <- 1:5)
#> [1] 1 2 3 4 5
dim(x)
#> NULL

(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

dim(x1)
#> [1] 1 1 5
dim(x2)
#> [1] 1 5 1
dim(x3)
#> [1] 5 1 1
```

We can look at the dim attribute

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
```

3.3 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
```

3.4 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 00:01:00 UTC" "1947-08-15 00:00:00 UTC"

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

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
```

3.5 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 4

Subsetting

4.1 Exercise 4.2.6

Q1. Fix each of the following common data frame subsetting errors:

```
mtcars[mtcars$cyl = 4, ]
mtcars[-1:4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]
```

A1. Fixed versions of these commands:

```
mtcars[mtcars$cyl == 4, ]
mtcars[-(1:4), ]
mtcars[mtcars$cyl <= 5, ]
mtcars[mtcars$cyl == 4 | mtcars$cyl == 6, ]
```

Q2. Why does the following code yield five missing values?

```
x <- 1:5
x[NA]
#> [1] NA NA NA NA NA
```

A2. This is because of two reasons:

- The default type of `NA` in R is of `logical` type.

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

- R recycles indexes to match the length of the vector.

```
x <- 1:5
x[c(TRUE, FALSE)] # recycled to c(TRUE, FALSE, TRUE, FALSE, TRUE)
#> [1] 1 3 5
```

Q3. What does `upper.tri()` return? How does subsetting a matrix with it work? Do we need any additional subsetting rules to describe its behaviour?

```
x <- outer(1:5, 1:5, FUN = "*")
x[upper.tri(x)]
```

A3. The documentation for `upper.tri()` states-

> Returns a matrix of logicals the same size of a given matrix with entries `TRUE` in the **upper triangle**

That is, `upper.tri()` return a matrix of logicals.

```
(x <- outer(1:5, 1:5, FUN = "*"))
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    2    3    4    5
#> [2,]    2    4    6    8   10
#> [3,]    3    6    9   12   15
#> [4,]    4    8   12   16   20
#> [5,]    5   10   15   20   25

upper.tri(x)
#>       [,1]  [,2]  [,3]  [,4]  [,5]
#> [1,] FALSE  TRUE  TRUE  TRUE  TRUE
#> [2,] FALSE FALSE  TRUE  TRUE  TRUE
#> [3,] FALSE FALSE FALSE  TRUE  TRUE
#> [4,] FALSE FALSE FALSE FALSE  TRUE
#> [5,] FALSE FALSE FALSE FALSE FALSE
```

When used with a matrix for subsetting, this logical matrix returns a vector:

```
x[upper.tri(x)]
#>  [1]  2  3  6  4  8 12  5 10 15 20
```

Q4. Why does `mtcars[1:20]` return an error? How does it differ from the similar `mtcars[1:20, ]`?

When indexed like a list, data frame columns at given indices will be selected.

```
head(mtcars[1:2])
#>                    mpg cyl
#> Mazda RX4         21.0   6
#> Mazda RX4 Wag     21.0   6
#> Datsun 710        22.8   4
#> Hornet 4 Drive    21.4   6
#> Hornet Sportabout 18.7   8
#> Valiant           18.1   6
```

`mtcars[1:20]` doesn't work because there are 11 columns in `mtcars` dataset.

On the other hand, `mtcars[1:20, ]` indexes a dataframe like a matrix, and because there are indeed 20 rows in `mtcars`, all columns with these rows are selected.

```
nrow(mtcars[1:20, ])
#> [1] 20
```

Q5. Implement your own function that extracts the diagonal entries from a matrix (it should behave like `diag(x)` where `x` is a matrix).

A5. We can combine the existing functions to our advantage:

```
x[!upper.tri(x) & !lower.tri(x)]
#> [1]  1  4  9 16 25

diag(x)
#> [1]  1  4  9 16 25
```

Q6. What does `df[is.na(df)] <- 0` do? How does it work?

A6. This command replaces every instance of `NA` in a dataframe with `0`.

`is.na(df)` produces a matrix of logical values, which provides a way to select and assign.

```
(df <- tibble(x = c(1, 2, NA), y = c(NA, 5, NA)))
#> # A tibble: 3 x 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1    NA
#> 2     2     5
```

```
#> 3    NA    NA

is.na(df)
#>          x     y
#> [1,] FALSE  TRUE
#> [2,] FALSE FALSE
#> [3,]  TRUE  TRUE

class(is.na(df))
#> [1] "matrix" "array"
```

4.2 Exercise 4.3.5

Q1. Brainstorm as many ways as possible to extract the third value from the `cyl` variable in the `mtcars` dataset.

A1. Possible ways to do this:

```
mtcars$cyl[[3]]
#> [1] 4
mtcars[, "cyl"][[3]]
#> [1] 4
mtcars[["cyl"]][[3]]
#> [1] 4

mtcars[3, ]$cyl
#> [1] 4
mtcars[3, "cyl"]
#> [1] 4
mtcars[3, ][["cyl"]]
#> [1] 4

mtcars[[c(2, 3)]]
#> [1] 4
mtcars[3, 2]
#> [1] 4
```

Q2. Given a linear model, e.g., `mod <- lm(mpg ~ wt, data = mtcars)`, extract the residual degrees of freedom. Then extract the R squared from the model summary (`summary(mod)`)

A2. Specified linear model:

```
mod <- lm(mpg ~ wt, data = mtcars)
```

- extracting the residual degrees of freedom

```
mod$df.residual
#> [1] 30

# or

mod[["df.residual"]]
#> [1] 30
```

- extracting the R squared from the model summary

```
summary(mod)$r.squared
#> [1] 0.7528328
```

4.3 Exercise 4.5.9

Q1. How would you randomly permute the columns of a data frame? (This is an important technique in random forests.) Can you simultaneously permute the rows and columns in one step?

Q2. How would you select a random sample of `m` rows from a data frame? What if the sample had to be contiguous (i.e., with an initial row, a final row, and every row in between)?

Q3. How could you put the columns in a data frame in alphabetical order?

Chapter 5

Control flow

5.1 Exercise 5.2.4

Q1. What type of vector does each of the following calls to `ifelse()` return?

```
ifelse(TRUE, 1, "no")
ifelse(FALSE, 1, "no")
ifelse(NA, 1, "no")
```

Read the documentation and write down the rules in your own words.

