# Advanced R Exercises

Indrajeet Patil

2025-03-16

# Contents

# About

This book provides solutions to exercises from Hadley Wickham's *Advanced R* (2nd edition) book.

I started working on this book as part of my process to learn by solving each of the book's exercises. While comparing solutions to the official solutions manual, I realized that some solutions took different approaches or were at least explained differently. I'm sharing these solutions in case others might find another perspective or explanation than the official solution manual helpful for building understanding.

Although I have tried to make sure that all solutions are correct, the blame for any inaccuracies lies solely with me. I'd very much appreciate any suggestions or corrections.

# Chapter 1

# Introduction

No exercises.

# Chapter 2

# Names and values

Loading the needed libraries:

```r
library(lobstr)
```

## 2.1 Binding basics (Exercise 2.2.2)

---

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

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

**A1.** The names (`a`, `b`, and `c`) have same values and point to the same object in memory, as can be seen by their identical memory addresses:

```r
obj_addrs <- obj_addrs(list(a, b, c))
unique(obj_addrs)
#> [1] "0x5642a358d2e0"
```

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

```
obj_addr(d)
#> [1] "0x5642a33e17a0"
```

---

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

```
obj_addrs <- obj_addrs(list(
  mean,
  base::mean,
  get("mean"),
  evalq(mean),
  match.fun("mean")
))

unique(obj_addrs)
#> [1] "0x5642a3d93018"
```

---

**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.** `make.names()` uses following rules to convert non-syntactic names into syntactic ones:

- it prepends non-syntactic names with `X`
- it converts invalid characters (like `@`) to `.`
- it adds a `.` as a suffix if the name is a reserved keyword

```
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.** `.123e1` is not a syntacti name because it is parsed as a number, and not as a string:

```
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**.

---

## 2.2 Copy-on-modify (Exercise 2.3.6)

---

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

**A1.** `tracemem()` traces copying of objects in R. For example:

```r
x <- 1:10

tracemem(x)
#> [1] "<0x5642a5f5c5c8>"

x <- x + 1

untracemem(x)
```

But since the object created in memory by `1:10` is not assigned a name, it can't be addressed or modified from R, and so there is nothing to trace.

```r
obj_addr(1:10)
#> [1] "0x5642a4a43d68"

tracemem(1:10)
#> [1] "<0x5642a4072bf8>"
```

---

**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.

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

x[[3]] <- 4
untracemem(x)
```

**A2.** This is because the initial atomic vector is of type `integer`, but `4` (and not `4L`) is of type `double`. This is why a new copy is created.

```r
x <- c(1L, 2L, 3L)
typeof(x)
#> [1] "integer"
tracemem(x)
#> [1] "<0x5642a60789b8>"

x[[3]] <- 4
#> tracemem[0x5642a60789b8 -> 0x5642a442fc08]: eval eval
↪  withVisible withCallingHandlers eval eval with_handlers
↪  doWithOneRestart withOneRestart withRestartList
↪  doWithOneRestart withOneRestart withRestartList withRestarts
↪  <Anonymous> evaluate in_dir in_input_dir eng_r block_exec
↪  call_block process_group withCallingHandlers <Anonymous>
↪  process_file <Anonymous> <Anonymous> do.call eval eval eval
↪  eval eval.parent local
```

```
#> tracemem[0x5642a442fc08 -> 0x5642a685bef8]: eval eval
↪  withVisible withCallingHandlers eval eval with_handlers
↪  doWithOneRestart withOneRestart withRestartList
↪  doWithOneRestart withOneRestart withRestartList withRestarts
↪  <Anonymous> evaluate in_dir in_input_dir eng_r block_exec
↪  call_block process_group withCallingHandlers <Anonymous>
↪  process_file <Anonymous> <Anonymous> do.call eval eval eval
↪  eval eval.parent local
untracemem(x)

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

Trying with an integer should not create another copy:

```
x <- c(1L, 2L, 3L)
typeof(x)
#> [1] "integer"
tracemem(x)
#> [1] "<0x5642a6a9b718>"

x[[3]] <- 4L
#> tracemem[0x5642a6a9b718 -> 0x5642a66e2308]: eval eval
↪  withVisible withCallingHandlers eval eval with_handlers
↪  doWithOneRestart withOneRestart withRestartList
↪  doWithOneRestart withOneRestart withRestartList withRestarts
↪  <Anonymous> evaluate in_dir in_input_dir eng_r block_exec
↪  call_block process_group withCallingHandlers <Anonymous>
↪  process_file <Anonymous> <Anonymous> do.call eval eval eval
↪  eval eval.parent local
untracemem(x)

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

To understand why this still produces a copy, here is an explanation from the official 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:

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

**A3.** We can understand the relationship between these objects by looking at their memory addresses:

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

ref(a)
#> [1:0x5642a4fb3718] <int>

ref(b)
#>  [1:0x5642a6885ec8] <list>
#>  [2:0x5642a4fb3718] <int>
#>  [2:0x5642a4fb3718]

ref(c)
#>  [1:0x5642a6a6c368] <list>
#>   [2:0x5642a6885ec8] <list>
#>   [3:0x5642a4fb3718] <int>
#>   [3:0x5642a4fb3718]
#>  [3:0x5642a4fb3718]
#>  [4:0x5642a526ff58] <int>
```

Here is what we learn:

- The name `a` references object `1:10` in the memory.
- The name `b` is bound to a list of two references to the memory address of `a`.
- The name `c` is also bound to a list of references to `a` and `b`, and `1:10` object (not bound to any name).

---

**Q4.** What happens when you run this code?

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

Draw a picture.

**A4.**

```r
x <- list(1:10)
x
#> [[1]]
#>  [1]  1  2  3  4  5  6  7  8  9 10
obj_addr(x)
#> [1] "0x5642a7d6ba48"

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] "0x5642a7e49718"

ref(x)
#>  [1:0x5642a7e49718] <list>
#> [2:0x5642a7d7bbd0] <int>
#>   [3:0x5642a7d6ba48] <list>
#>    [2:0x5642a7d7bbd0]
```

I don't have access to OmniGraffle software, so I am including here the figure from the official solution manual:

---

## 2.3 Object size (Exercise 2.4.1)

---

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

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

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

**A1.** As mentioned in the docs for `object.size()`:

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

This is why the sizes are so different:

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

object.size(y)
#> 8005648 bytes

obj_size(y)
#> 80.90 kB
```

---

**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 as part of base packages, so doesn't make much sense to measure their size because they are never going to be *not* available.

```
funs <- list(mean, sd, var)
obj_size(funs)
#> 18.76 kB
```

---

**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

```r
a <- runif(1e6)
obj_size(a)
#> 8.00 MB

b <- list(a, a)
obj_size(b)
#> 8.00 MB
obj_size(a, b)
#> 8.00 MB

b[[1]][[1]] <- 10
obj_size(b)
#> 16.00 MB
obj_size(a, b)
#> 16.00 MB

b[[2]][[1]] <- 10
obj_size(b)
#> 16.00 MB
obj_size(a, b)
#> 24.00 MB
```

Key pieces of information to keep in mind to make correct predictions:

- Size of empty vector

```r
obj_size(double())
#> 48 B
```

- Size of a single double: 8 bytes

```r
obj_size(double(1))
#> 56 B
```

- Copy-on-modify semantics

---

## 2.4 Modify-in-place (Exercise 2.5.3)

---

**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] "0x5642a8065c70"

tracemem(x)
#> [1] "<0x5642a8065c70>"

x[[1]] <- x
#> tracemem[0x5642a8065c70 -> 0x5642a81429f8]: eval eval
↪  withVisible withCallingHandlers eval eval with_handlers
↪  doWithOneRestart withOneRestart withRestartList
↪  doWithOneRestart withOneRestart withRestartList withRestarts
↪  <Anonymous> evaluate in_dir in_input_dir eng_r block_exec
↪  call_block process_group withCallingHandlers <Anonymous>
↪  process_file <Anonymous> <Anonymous> do.call eval eval eval
↪  eval eval.parent local

obj_addr(x[[1]])
#> [1] "0x5642a8065c70"
```

---

**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?

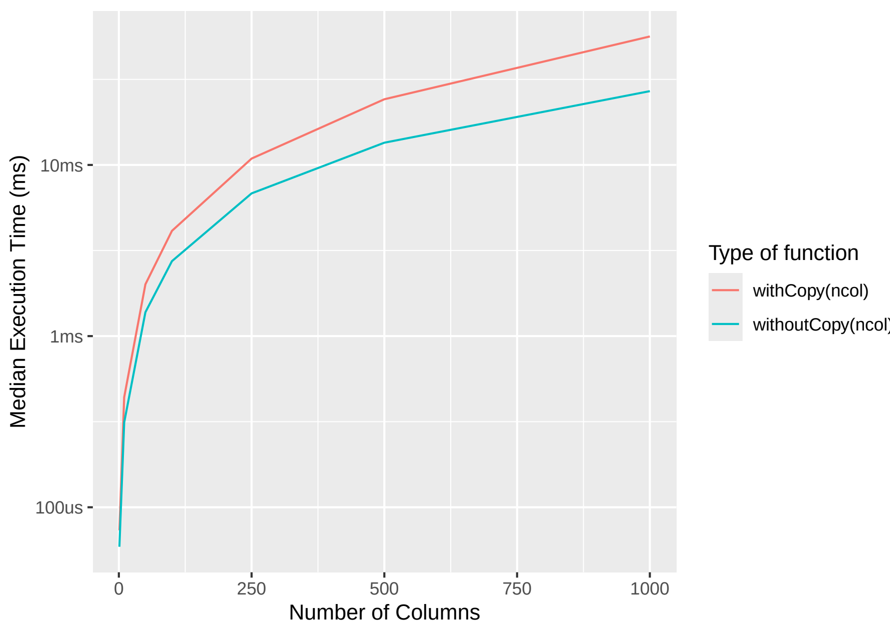
**A2.** Let's first microbenchmark functions that do and do not create copies for varying lengths of number of columns.

```
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 data frames 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 for `tracemem()` makes it clear 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
```