A1. Here are *da rulz*:

- It's type unstable, i.e. the type of return will depend on the type of each condition (`yes` and `no`, i.e.):

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

- It works only for cases where `test` argument evaluates to a `logical` type:

```
ifelse(NA_real_, 1, "no")
#> [1] NA
ifelse(NaN, 1, "no")
#> [1] NA
```

- If the `test` argument doesn't resolve to `logical` type, it will try to coerce the output to `logical` type:

```
# 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
```

To quote the docs for this function:

> A vector of the same length and attributes (including dimensions and `"class"`) as `test` and data values from the values of `yes` or `no`. The mode of the answer will be coerced from logical to accommodate first any values taken from yes and then any values taken from `no`.

Q2. Why does the following code work?

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

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

A2. The code works because the conditions - even though of `numeric` type - are successfully coerced to a `logical` type.

```
as.logical(length(1:10))
#> [1] TRUE

as.logical(length(numeric()))
#> [1] FALSE
```

5.2 Exercise 5.3.3

Q1. 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
```

A1. 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(paste("i:", i, ", x[i]:", x[i], ", out[i]:", out[i]))

  out[i] <- x[i]^2
}
#> [1] "i: 1 , x[i]: NA , out[i]: NULL"
#> [1] "i: 0 , x[i]:  , out[i]: "

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. 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
```

A2. The iterator variable `x` initially takes all values of the vector `xs`. We can check this by printing `x` for each iteration:

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

It is worth noting that `x` is not updated after each iteration, otherwise it will take increasingly bigger values of `xs`, and the loop will never end executing.

Q3. What does the following code tell you about when the index is updated?

```
for (i in 1:3) {
  i <- i * 2
  print(i)
}
#> [1] 2
#> [1] 4
#> [1] 6
```

A3. In a `for` loop the index is updated in the **beginning** of each iteration. Otherwise, we will encounter an infinite loop.

```
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
```

Also, worth contrasting the behavior of `for` loop with that of `while` loop:

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

Chapter 6

Functions

6.1 Exercise 6.2.5

Q1. Function names

Given a name, `match.fun()` lets you find a function.

```
match.fun("mean")
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x15066ce90>
#> <environment: namespace:base>
```

But, given a function, it doesn't make sense to find its name in R because there can be multiple names bound to the same function.

```
f1 <- function(x) mean(x)
f2 <- f1

match.fun("f1")
#> function(x) mean(x)

match.fun("f2")
#> function(x) mean(x)
```

Q2. Correct way to call anonymous functions

This is not correct since the function will evaluate `3()`, which is syntactically not allowed since literals can't be treated like functions.

```
(function(x) 3())()
#> Error in (function(x) 3())(): attempt to apply non-function
```

This is correct.

```
(function(x) 3)()
#> [1] 3
```

Q3. Scan code for opportunities to use anonymous function

Self activity.

Q4. Detecting functions and primitive functions

Use `is.function()` to check if an object is a function:

```
# these are functions
f <- function(x) 3
is.function(mean)
#> [1] TRUE
is.function(f)
#> [1] TRUE

# these aren't
is.function("x")
#> [1] FALSE
is.function(new.env())
#> [1] FALSE
```

Use `is.primitive()` to check if a function is primitive:

```
# primitive
is.primitive(sum)
#> [1] TRUE
is.primitive(`+`)
#> [1] TRUE

# not primitive
is.primitive(mean)
#> [1] FALSE
is.primitive(read.csv)
#> [1] FALSE
```

Q5. Detecting functions and primitive functions

```
objs <- mget(ls("package:base", all = TRUE), inherits = TRUE)
funs <- Filter(is.function, objs)
```

Which base function has the most arguments?

`scan()` function has the most arguments.

```
library(tidyverse)

df_formals <- purrr::map_df(funs, ~ length(formals(.))) %>%
  tidyr::pivot_longer(
    cols = dplyr::everything(),
    names_to = "function",
    values_to = "argumentCount"
  ) %>%
  dplyr::arrange(desc(argumentCount))
```

How many base functions have no arguments? What's special about those functions?

At the time of writing, 253 base functions have no arguments. Most of these are primitive functions

```
dplyr::filter(df_formals, argumentCount == 0)
#> # A tibble: 251 x 2
#>    `function` argumentCount
#>    <chr>               <int>
#>  1 -                       0
#>  2 :                       0
#>  3 ::                      0
#>  4 :::                     0
#>  5 !                       0
#>  6 !=                      0
#>  7 ...elt                  0
#>  8 ...length               0
#>  9 ...names                0
#> 10 .C                      0
#> # ... with 241 more rows
```

How could you adapt the code to find all primitive functions?

```
objs <- mget(ls("package:base", all = TRUE), inherits = TRUE)
funs <- Filter(is.function, objs)
primitives <- Filter(is.primitive, funs)

names(primitives)
#>   [1] "-"                  ":"
#>   [3] "::"                 ":::"
#>   [5] "!"                  "!="
#>   [7] "...elt"             "...length"
#>   [9] "...names"           ".C"
#>  [11] ".cache_class"       ".Call"
#>  [13] ".Call.graphics"     ".class2"
#>  [15] ".External"          ".External.graphics"
#>  [17] ".External2"         ".Fortran"
#>  [19] ".Internal"          ".isMethodsDispatchOn"
#>  [21] ".Primitive"         ".primTrace"
#>  [23] ".primUntrace"       ".subset"
#>  [25] ".subset2"           "("
#>  [27] "["                  "[["
#>  [29] "[[<-"               "[<-"
#>  [31] "{"                  "@"
#>  [33] "@<-"                "*"
#>  [35] "/"                  "&"
#>  [37] "&&"                 "%*%"
#>  [39] "%/%"                "%%"
#>  [41] "^"                  "+"
#>  [43] "<"                  "<-"
#>  [45] "<<-"                "<="
#>  [47] "="                  "=="
#>  [49] ">"                  ">="
#>  [51] "|"                  "||"
#>  [53] "~"                  "$"
#>  [55] "$<-"                "abs"
#>  [57] "acos"               "acosh"
#>  [59] "all"                "any"
#>  [61] "anyNA"              "Arg"
#>  [63] "as.call"            "as.character"
#>  [65] "as.complex"         "as.double"
#>  [67] "as.environment"     "as.integer"
#>  [69] "as.logical"         "as.numeric"
#>  [71] "as.raw"             "asin"
#>  [73] "asinh"              "atan"
#>  [75] "atanh"              "attr"
#>  [77] "attr<-"             "attributes"
#>  [79] "attributes<-"       "baseenv"
```

```
#>  [81] "break"                "browser"
#>  [83] "c"                    "call"
#>  [85] "ceiling"              "class"
#>  [87] "class<-"              "Conj"
#>  [89] "cos"                  "cosh"
#>  [91] "cospi"                "cummax"
#>  [93] "cummin"               "cumprod"
#>  [95] "cumsum"               "digamma"
#>  [97] "dim"                  "dim<-"
#>  [99] "dimnames"             "dimnames<-"
#> [101] "emptyenv"             "enc2native"
#> [103] "enc2utf8"             "environment<-"
#> [105] "exp"                  "expm1"
#> [107] "expression"           "floor"
#> [109] "for"                  "forceAndCall"
#> [111] "function"             "gamma"
#> [113] "gc.time"              "globalenv"
#> [115] "if"                   "Im"
#> [117] "interactive"          "invisible"
#> [119] "is.array"             "is.atomic"
#> [121] "is.call"              "is.character"
#> [123] "is.complex"           "is.double"
#> [125] "is.environment"       "is.expression"
#> [127] "is.finite"            "is.function"
#> [129] "is.infinite"          "is.integer"
#> [131] "is.language"          "is.list"
#> [133] "is.logical"           "is.matrix"
#> [135] "is.na"                "is.name"
#> [137] "is.nan"               "is.null"
#> [139] "is.numeric"           "is.object"
#> [141] "is.pairlist"          "is.raw"
#> [143] "is.recursive"         "is.single"
#> [145] "is.symbol"            "isS4"
#> [147] "lazyLoadDBfetch"      "length"
#> [149] "length<-"             "levels<-"
#> [151] "lgamma"               "list"
#> [153] "log"                  "log10"
#> [155] "log1p"                "log2"
#> [157] "max"                  "min"
#> [159] "missing"              "Mod"
#> [161] "names"                "names<-"
#> [163] "nargs"                "next"
#> [165] "nzchar"               "oldClass"
#> [167] "oldClass<-"           "on.exit"
#> [169] "pos.to.env"           "proc.time"
```

```
#> [171] "prod"                "quote"
#> [173] "range"               "Re"
#> [175] "rep"                 "repeat"
#> [177] "retracemem"          "return"
#> [179] "round"               "seq_along"
#> [181] "seq_len"             "seq.int"
#> [183] "sign"                "signif"
#> [185] "sin"                 "sinh"
#> [187] "sinpi"               "sqrt"
#> [189] "standardGeneric"     "storage.mode<-"
#> [191] "substitute"          "sum"
#> [193] "switch"              "tan"
#> [195] "tanh"                "tanpi"
#> [197] "tracemem"            "trigamma"
#> [199] "trunc"               "unclass"
#> [201] "untracemem"          "UseMethod"
#> [203] "while"               "xtfrm"
```