---

## 2.5 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bench       * 1.1.4   2025-01-16 [1] RSPM
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  colorspace    2.1-1   2024-07-26 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  crayon        1.5.3   2024-06-20 [1] RSPM
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  dplyr       * 1.1.4   2023-11-17 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  farver        2.1.2   2024-05-13 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  forcats     * 1.0.0   2023-01-29 [1] RSPM
```

```
#>  generics      0.1.3   2022-07-05 [1] RSPM
#>  ggplot2     * 3.5.1   2024-04-23 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  grid          4.4.3   2025-02-28 [3] local
#>  gtable        0.3.6   2024-10-25 [1] RSPM
#>  hms           1.1.3   2023-03-21 [1] RSPM
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  labeling      0.4.3   2023-08-29 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  lobstr      * 1.1.2   2022-06-22 [1] RSPM
#>  lubridate   * 1.9.4   2024-12-08 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  munsell       0.5.1   2024-04-01 [1] RSPM
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  prettyunits   1.2.0   2023-09-24 [1] RSPM
#>  profmem       0.6.0   2020-12-13 [1] RSPM
#>  purrr       * 1.0.4   2025-02-05 [1] RSPM
#>  R6            2.6.1   2025-02-15 [1] RSPM
#>  readr       * 2.1.5   2024-01-10 [1] RSPM
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  scales        1.3.0   2023-11-28 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr     * 1.5.1   2023-11-14 [1] RSPM
#>  tibble      * 3.2.1   2023-03-20 [1] RSPM
#>  tidyr       * 1.3.1   2024-01-24 [1] RSPM
#>  tidyselect    1.2.1   2024-03-11 [1] RSPM
#>  tidyverse   * 2.0.0   2023-02-22 [1] RSPM
#>  timechange    0.3.0   2024-01-18 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  tzdb          0.4.0   2023-05-12 [1] RSPM
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  withr         3.0.2   2024-10-28 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
```

```
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 3

# Vectors

## 3.1 Atomic vectors (Exercises 3.2.5)

**Q1.** How do you create raw and complex scalars? (See `?raw` and `?complex`.)

**A1.** In R, scalars are nothing but vectors of length 1, and can be created using the same constructor.

- Raw vectors

The raw type holds raw bytes, and can be created using `charToRaw()`. For example,

```
x <- "A string"

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

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

An alternative is to use `as.raw()`:

```
as.raw("-") # en-dash
#> Warning: NAs introduced by coercion
#> Warning: out-of-range values treated as 0 in coercion to
#> raw
#> [1] 00
as.raw("-") # em-dash
```

```
#> Warning: NAs introduced by coercion
#> Warning: out-of-range values treated as 0 in coercion to
#> raw
#> [1] 00
```

- Complex vectors

Complex vectors are 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.** Test your knowledge of the vector coercion rules by predicting the output of the following uses of `c()`:

```
c(1, FALSE)
c("a", 1)
c(TRUE, 1L)
```

**A2.** The vector coercion rules dictate that the data type with smaller size will be converted to data type with bigger size.

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

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

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

**Q3.** Why is `1 == "1"` true? Why is `-1 < FALSE` true? Why is `"one" < 2` false?

**A3.** The coercion rules for vectors reveal why some of these comparisons return the results that they do.

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

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

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

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

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

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

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

**Q4.** Why is the default missing value, `NA`, a logical vector? What's special about logical vectors? (Hint: think about `c(FALSE, NA_character_)`.)

**A4.** 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.

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

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

**Q5.** Precisely what do `is.atomic()`, `is.numeric()`, and `is.vector()` test for?

**A5.** Let's discuss them one-by-one.

- `is.atomic()`

This function checks if the object is a vector of atomic *type* (or `NULL`).

Quoting docs:

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

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

is.atomic(list(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`).

```r
is.numeric(1L)
#> [1] TRUE

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

- `is.vector()`

As per its documentation:

> `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 incorrectif the object has attributes other than `names`.

```r
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:

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

## 3.2 Attributes (Exercises 3.3.4)

**Q1.** How is `setNames()` implemented? How is `unname()` implemented? Read the source code.

**A1.** Let's have a look at implementations for these functions.

- `setNames()`

```r
setNames
#> function (object = nm, nm)
#> {
#>     names(object) <- nm
#>     object
#> }
#> <bytecode: 0x564b3783da38>
#> <environment: namespace:stats>
```

Given this function signature, we can see why, when no first argument is given, the result is still a named vector.

```r
setNames(, c("a", "b"))
#>   a   b
#> "a" "b"

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

- `unname()`

```
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: 0x564b370287e8>
#> <environment: namespace:base>
```

`unname()` removes existing names (or dimnames) by setting them to `NULL`.

```
unname(setNames(, c("a", "b")))
#> [1] "a" "b"
```

**Q2.** What does `dim()` return when applied to a 1-dimensional vector? When might you use `NROW()` or `NCOL()`?

**A2.** Dimensions for a 1-dimensional vector are `NULL`. For example,

```
dim(c(1, 2))
#> NULL
```

`NROW()` and `NCOL()` are helpful for getting dimensions for 1D vectors by treating them as if they were matrices or dataframes.

```
# example-1
x <- character(0)

dim(x)
#> NULL

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

ncol(x)
#> NULL
```

```r
NCOL(x)
#> [1] 1

# example-2
y <- 1:4

dim(y)
#> NULL

nrow(y)
#> NULL
NROW(y)
#> [1] 4

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

**Q3.** How would you describe the following three objects? What makes them different from `1:5`?

```r
x1 <- array(1:5, c(1, 1, 5))
x2 <- array(1:5, c(1, 5, 1))
x3 <- array(1:5, c(5, 1, 1))
```

**A3.** `x1`, `x2`, and `x3` are one-dimensional **array**s, but with different "orientations", if we were to mentally visualize them.

`x1` has 5 entries in the third dimension, `x2` in the second dimension, while `x1` in the first dimension.

**Q4.** An early draft used this code to illustrate `structure()`:

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

But when you print that object you don't see the comment attribute. Why? Is the attribute missing, or is there something else special about it? (Hint: try using help.)

**A4.** 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.

```r
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 S3 atomic vectors (Exercises 3.4.5)

**Q1.** What sort of object does `table()` return? What is its type? What attributes does it have? How does the dimensionality change as you tabulate more variables?

**A1.** `table()` returns an array of `integer` type and its dimensions scale with the number of variables present.

```r
(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.** What happens to a factor when you modify its levels?

```
f1 <- factor(letters)
levels(f1) <- rev(levels(f1))
```

**A2.** Its levels change 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.** What does this code do? How do `f2` and `f3` differ from `f1`?

```
f2 <- rev(factor(letters))
f3 <- factor(letters, levels = rev(letters))
```

**A3.** In this code:

- `f2`: Only the underlying integers are reversed, but levels remain unchanged.

```
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`: Both the levels and the underlying integers are reversed.

```
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 Lists (Exercises 3.5.4)

**Q1.** List all the ways that a list differs from an atomic vector.

**A1.** Here is a table of comparison:

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

**Q2.** Why do you need to use `unlist()` to convert a list to an atomic vector? Why doesn't `as.vector()` work?

**A2.** A list already *is* a (generic) vector, so `as.vector()` is not going to change anything, and there is no `as.atomic.vector`. Thus, we need to use `unlist()`.

[1] a list can contain a mix of types
[2] a list can contain itself
[3] `lobstr::ref(c(1, 2))`
[4] `lobstr::ref(list(1, 2))`

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

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

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

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

**Q3.** Compare and contrast `c()` and `unlist()` when combining a date and date-time into a single vector.

**A3.** Let's first create a date and datetime object

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

And check their attributes and underlying `double` representation:

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

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()`

Since `S3` method for `c()` dispatches on the first argument, the resulting class of the vector is going to be the same as the first argument. Because of this, some attributes will be lost.

```
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()`

It 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 Data frames and tibbles (Exercises 3.6.8)

**Q1.** Can you have a data frame with zero rows? What about zero columns?

**A1.** Data frame with 0 rows is possible. This is basically a list with a vector of length 0.

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

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

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

And, finally, data frame with 0 rows *and* columns is also possible:

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

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

Although, it might not be common to *create* such data frames, they can be results of subsetting. For example,

```r
BOD[0, ]
#> [1] Time   demand
#> <0 rows> (or 0-length row.names)

BOD[, 0]
#> data frame with 0 columns and 6 rows

BOD[0, 0]
#> data frame with 0 columns and 0 rows
```

**Q2.** What happens if you attempt to set rownames that are not unique?

**A2.** If you attempt to set data frame rownames that are not unique, it will not work.

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

**Q3.** If `df` is a data frame, what can you say about `t(df)`, and `t(t(df))`? Perform some experiments, making sure to try different column types.

**A3.** Transposing a data frame:

- transforms it into a matrix
- 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?
```

```r
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.** What does `as.matrix()` do when applied to a data frame with columns of different types? How does it differ from `data.matrix()`?

**A4.** The return type of `as.matrix()` depends on the data frame column types.

As docs for `as.matrix()` mention:

> The method for data frames will return a character matrix if there is only atomic columns and any non-(numeric/logical/complex) column, applying as.vector to factors and format to other non-character columns. Otherwise the usual coercion hierarchy (logical < integer < double < complex) will be used, e.g. all-logical data frames will be coerced to a logical matrix, mixed logical-integer will give an integer matrix, etc.

Let's experiment:

```r
# 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
```

On the other hand, `data.matrix()` always returns a numeric matrix.

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. Factors and ordered factors are replaced by their internal codes. [...] Character columns are first converted to factors and then to integers.