Q6. Important components of a function

Except for primitive functions, all functions have 3 important components:

- `formals()`
- `body()`
- `environment()`

Q7. Printing of function environment

All package functions print their environment:

```
# base
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x15066ce90>
#> <environment: namespace:base>

# other package function
purrr::map
#> function (.x, .f, ...)
#> {
#>     .f <- as_mapper(.f, ...)
#>     .Call(map_impl, environment(), ".x", ".f", "list")
#> }
#> <bytecode: 0x13050f750>
#> <environment: namespace:purrr>
```

There are two exceptions to this rule:

- primitive functions:

```
sum
#> function (..., na.rm = FALSE)  .Primitive("sum")
```

- functions created in the global environment:

```
f <- function(x) mean(x)
f
#> function(x) mean(x)
```

6.2 Exercise 6.4.5

Q1. All about *c*

In `c(c = c)`: * first *c* is interpreted as a function `c()` * second *c* as a name for the vector element * third *c* as a variable with value `10`

```
c <- 10
c(c = c)
#>  c
#> 10
```

Q2. Four principles that govern how R looks for values

1. Name masking (names defined inside a function mask names defined outside a function)
2. Functions vs. variables (the rule above also applies to function names)
3. A fresh start (every time a function is called a new environment is created to host its execution)
4. Dynamic look-up (R looks for values when the function is run, not when the function is created)

Q3. Predict the return

Correctly predicted

```
f <- function(x) {
  f <- function(x) {
    f <- function() {
      x^2
    }
    f() + 1
  }
  f(x) * 2
}

f(10)
#> [1] 202
```

6.3 Exercise 6.5.4

Q1. Property of `&&`

`&&` evaluates left to right and short-circuit evaluation, i.e., if the first operand is `TRUE`, R will short-circuit and not even look at the second operand.

```
x_ok <- function(x) {
  !is.null(x) && length(x) == 1 && x > 0
}

x_ok(NULL)
#> [1] FALSE

x_ok(1)
#> [1] TRUE

x_ok(1:3)
#> [1] FALSE
```

Replacing `&&` is `&` is undesirable because it performs element-wise logical comparisons and returns a vector of values that is not always useful for decision (`TRUE`, `FALSE`, or `NA`).

```
x_ok <- function(x) {
  !is.null(x) & length(x) == 1 & x > 0
}

x_ok(NULL)
#> logical(0)
```

```
x_ok(1)
#> [1] TRUE

x_ok(1:3)
#> [1] FALSE FALSE FALSE
```

Q2. Principle behind return

The function returns `100`, and the principle at work here is lazy evaluation. When function environment encounters `x`, it evaluates argument `x = z` and since the name `z` is already bound to value 100, `x` is also bound to the same value.

We can check this by looking at the memory addresses:

```
f2 <- function(x = z) {
  z <- 100
  print(x)

  print(lobstr::obj_addrs(list(x, z)))
}

f2()
#> [1] 100
#> [1] "0x133352240" "0x133352240"
```

Q3. Principle behind return

TODO:

```
y <- 10
f1 <- function(x =
                 {
                   y <- 1
                   2
                 },
               y = 0)
{
  c(x, y)
}

f1()
#> [1] 2 1

y
#> [1] 10
```

6.4 Exercise 6.6.1

Q1. Explain results

```
sum(1, 2, 3)
#> [1] 6

mean(1, 2, 3)
#> [1] 1

sum(1, 2, 3, na.omit = TRUE)
#> [1] 7

mean(1, 2, 3, na.omit = TRUE)
#> [1] 1
```

Let's look at arguments for these functions:

```
str(sum)
#> function (..., na.rm = FALSE)
str(mean)
#> function (x, ...)
```

As can be seen, `sum()` function doesn't have `na.omit` argument. So, the input `na.omit = TRUE` is treated as `1` (logical implicitly coerced to numeric), and thus the results. So, the expression evaluates to `sum(1, 2, 3, 1)`.

For `mean()` function, there is only one parameter (`x`) and it's matched by the first argument (`1`). So, the expression evaluates to `mean(1)`.

Q2. Finding documentation for `plot` arguments

First, check documentation for `plot()`:

```
str(plot)
#> function (x, y, ...)
```

Since `...` are passed to `par()`, we can look at its documentation:

```
str(par)
#> function (..., no.readonly = FALSE)
```

The docs for all parameters of interest reside there.