Let's experiment:

```
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" ...
```

## 3.6 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> ─ Session info ──────────────────────────────────────────
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
```

```
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  purrr         1.0.4   2025-02-05 [1] RSPM
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 4

# Subsetting

Attaching the needed libraries:

```r
library(tibble)
```

## 4.1 Selecting multiple elements (Exercises 4.2.6)

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

```r
# styler: off
mtcars[mtcars$cyl = 4, ]
mtcars[-1:4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]
# styler: on
```

**A1.** Fixed versions of these commands:

```r
# `==` instead of `=`
mtcars[mtcars$cyl == 4, ]

# `-(1:4)` instead of `-1:4`
mtcars[-(1:4), ]

# `,` was missing
mtcars[mtcars$cyl <= 5, ]
```

```
# correct subsetting syntax
mtcars[mtcars$cyl == 4 | mtcars$cyl == 6, ]
mtcars[mtcars$cyl %in% c(4, 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**

```
(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, elements corresponding to `TRUE` in the subsetting matrix are selected. But, instead of a matrix, this 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, ]`?

**A4.** 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 only 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 expression replaces every instance of `NA` in `df` with `0`.

`is.na(df)` produces a matrix of logical values, which provides a way of subsetting.

```
(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 Selecting a single element (Exercises 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 to extract the third value from the `cyl` variable in the `mtcars` dataset:

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

```
mtcars[3, ]$cyl
#> [1] 4
mtcars[3, "cyl"]
#> [1] 4
mtcars[, "cyl"][[3]]
#> [1] 4
mtcars[3, 2]
#> [1] 4
mtcars$cyl[[3]]
#> [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.** Given that objects of class `lm` are lists, we can use subsetting operators to extract elements we want.

```
mod <- lm(mpg ~ wt, data = mtcars)
class(mod)
#> [1] "lm"
typeof(mod)
#> [1] "list"
```

- extracting the residual degrees of freedom

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

- extracting the R squared from the model summary

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

## 4.3 Applications (Exercises 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?

**A1.** Let's create a small data frame to work with.

```
df <- head(mtcars)

# original
df
#>                    mpg cyl disp  hp drat    wt  qsec vs am
#> Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1
#> Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0
#> Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0
#> Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0
#>                   gear carb
#> Mazda RX4            4    4
#> Mazda RX4 Wag        4    4
#> Datsun 710           4    1
#> Hornet 4 Drive       3    1
#> Hornet Sportabout    3    2
#> Valiant              3    1
```

To randomly permute the columns of a data frame, we can combine `[` and `sample()` as follows:

- randomly permute columns

```
df[sample.int(ncol(df))]
#>                   drat    wt carb am  qsec vs  hp  mpg disp
#> Mazda RX4         3.90 2.620    4  1 16.46  0 110 21.0  160
#> Mazda RX4 Wag     3.90 2.875    4  1 17.02  0 110 21.0  160
#> Datsun 710        3.85 2.320    1  1 18.61  1  93 22.8  108
#> Hornet 4 Drive    3.08 3.215    1  0 19.44  1 110 21.4  258
#> Hornet Sportabout 3.15 3.440    2  0 17.02  0 175 18.7  360
#> Valiant           2.76 3.460    1  0 20.22  1 105 18.1  225
#>                   cyl gear
#> Mazda RX4           6    4
#> Mazda RX4 Wag       6    4
#> Datsun 710          4    4
#> Hornet 4 Drive      6    3
#> Hornet Sportabout   8    3
#> Valiant             6    3
```

- randomly permute rows

```
df[sample.int(nrow(df)), ]
#>                    mpg cyl disp  hp drat    wt  qsec vs am
#> Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1
#> Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1
#> Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0
#> Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0
#>                   gear carb
#> Datsun 710           4    1
#> Mazda RX4 Wag        4    4
#> Mazda RX4            4    4
#> Hornet Sportabout    3    2
#> Hornet 4 Drive       3    1
#> Valiant              3    1
```

- randomly permute columns and rows

```
df[sample.int(nrow(df)), sample.int(ncol(df))]
#>                    qsec vs gear am    wt drat carb disp  hp
#> Mazda RX4         16.46  0    4  1 2.620 3.90    4  160 110
#> Hornet 4 Drive    19.44  1    3  0 3.215 3.08    1  258 110
#> Datsun 710        18.61  1    4  1 2.320 3.85    1  108  93
#> Mazda RX4 Wag     17.02  0    4  1 2.875 3.90    4  160 110
#> Valiant           20.22  1    3  0 3.460 2.76    1  225 105
#> Hornet Sportabout 17.02  0    3  0 3.440 3.15    2  360 175
#>                    mpg cyl
#> Mazda RX4         21.0   6
#> Hornet 4 Drive    21.4   6
#> Datsun 710        22.8   4
#> Mazda RX4 Wag     21.0   6
#> Valiant           18.1   6
#> Hornet Sportabout 18.7   8
```

**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)?

**A2.** Let's create a small data frame to work with.

```
df <- head(mtcars)

# original
df
```

```
#>                    mpg cyl disp  hp drat    wt  qsec vs am
#> Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1
#> Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0
#> Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0
#> Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0
#>                   gear carb
#> Mazda RX4            4    4
#> Mazda RX4 Wag        4    4
#> Datsun 710           4    1
#> Hornet 4 Drive       3    1
#> Hornet Sportabout    3    2
#> Valiant              3    1

# number of rows to sample
m <- 2L
```

To select a random sample of `m` rows from a data frame, we can combine `[` and `sample()` as follows:

- random and non-contiguous sample of `m` rows from a data frame

```
df[sample(nrow(df), m), ]
#>                mpg cyl disp  hp drat    wt  qsec vs am gear
#> Valiant       18.1   6  225 105 2.76 3.460 20.22  1  0    3
#> Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4
#>               carb
#> Valiant          1
#> Mazda RX4 Wag    4
```

- random and contiguous sample of `m` rows from a data frame

```
# select a random starting position from available number of rows
start_row <- sample(nrow(df) - m + 1, size = 1)

# adjust ending position while avoiding off-by-one error
end_row <- start_row + m - 1

df[start_row:end_row, ]
#>               mpg cyl disp  hp drat    wt  qsec vs am gear
#> Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4
#> Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4
```

```
#>                carb
#> Mazda RX4        4
#> Mazda RX4 Wag    4
```

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

**A3.** we can sort columns in a data frame in the alphabetical order using `[` with `order()`:

```
# columns in original order
names(mtcars)
#>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"
#>  [9] "am"   "gear" "carb"

# columns in alphabetical order
names(mtcars[order(names(mtcars))])
#>  [1] "am"   "carb" "cyl"  "disp" "drat" "gear" "hp"   "mpg"
#>  [9] "qsec" "vs"   "wt"
```

## 4.4 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
```

```
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tibble      * 3.2.1   2023-03-20 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 5

# Control flow

## 5.1 Choices (Exercises 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 the rules about what a call to `ifelse()` might return:

- It is type unstable, i.e. the type of return will depend on the type of which condition is true (`yes` or `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 `test` is argument is of logical type, but `NA`, it will return `NA`:

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

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

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

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

This is also clarified in 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 conditional expressions in `if()` - even though of `numeric` type - can be successfully coerced to a `logical` type.

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

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

## 5.2 Loops (Exercises 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)` works in both positive and negative directions.

```
1:2
#> [1] 1 2
1:0
#> [1] 1 0
1:-3
#> [1]  1  0 -1 -2 -3
```

In this case, since `x` is of length `0`, `i` will go from `1` to `0`.

Additionally, since out-of-bound (OOB) value for atomic vectors is `NA`, all related operations with OOB values will also produce `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 is to use `seq_along()` instead:

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

for (i in seq_along(x)) {
  out[i] <- x[i]^2
}

out
#> list()
```

**Q2.** When the following code is evaluated, what can you say about the vector being iterated?

```r
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:

```r
xs <- c(1, 2, 3)
for (x in xs) {
  cat("x:", x, "\n")
  xs <- c(xs, x * 2)
  cat("xs:", paste(xs), "\n")
}
#> x: 1
#> xs: 1 2 3 2
#> x: 2
#> xs: 1 2 3 2 4
#> x: 3
#> xs: 1 2 3 2 4 6
```

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?

```r
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
```

## 5.3 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
```

```
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
 ↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 6

# Functions

Attaching the needed libraries:

```
library(tidyverse, warn.conflicts = FALSE)
```

## 6.1 Function fundamentals (Exercises 6.2.5)

**Q1.** Given a name, like `"mean"`, `match.fun()` lets you find a function. Given a function, can you find its name? Why doesn't that make sense in R?

**A1.** Given a name, `match.fun()` lets you find a function.

```
match.fun("mean")
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x5616db5e80c0>
#> <environment: namespace:base>
```

But, given a function, it doesn't make sense to find its name 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)
```

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

**Q2.** It's possible (although typically not useful) to call an anonymous function. Which of the two approaches below is correct? Why?

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

**A2.** The first expression is not correct since the function will evaluate `3()`, which is syntactically not allowed since literals can't be treated like functions.

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

rlang::is_syntactic_literal(3)
#> [1] TRUE
```

This is the correct way to call an anonymous function.

```r
g <- (function(x) 3)
g
#> function (x)
#> 3
g()
#> [1] 3
```

**Q3.** A good rule of thumb is that an anonymous function should fit on one line and shouldn't need to use `{}`. Review your code. Where could you have used an anonymous function instead of a named function? Where should you have used a named function instead of an anonymous function?

**A3.** Self activity.

**Q4.** What function allows you to tell if an object is a function? What function allows you to tell if a function is a primitive function?

**A4.** 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.** This code makes a list of all functions in the base package.

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

Use it to answer the following questions:

a. Which base function has the most arguments?
b. How many base functions have no arguments? What's special about those functions?
c. How could you adapt the code to find all primitive functions?

**A5.** The provided code is the following:

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

a. Which base function has the most arguments?

We can use `formals()` to extract number of arguments, but because this function returns `NULL` for primitive functions.

```r
formals("!")
#> NULL

length(formals("!"))
#> [1] 0
```

Therefore, we will focus only on non-primitive functions.

```r
funs <- purrr::discard(funs, is.primitive)
```

`scan()` function has the most arguments.