Q3. Reading source code for `plot.default`

Source code can be found here.

`plot.default()` passes `...` to `localTitle()`, which passes it to `title()`.

`title()` has four parts: `main`, `sub`, `xlab`, `ylab`.

So having a single argument `col` would not work as it will be ambiguous as to which element to apply this argument to.

```
localTitle <- function(..., col, bg, pch, cex, lty, lwd) title(...)

title <- function(main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
                  line = NA, outer = FALSE, ...) {
  main <- as.graphicsAnnot(main)
  sub <- as.graphicsAnnot(sub)
  xlab <- as.graphicsAnnot(xlab)
  ylab <- as.graphicsAnnot(ylab)
  .External.graphics(C_title, main, sub, xlab, ylab, line, outer, ...)
  invisible()
}
```

Chapter 7

Functionals

7.1 Exercise 9.2.6

Q1. Use `as_mapper()` to explore how `{purrr}` generates anonymous functions for the integer, character, and list helpers. What helper allows you to extract attributes? Read the documentation to find out.

A1.

- Experiments with `{purrr}`:

```
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: 0x106b696d0>

map(x, list(2, 1))
#> [[1]]
```

```
#> NULL
#>
#> [[2]]
#> [1] 3
as_mapper(list(2, 1))
#> function (x, ...)
#> pluck(x, 2, 1, .default = NULL)
#> <environment: 0x106c71708>

# 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: 0x106e4f388>

# 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: 0x107008710>

# 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. `map(1:3, ~ runif(2))` is a useful pattern for generating random numbers, but `map(1:3, runif(2))` is not. Why not? Can you explain why it returns the result that it does?

A2.

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] 0.3277057 0.5281832
#>
#> [[2]]
#> [1] 0.1145581 0.3668396
#>
#> [[3]]
#> [1] 0.2415530 0.2160025
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, 0.161950123263523, 0.9537498415447777, .default = NULL)
#> <environment: 0x107fccb98>
```

Q3. Use the appropriate `map()` function to:

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

`a) Compute the standard deviation of every numeric column in a mixed data frame. (Hint`

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

A3.

- 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. The following code simulates the performance of a t-test for non-normal data. Extract the p-value from each test, then visualise.

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

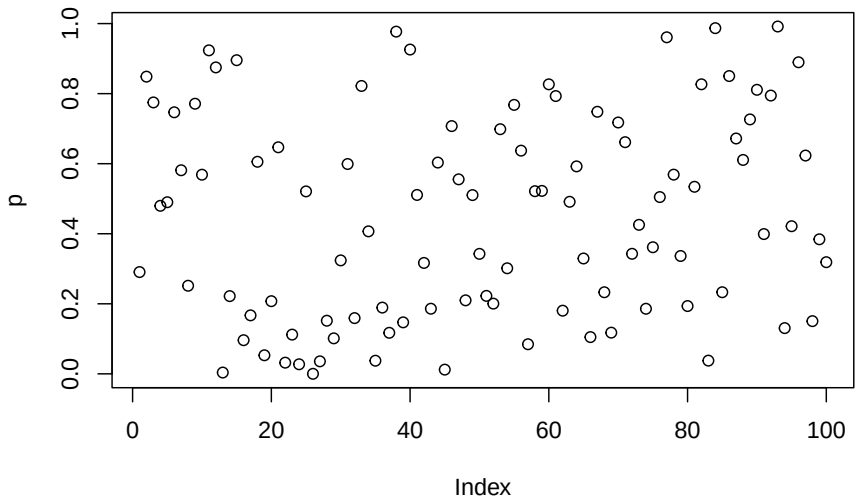
A4.

- Extract the p-value from each test:

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

(p <- map_dbl(trials, "p.value"))
#>   [1] 2.905993e-01 8.485220e-01 7.749201e-01 4.797501e-01
#>   [5] 4.899823e-01 7.467550e-01 5.812470e-01 2.515032e-01
#>   [9] 7.712268e-01 5.685582e-01 9.236198e-01 8.748857e-01
#>  [13] 3.719542e-03 2.219313e-01 8.955004e-01 9.609371e-02
#>  [17] 1.669638e-01 6.054486e-01 5.300681e-02 2.074621e-01
#>  [21] 6.468552e-01 3.196939e-02 1.120017e-01 2.731860e-02
#>  [25] 5.209050e-01 9.413047e-05 3.572049e-02 1.516687e-01
#>  [29] 1.013473e-01 3.239340e-01 5.990808e-01 1.591380e-01
#>  [33] 8.218988e-01 4.068413e-01 3.745831e-02 1.891550e-01
#>  [37] 1.172247e-01 9.770973e-01 1.470017e-01 9.258676e-01
#>  [41] 5.107506e-01 3.166517e-01 1.859580e-01 6.032471e-01
#>  [45] 1.206745e-02 7.073449e-01 5.552909e-01 2.098825e-01
#>  [49] 5.103633e-01 3.425824e-01 2.225584e-01 2.004672e-01
#>  [53] 6.984613e-01 3.014092e-01 7.678901e-01 6.372488e-01
#>  [57] 8.451530e-02 5.214836e-01 5.223274e-01 8.265587e-01
#>  [61] 7.931288e-01 1.803332e-01 4.912102e-01 5.923250e-01
#>  [65] 3.293817e-01 1.049301e-01 7.482425e-01 2.329628e-01
#>  [69] 1.176367e-01 7.176388e-01 6.614496e-01 3.427500e-01
#>  [73] 4.254455e-01 1.858013e-01 3.614359e-01 5.047116e-01
#>  [77] 9.608684e-01 5.687391e-01 3.364789e-01 1.935490e-01
#>  [81] 5.340901e-01 8.266896e-01 3.778229e-02 9.869634e-01
#>  [85] 2.330069e-01 8.502332e-01 6.719423e-01 6.106811e-01
#>  [89] 7.264387e-01 8.109384e-01 3.990860e-01 7.945415e-01
#>  [93] 9.916576e-01 1.305803e-01 4.214985e-01 8.895582e-01
#>  [97] 6.232296e-01 1.504816e-01 3.841351e-01 3.187831e-01
```

- Visualise the extracted p-values:

```
plot(p)
```



Q5. The following code uses a map nested inside another map to apply a function to every element of a nested list. Why does it fail, and what do you need to do to make it work?

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

triple <- function(x) x * 3
map(x, map, .f = triple)
#> Error in .f(.x[[i]], ...): unused argument (function (.x, .f, ...)
#> {
#>     .f <- as_mapper(.f, ...)
#>     .Call(map_impl, environment(), ".x", ".f", "list")
#> })
```

A5. Here is the fixed version:

```
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 using the formulas stored in this list:

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

A6. Fitting linear models to the `mtcars` dataset using the provided formulas:

```
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. Fit the model `mpg ~ disp` to each of the bootstrap replicates of `mtcars` in the list below, then extract the R^2 of the model fit (Hint: you can compute the R^2 with `summary()`.)

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

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

A7. This can be done using `map_dbl()`:

```
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] 0.6980447 0.7153403 0.6408987 0.7627210 0.6908151
#>  [6] 0.8170847 0.7867446 0.7288730 0.6950510 0.8300409
```

7.2 Exercise 9.4.6

Q1. Explain the results of `modify(mtcars, 1)`.

A1. `modify()` returns the object of type same as the input. Since the input here is a data frame of certain dimensions and `.f = 1` translates to plucking the first element in each column, it returns a data frames 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. Rewrite the following code to use `iwalk()` instead of `walk2()`. What are the advantages and disadvantages?

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

A2. Rewritten versions are below:

- 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 how the following code transforms a data frame using functions stored in a list.

```
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))
```

Compare and contrast the `map2()` approach to this `map()` approach:

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

A3. `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. What does `write.csv()` return, i.e. what happens if you use it with `map2()` instead of `walk2()`?

A4. If we use `map2()`, it will work, but it will print `NULL`s to the terminal for every list element.

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

7.3 Exercise 9.6.3

Q1. Why isn't `is.na()` a predicate function? What base R function is closest to being a predicate version of `is.na()`?

A1. As mentioned in the docs:

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

The `is.na()` function does not return a `logical` scalar, 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. `simple_reduce()` has a problem when `x` is length 0 or length 1. Describe the source of the problem and how you might go about fixing it.

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

A2. Supplied function:

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

This function 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
```

Q3. Implement the `span()` function from Haskell: given a list `x` and a predicate function `f`, `span(x, f)` returns the location of the longest sequential run of elements where the predicate is true. (Hint: you might find `rle()` helpful.)

Q4. Implement `arg_max()`. It should take a function and a vector of inputs, and return the elements of the input where the function returns the highest

value. For example, `arg_max(-10:5, function(x) x ^ 2)` should return -10. `arg_max(-5:5, function(x) x ^ 2)` should return `c(-5, 5)`. Also implement the matching `arg_min()` function.

Q5. The function below scales a vector so it falls in the range [0, 1]. How would you apply it to every column of a data frame? How would you apply it to every numeric column in a data frame?

```
scale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

7.4 Exercise 9.7.3

Q1. How does `apply()` arrange the output? Read the documentation and perform some experiments.

Q2. What do `eapply()` and `rapply()` do? Does purrr have equivalents?

A2. As mentioned in the documentation:

- `eapply()`

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

Here is an example:

```
library(rlang)
#>
#> Attaching package: 'rlang'
#> The following objects are masked from 'package:purrr':
#>
#>     %@%, as_function, flatten, flatten_chr,
#>     flatten_dbl, flatten_int, flatten_lgl,
#>     flatten_raw, invoke, splice

e <- env("x" = 1, "y" = 2)
rlang::env_print(e)
#> <environment: 0x106ba6518>
#> Parent: <environment: global>
#> Bindings:
#> * x: <dbl>
```

```
#> * y: <dbl>

eapply(e, as.character)
#> $x
#> [1] "1"
#>
#> $y
#> [1] "2"
```

{purrr} doesn't have any function to iterate over environments.

- `rapply()`

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

Here is an example:

```
X <- list(list(a = TRUE, b = list(c = c(4L, 3.2))), d = 9.0)

rapply(X, as.character, classes = "numeric", how = "replace")
#> [[1]]
#> [[1]]$a
#> [1] TRUE
#>
#> [[1]]$b
#> [[1]]$b$c
#> [1] "4"   "3.2"
#>
#>
#>
#> $d
#> [1] "9"
```

{purrr} has something similar in `modify_depth()`.

```
X <- list(list(a = TRUE, b = list(c = c(4L, 3.2))), d = 9.0)

purrr::modify_depth(X, .depth = 2L, .f = length)
#> [[1]]
#> [[1]]$a
#> [1] 1
#>
#> [[1]]$b
```

```
#> [1] 1
#>
#>
#> $d
#> [1] 1
```

Q3. Challenge: read about the fixed point algorithm. Complete the exercises using R.

Chapter 8

Function factories

Chapter 9

Base Types

No exercises.

Chapter 10

S3

10.1 Exercise 13.2.1

Q1. Describe the difference between `t.test()` and `t.data.frame()`. When is each function called?

A1.

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

```
library(sloop)

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

Q2. Make a list of commonly used base R functions that contain `.` in their name but are not `S3` methods.

A2. Here are a few common R functions with `.` but that are not `S3` methods:

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

- `on.exit()` etc.

For full list, you could do:

```
base_functions <- getNamespaceExports("base")

base_functions[grepl("(\\w+)(\\.)(\\w+)", base_functions)]
```

For example,

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

Q3. What does the `as.data.frame.data.frame()` method do? Why is it confusing? How could you avoid this confusion in your own code?

A3. It's an `S3` **method** for **generic** `as.data.frame()`.

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

It can be seen in all methods supported by this generic:

```
s3_methods_generic("as.data.frame") %>%
  dplyr::filter(class == "data.frame")
#> # A tibble: 1 x 4
#>   generic       class      visible source
#>   <chr>         <chr>      <lgl>   <chr>
#> 1 as.data.frame data.frame TRUE    base
```

Q4. Describe the difference in behaviour in these two calls.

```
set.seed(1014)
some_days <- as.Date("2017-01-31") + sample(10, 5)
mean(some_days)
#> [1] "2017-02-06"
mean(unclass(some_days))
#> [1] 17203.4
```

A4.

- Before unclassing, the `mean` generic dispatches `.Date` method:

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

some_days
#> [1] "2017-02-06" "2017-02-09" "2017-02-05" "2017-02-08"
#> [5] "2017-02-07"

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

mean(some_days)
#> [1] "2017-02-07"
```

- After unclassing, the `mean` generic dispatches `.numeric` method:

```
unclass(some_days)
#> [1] 17203 17206 17202 17205 17204

mean(unclass(some_days))
#> [1] 17204

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

Q5. What class of object does the following code return? What base type is it built on? What attributes does it use?

```
x <- ecdf(rpois(100, 10))
x
```

A5. The object is based on base type `closure`[1], which is a type of function.