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

df_formals
#> # A tibble: 1,145 x 2
#>    `function`         argumentCount
#>    <chr>                      <int>
#>  1 scan                          22
#>  2 source                        17
#>  3 format.default                16
#>  4 formatC                       15
#>  5 library                       13
#>  6 merge.data.frame              13
#>  7 prettyNum                     13
#>  8 system2                       12
#>  9 system                        11
#> 10 all.equal.numeric             10
#> # i 1,135 more rows
```

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

At the time of writing, 47 base (non-primitive) functions have no arguments.

```
dplyr::filter(df_formals, argumentCount == 0)
#> # A tibble: 47 x 2
#>    `function`                argumentCount
#>    <chr>                             <int>
#>  1 .First.sys                            0
#>  2 .NotYetImplemented                    0
#>  3 .OptRequireMethods                    0
#>  4 .standard_regexps                     0
#>  5 .tryResumeInterrupt                   0
#>  6 closeAllConnections                   0
#>  7 contributors                          0
#>  8 Cstack_info                           0
#>  9 default.stringsAsFactors              0
#> 10 extSoftVersion                        0
#> # i 37 more rows
```

c. 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)

length(primitives)
#> [1] 210

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"                  "crossprod"
#>  [93] "cummax"                 "cummin"
#>  [95] "cumprod"                "cumsum"
#>  [97] "declare"                "digamma"
#>  [99] "dim"                    "dim<-"
#> [101] "dimnames"               "dimnames<-"
#> [103] "emptyenv"               "enc2native"
#> [105] "enc2utf8"               "environment<-"
#> [107] "Exec"                   "exp"
#> [109] "expm1"                  "expression"
#> [111] "floor"                  "for"
#> [113] "forceAndCall"           "function"
#> [115] "gamma"                  "gc.time"
#> [117] "globalenv"              "if"
#> [119] "Im"                     "interactive"
```

```
#> [121] "invisible"              "is.array"
#> [123] "is.atomic"              "is.call"
#> [125] "is.character"           "is.complex"
#> [127] "is.double"              "is.environment"
#> [129] "is.expression"          "is.finite"
#> [131] "is.function"            "is.infinite"
#> [133] "is.integer"             "is.language"
#> [135] "is.list"                "is.logical"
#> [137] "is.matrix"              "is.na"
#> [139] "is.name"                "is.nan"
#> [141] "is.null"                "is.numeric"
#> [143] "is.object"              "is.pairlist"
#> [145] "is.raw"                 "is.recursive"
#> [147] "is.single"              "is.symbol"
#> [149] "isS4"                   "lazyLoadDBfetch"
#> [151] "length"                 "length<-"
#> [153] "levels<-"               "lgamma"
#> [155] "list"                   "log"
#> [157] "log10"                  "log1p"
#> [159] "log2"                   "max"
#> [161] "min"                    "missing"
#> [163] "Mod"                    "names"
#> [165] "names<-"                "nargs"
#> [167] "next"                   "nzchar"
#> [169] "oldClass"               "oldClass<-"
#> [171] "on.exit"                "pos.to.env"
#> [173] "proc.time"              "prod"
#> [175] "quote"                  "range"
#> [177] "Re"                     "rep"
#> [179] "repeat"                 "retracemem"
#> [181] "return"                 "round"
#> [183] "seq_along"              "seq_len"
#> [185] "seq.int"                "sign"
#> [187] "signif"                 "sin"
#> [189] "sinh"                   "sinpi"
#> [191] "sqrt"                   "standardGeneric"
#> [193] "storage.mode<-"         "substitute"
#> [195] "sum"                    "switch"
#> [197] "Tailcall"               "tan"
#> [199] "tanh"                   "tanpi"
#> [201] "tcrossprod"             "tracemem"
#> [203] "trigamma"               "trunc"
#> [205] "unCfillPOSIXlt"         "unclass"
#> [207] "untracemem"             "UseMethod"
#> [209] "while"                  "xtfrm"
```

**Q6.** What are the three important components of a function?

**A6.** Except for primitive functions, all functions have 3 important components:

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

**Q7.** When does printing a function not show the environment it was created in?

**A7.** All package functions print their environment:

```
# base
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x5616db5e80c0>
#> <environment: namespace:base>

# other package function
purrr::map
#> function (.x, .f, ..., .progress = FALSE)
#> {
#>     map_("list", .x, .f, ..., .progress = .progress)
#> }
#> <bytecode: 0x5616ddc25e28>
#> <environment: namespace:purrr>
```

There are two exceptions where the enclosing environment won't be printed:

- 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 Lexical scoping (Exercises 6.4.5)

**Q1.** What does the following code return? Why? Describe how each of the three `c`'s is interpreted.

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

**A1.** In `c(c = c)`:

- first *c* is interpreted as a function call `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
```

You can also see this in the lexical analysis of this expression:

```
p_expr <- parse(text = "c(c = c)", keep.source = TRUE)
getParseData(p_expr) %>% select(token, text)
#>                   token text
#> 12                 expr     
#> 1  SYMBOL_FUNCTION_CALL    c
#> 3                  expr     
#> 2                   '('    (
#> 4            SYMBOL_SUB    c
#> 5                EQ_SUB    =
#> 6                SYMBOL    c
#> 8                  expr     
#> 7                   ')'    )
```

**Q2.** What are the four principles that govern how R looks for values?

**A2.** 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.** What does the following function return? Make a prediction before running the code yourself.