```
x <- ecdf(rpois(100, 10))
x
#> Empirical CDF
#> Call: ecdf(rpois(100, 10))
#>  x[1:18] =      2,      3,      4,  ...,     18,     19

otype(x)
#> [1] "S3"
typeof(x)
#> [1] "closure"
```

[1] of "object of type 'closure' is not subsettable" fame

Its class is `ecdf`, which has other superclasses.

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

Apart from `class`, it has the following attributes:

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

Q6. What class of object does the following code return? What base type is it built on? What attributes does it use?

```
x <- table(rpois(100, 5))
x
```

A6. The object is based on base type `integer`.

```
x <- table(rpois(100, 5))
x
#>
#>  1  2  3  4  5  6  7  8  9 10
#>  7  7 18 13 14 14 16  4  4  3

otype(x)
#> [1] "S3"
typeof(x)
#> [1] "integer"
```

Its class is `table`.

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

Apart from `class`, it has the following attributes:

```
attributes(x)
#> $dim
#> [1] 10
```

```
#>
#> $dimnames
#> $dimnames[[1]]
#>  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
#>
#>
#> $class
#> [1] "table"
```

10.2 Exercise 13.3.4

Q1. Write a constructor for `data.frame` objects. What base type is a data frame built on? What attributes does it use? What are the restrictions placed on the individual elements? What about the names?

A1.

```
my_data_frame <- function(...,
                          row.names = NULL,
                          check.rows = FALSE,
                          check.names = TRUE,
                          fix.empty.names = TRUE,
                          stringsAsFactors = FALSE) {
  structure(
    df,
    class = "data.frame"
  )
}
```

Q2. Enhance my `factor()` helper to have better behaviour when one or more `values` is not found in `levels`. What does `base::factor()` do in this situation?

Q3. Carefully read the source code of `factor()`. What does it do that my constructor does not?

Q4. Factors have an optional "contrasts" attribute. Read the help for `C()`, and briefly describe the purpose of the attribute. What type should it have? Rewrite the `new_factor()` constructor to include this attribute.

Q5. Read the documentation for `utils::as.roman()`. How would you write a constructor for this class? Does it need a validator? What might a helper do?

Chapter 11

R6

11.1 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
```

11.2 Exercise 14.3.3

Q2. Class to store and check password

```
library(R6)

checkCredentials <- R6Class(
  "checkCredentials",
  public = list(
    # setter
    set_password = function(password) {
      private$.password <- password
    },

    # checker
    check_password = function(password) {
      if (is.null(private$.password)) {
        stop("No password set to check against.")
      }

      identical(password, private$.password)
    },

    # the default print method prints the private fields as well
    print = function() {
      "Password: XXXX"

      # for method chaining
      invisible(self)
    }
  ),
```

```
  private = list(
    .password = NULL
  )
)

myCheck <- checkCredentials$new()
myCheck

myCheck$set_password("1234")

myCheck$check_password("abcd")
#> [1] FALSE
myCheck$check_password("1234")
#> [1] TRUE
```

But, of course, everything is possible:

```
myCheck$.__enclos_env__$private$.password
#> [1] "1234"
```

Q4. Inheriting private fields and methods from superclass

Unlike classical OOP in other languages (e.g. C++), R6 subclasses also have access to the private methods in superclass (or base class).

For instance, in the following example, the `Duck` class has a private method `$quack()`, but its subclass `Mallard` can access it using `super$quack()`.

```
Duck <- R6Class("Duck",
  private = list(quack = function() print("Quack Quack"))
)

Mallard <- R6Class("Mallard",
  inherit = Duck,
  public = list(quack = function() super$quack())
)

myMallard <- Mallard$new()
myMallard$quack()
#> [1] "Quack Quack"
```

11.3 Exercise 14.4.4

Q1. Write R6 class to edit file

```
library(R6)

fileEditor <- R6Class(
  "fileEditor",
  public = list(
    initialize = function(filePath) {
      private$.connection <- file(filePath, open = "wt")
    },
    append_line = function(text) {
      cat(
        text,
        file = private$.connection,
        sep = "\n",
        append = TRUE
      )
    }
  ),
  private = list(
    .connection = NULL,
    # according to R6 docs, the destructor method should be private
    finalize = function() {
      print("Closing the file connection!")
      close(private$.connection)
    }
  )
)
```

Let's check if it works as expected:

```
greetMom <- function() {
  f <- tempfile()
  myfileEditor <- fileEditor$new(f)

  readLines(f)

  myfileEditor$append_line("Hi mom!")
  myfileEditor$append_line("It's a beautiful day!")

  readLines(f)
}

greetMom()
#> [1] "Hi mom!"                "It's a beautiful day!"

# force garbage collection
```

```
gc()
#> [1] "Closing the file connection!"
#>           used (Mb) gc trigger  (Mb) limit (Mb) max used
#> Ncells 1085410 58.0    2225400 118.9         NA  1418000
#> Vcells 1879015 14.4    8388608  64.0      16384  2819107
#>        (Mb)
#> Ncells 75.8
#> Vcells 21.6
```

Chapter 12

Big Picture

No exercises.

Chapter 13

Debugging

No exercises.

Chapter 14

Measuring performance

14.1 Exercise 23.2.4

Q1. Profile the following function with `torture = TRUE`. What is surprising? Read the source code of `rm()` to figure out what's going on.

```
f <- function(n = 1e5) {
  x <- rep(1, n)
  rm(x)
}
```

A1.

Let's first source the functions mentioned in exercises.

```
library(profvis)

source("profiling-exercises.R")
```

First, we try without `torture = TRUE`: it returns no meaningful results.

```
profvis(f())
#> Error in parse_rprof(prof_output, expr_source): No parsing data available. Maybe your functio
```

Maybe because the function runs too fast?

```
bench::mark(f(), check = FALSE, iterations = 1000)
#> # A tibble: 1 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
```

```
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 f()           102us    145us     6479.     792KB     98.7
```

As mentioned in the docs, setting `torture = TRUE`

> Triggers garbage collection after every torture memory allocation call.

This process somehow never seems to finish and crashes the RStudio session when it stops!

```
profvis(f(), torture = TRUE)
```

The question says that documentation for `rm()` may provide clues:

```
rm
#> function (..., list = character(), pos = -1, envir = as.environment(pos),
#>     inherits = FALSE)
#> {
#>     dots <- match.call(expand.dots = FALSE)$...
#>     if (length(dots) && !all(vapply(dots, function(x) is.symbol(x) ||
#>         is.character(x), NA, USE.NAMES = FALSE)))
#>         stop("... must contain names or character strings")
#>     names <- vapply(dots, as.character, "")
#>     if (length(names) == 0L)
#>         names <- character()
#>     list <- .Primitive("c")(list, names)
#>     .Internal(remove(list, envir, inherits))
#> }
#> <bytecode: 0x150bdef48>
#> <environment: namespace:base>
```

I still couldn't figure out why. I would recommend checking out the official answer.

14.2 Exercise 23.3.3

Q1. Instead of using `bench::mark()`, you could use the built-in function `system.time()`. But `system.time()` is much less precise, so you'll need to repeat each operation many times with a loop, and then divide to find the average time of each operation, as in the code below.

```
n <- 1e6
system.time(for (i in 1:n) sqrt(x)) / n
system.time(for (i in 1:n) x^0.5) / n
```

How do the estimates from `system.time()` compare to those from `bench::mark()`? Why are they different?

A1.

```
library(dplyr)

n <- 1e6
x <- runif(100)

# bench -------------------

bench_df <- bench::mark(
  sqrt(x),
  x^0.5,
  iterations = n
)

t_bench_df <- bench_df %>%
  dplyr::select(expression, time) %>%
  dplyr::rowwise() %>%
  dplyr::mutate(mean = mean(unlist(time))) %>%
  dplyr::ungroup() %>%
  dplyr::select(-time)

# system.time -------------------

# garbage collection performed immediately before the timing
t1_systime_gc <- system.time(for (i in 1:n) sqrt(x), gcFirst = TRUE) / n
t2_systime_gc <- system.time(for (i in 1:n) x^0.5, gcFirst = TRUE) / n

# garbage collection not performed immediately before the timing
t1_systime_nogc <- system.time(for (i in 1:n) sqrt(x), gcFirst = FALSE) / n
t2_systime_nogc <- system.time(for (i in 1:n) x^0.5, gcFirst = FALSE) / n

t_systime_df <- tibble(
  "expression" = bench_df$expression,
  "systime_with_gc_us" = c(t1_systime_gc["elapsed"], t2_systime_gc["elapsed"]),
  "systime_with_nogc_us" = c(t1_systime_nogc["elapsed"], t2_systime_nogc["elapsed"])
) %>%
  dplyr::mutate(
```

```
    systime_with_gc_us = systime_with_gc_us * 1e6,
    systime_with_nogc_us = systime_with_nogc_us * 1e6
  )
```

Compare results from these alternatives:

```
t_bench_df
#> # A tibble: 2 x 2
#>   expression     mean
#>   <bch:expr> <bch:tm>
#> 1 sqrt(x)     424.3ns
#> 2 x^0.5         1.3us

t_systime_df
#> # A tibble: 2 x 3
#>   expression systime_with_gc_us systime_with_nogc_us
#>   <bch:expr>              <dbl>                <dbl>
#> 1 sqrt(x)                 0.397                0.403
#> 2 x^0.5                   1.22                 1.24
```

The comparison reveals that these two approaches yield quite similar results.

Q2. Here are two other ways to compute the square root of a vector. Which do you think will be fastest? Which will be slowest? Use microbenchmarking to test your answers.

```
x^(1 / 2)
exp(log(x) / 2)
```

A2.

Microbenchmarking all ways to compute square root of a vector mentioned in this chapter.

```
x <- runif(1000)

bench::mark(
  sqrt(x),
  x^0.5,
  x^(1 / 2),
  exp(log(x) / 2),
  iterations = 1000
) %>%
  dplyr::arrange(median)
#> # A tibble: 4 x 6
```

```
#>   expression          min   median `itr/sec` mem_alloc
#>   <bch:expr>     <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 sqrt(x)           984ns   1.44us   565188.    7.86KB
#> 2 exp(log(x)/2)    6.19us   7.09us   139420.    7.86KB
#> 3 x^0.5            9.14us  10.04us   101038.    7.86KB
#> 4 x^(1/2)          9.18us  10.21us    98318.    7.86KB
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0
#> 3        0
#> 4        0
```

The specialized primitive function `sqrt()` (written in `C`) is the fastest way to compute square root.

Chapter 15

Rewriting R code in C++

15.1 Exercise 25.2.6

Q1. Figure out base function corresponding to Rccp code

```
library(Rcpp)
```

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double f1(NumericVector x) {
  int n = x.size();
  double y = 0;

  for(int i = 0; i < n; ++i) {
    y += x[i] / n;
  }
  return y;
}

// [[Rcpp::export]]
NumericVector f2(NumericVector x) {
  int n = x.size();
  NumericVector out(n);

  out[0] = x[0];
  for(int i = 1; i < n; ++i) {
    out[i] = out[i - 1] + x[i];
```

```
  }
  return out;
}

// [[Rcpp::export]]
bool f3(LogicalVector x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    if (x[i]) return true;
  }
  return false;
}

// [[Rcpp::export]]
int f4(Function pred, List x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    LogicalVector res = pred(x[i]);
    if (res[0]) return i + 1;
  }
  return 0;
}

// [[Rcpp::export]]
NumericVector f5(NumericVector x, NumericVector y) {
  int n = std::max(x.size(), y.size());
  NumericVector x1 = rep_len(x, n);
  NumericVector y1 = rep_len(y, n);

  NumericVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = std::min(x1[i], y1[i]);
  }

  return out;
}
```

`f1()` is the same as `mean()`:

```
x <- c(1, 2, 3, 4, 5, 6)

f1(x)
```

```
#> [1] 3.5
mean(x)
#> [1] 3.5
```

`f2()` is the same as `cumsum()`:

```
x <- c(1, 3, 5, 6)

f2(x)
#> [1]  1  4  9 15
cumsum(x)
#> [1]  1  4  9 15
```

`f3()` is the same as `any()`:

```
x1 <- c(TRUE, FALSE, FALSE, TRUE)
x2 <- c(FALSE, FALSE)

f3(x1)
#> [1] TRUE
any(x1)
#> [1] TRUE

f3(x2)
#> [1] FALSE
any(x2)
#> [1] FALSE
```

`f4()` is the same as `Position()`:

```
x <- list("a", TRUE, "m", 2)

f4(is.numeric, x)
#> [1] 4
Position(is.numeric, x)
#> [1] 4
```

`f5()` is the same as `pmin()`:

```
v1 <- c(1, 3, 4, 5, 6, 7)
v2 <- c(1, 2, 7, 2, 8, 1)

f5(v1, v2)
```

```
#> [1] 1 2 4 2 6 1
pmin(v1, v2)
#> [1] 1 2 4 2 6 1
```

Q2. Converting base function to Rcpp

The performance benefits are not going to be observed if the function is primitive since those are already tuned to the max in R for performance. So, expect performance gain only for `diff()` and `var()`.

```
is.primitive(all)
#> [1] TRUE
is.primitive(cumprod)
#> [1] TRUE
is.primitive(diff)
#> [1] FALSE
is.primitive(range)
#> [1] TRUE
is.primitive(var)
#> [1] FALSE
```

- `all()`

```
#include <vector>
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
bool allC(std::vector<bool> x)
{
    for (const auto& xElement : x)
    {
        if (!xElement) return false;
    }

    return true;
}
```

```
v1 <- rep(TRUE, 10)
v2 <- c(rep(TRUE, 5), rep(FALSE, 5))

all(v1)
#> [1] TRUE
allC(v1)
#> [1] TRUE
```

```
all(v2)
#> [1] FALSE
allC(v2)
#> [1] FALSE

# performance benefits?
bench::mark(
  all(c(rep(TRUE, 1000), rep(FALSE, 1000))),
  allC(c(rep(TRUE, 1000), rep(FALSE, 1000))),
  iterations = 100
)
#> # A tibble: 2 x 6
#>   expression                                      min
#>   <bch:expr>                                  <bch:tm>
#> 1 all(c(rep(TRUE, 1000), rep(FALSE, 1000)))    6.52us
#> 2 allC(c(rep(TRUE, 1000), rep(FALSE, 1000)))  12.46us
#>     median `itr/sec` mem_alloc `gc/sec`
#>   <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1   7.54us   132953.     15.8KB        0
#> 2  13.16us    73593.     18.3KB        0
```

- cumprod()

```
#include <vector>

// [[Rcpp::export]]
std::vector<double> cumulativeProduct(std::vector<double> x)
{
    std::vector<double> out = x;

    for (size_t i = 1; i < x.size(); i++)
    {
        out[i] = out[i - 1] * x[i];
    }

    return out;
}
```

```
v1 <- c(10, 4, 6, 8)

cumprod(v1)
#> [1]   10   40  240 1920
cumulativeProduct(v1)
#> [1]   10   40  240 1920
```

```
# performance benefits?
bench::mark(
  cumprod(v1),
  cumulativeProduct(v1),
  iterations = 100
)
#> # A tibble: 2 x 6
#>   expression                 min   median `itr/sec`
#>   <bch:expr>             <bch:tm> <bch:tm>     <dbl>
#> 1 cumprod(v1)               82ns  82.02ns  3464443.
#> 2 cumulativeProduct(v1)   1.48us   1.56us   414869.
#>   mem_alloc `gc/sec`
#>   <bch:byt>    <dbl>
#> 1        0B        0
#> 2    7.02KB        0
```

- diff()

TODO

- range()

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// [[Rcpp::export]]
std::vector<double> rangeC(std::vector<double> x)
{
    std::vector<double> rangeVec{0.0, 0.0};

    rangeVec.at(0) = *std::min_element(x.begin(), x.end());
    rangeVec.at(1) = *std::max_element(x.begin(), x.end());

    return rangeVec;
}
```

```
v1 <- c(10, 4, 6, 8)

range(v1)
#> [1]  4 10
rangeC(v1)
```

```
#> [1]  4 10

# performance benefits?
bench::mark(
  range(v1),
  rangeC(v1),
  iterations = 100
)
#> # A tibble: 2 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 range(v1)    1.39us   1.52us   614361.        0B        0
#> 2 rangeC(v1)   1.35us   1.76us   542606.    7.02KB        0
```

- var()

```
#include <vector>
#include <cmath>
#include <numeric>
using namespace std;

// [[Rcpp::export]]
double variance(std::vector<double> x)
{
    double sumSquared{0};

    double mean = std::accumulate(x.begin(), x.end(), 0.0) / x.size();

    for (const auto& xElement : x)
    {
        sumSquared += pow(xElement - mean, 2.0);
    }

    return sumSquared / (x.size() - 1);
}
```

```
v1 <- c(1, 4, 7, 8)

var(v1)
#> [1] 10
variance(v1)
#> [1] 10

# performance benefits?
```

```
bench::mark(
  var(v1),
  variance(v1),
  iterations = 100
)
#> # A tibble: 2 x 6
#>   expression        min   median `itr/sec` mem_alloc
#>   <bch:expr>   <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 var(v1)        6.31us   6.97us   141172.        0B
#> 2 variance(v1)   1.31us   1.84us   548957.    7.02KB
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0
```

15.2 Exercise 25.4.5

Q1. Rewrite functions with original `na.rm` argument

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <math.h>
#include <Rcpp.h>
using namespace std;

// [[Rcpp::export]]
std::vector<double> rangeC_NA(std::vector<double> x, bool removeNA = true)
{
    std::vector<double> rangeVec{0.0, 0.0};

    bool naPresent = std::any_of(
        x.begin(),
        x.end(),
        [](double d)
        { return isnan(d); });

    if (naPresent)
    {
        if (removeNA)
        {
            std::remove(x.begin(), x.end(), NAN);
        }
```

```
        else
        {
            rangeVec.at(0) = NA_REAL; // NAN;
            rangeVec.at(1) = NA_REAL; // NAN;

            return rangeVec;
        }
    }

    rangeVec.at(0) = *std::min_element(x.begin(), x.end());
    rangeVec.at(1) = *std::max_element(x.begin(), x.end());

    return rangeVec;
}
```

```
v1 <- c(10, 4, NA, 6, 8)

range(v1, na.rm = FALSE)
#> [1] NA NA
rangeC_NA(v1, FALSE)
#> [1] NA NA

range(v1, na.rm = TRUE)
#> [1]  4 10
rangeC_NA(v1, TRUE)
#> [1]  4 10
```

Q2. Rewrite functions without original `na.rm` argument

15.3 Exercise 25.5.7

Q1. `median.default()` using `partial_sort()`

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// [[Rcpp::export]]
double medianC(std::vector<double> x)
{
    int middleIndex = static_cast<int>(x.size() / 2);
```

```
    std::partial_sort(x.begin(), x.begin() + middleIndex, x.end());

    // for even number of observations
    if (x.size() % 2 == 0)
    {
        return (x[middleIndex - 1] + x[middleIndex]) / 2;
    }

    return x[middleIndex];
}
```

```
v1 <- c(1, 3, 3, 6, 7, 8, 9)
v2 <- c(1, 2, 3, 4, 5, 6, 8, 9)

median.default(v1)
#> [1] 6
medianC(v1)
#> [1] 6

median.default(v2)
#> [1] 4.5
medianC(v2)
#> [1] 4.5

# performance benefits?
bench::mark(
  median.default(v2),
  medianC(v2),
  iterations = 100
)
#> # A tibble: 2 x 6
#>   expression             min   median `itr/sec` mem_alloc
#>   <bch:expr>         <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 median.default(v2)  17.47us  18.49us    53184.        0B
#> 2 medianC(v2)          1.35us   1.68us   572404.     2.49KB
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0
```