```
f <- function(x) {
  f <- function(x) {
    f <- function() {
      x^2
    }
    f() + 1
  }
  f(x) * 2
}
f(10)
```

**A3.** Correctly predicted

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

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

Although there are multiple `f()` functions, the order of evaluation goes from inside to outside with `x^2` evaluated first and `f(x) * 2` evaluated last. This results in 202 (= `((10 ^ 2) + 1) * 2`).

## 6.3 Lazy evaluation (Exercises 6.5.4)

**Q1.** What important property of `&&` makes `x_ok()` work?

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

```r
}

x_ok(NULL)
x_ok(1)
x_ok(1:3)
```

What is different with this code? Why is this behaviour undesirable here?

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

x_ok(NULL)
x_ok(1)
x_ok(1:3)
```

**A1.** `&&` evaluates left to right and has short-circuit evaluation, i.e., if the first operand is `TRUE`, R will short-circuit and not even look at the second operand.

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

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

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

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

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

**Q2.** What does this function return? Why? Which principle does it illustrate?

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

**A2.** The function returns `100` due to lazy evaluation.

When function execution environment encounters `x`, it evaluates argument `x = z` and since the name `z` is already bound to the value 100 in this environment, `x` is also bound to the same value.

We can check this by looking at the memory addresses:

```r
f2 <- function(x = z) {
  z <- 100
  print(lobstr::obj_addrs(list(x, z)))
  x
}

f2()
#> [1] "0x5616e162eb40" "0x5616e162eb40"
#> [1] 100
```

**Q3.** What does this function return? Why? Which principle does it illustrate?

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

**A3.** Let's first look at what the function returns:

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

This is because of name masking. In the function call `c(x, y)`, when `x` is accessed in the function environment, the following promise is evaluated in the function environment:

```
x <- {
  y <- 1
  2
}
```

And, thus `y` gets assigned to `1`, and `x` to `2`, since its the last value in that scope.

Therefore, neither the promise `y = 0` nor global assignment `y <- 10` is ever consulted to find the value for `y`.

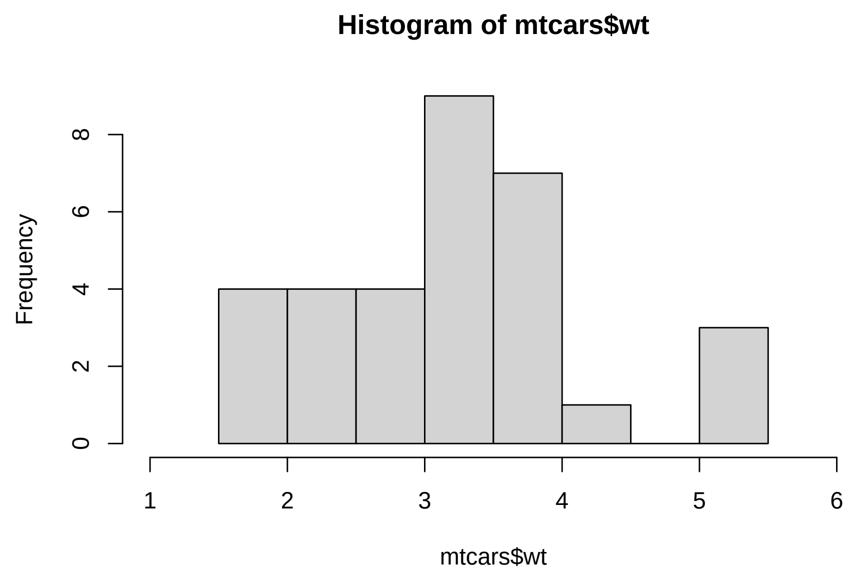
**Q4.** In `hist()`, the default value of `xlim` is `range(breaks)`, the default value for `breaks` is `"Sturges"`, and

```
range("Sturges")
#> [1] "Sturges" "Sturges"
```

Explain how `hist()` works to get a correct `xlim` value.

**A4.** The `xlim` defines the range of the histogram's `x`-axis.

```
hist(mtcars$wt, xlim = c(1, 6))
```




The default `xlim = range(breaks)` and `breaks = "Sturges"` arguments reveal that the function uses Sturges' algorithm to compute the number of breaks.

```
nclass.Sturges(mtcars$wt)
#> [1] 6
```

To see the implementation, run `sloop::s3_get_method("hist.default")`.

`hist()` ensures that the chosen algorithm returns a numeric vector containing at least two unique elements before `xlim` is computed.

**Q5.** Explain why this function works. Why is it confusing?

```
show_time <- function(x = stop("Error!")) {
  stop <- function(...) Sys.time()
  print(x)
}

show_time()
#> [1] "2025-03-16 00:14:43 UTC"
```

**A5.** Let's take this step-by-step.

The function argument `x` is missing in the function call. This means that `stop("Error!")` is evaluated in the function environment, and not global environment.

But, due to lazy evaluation, the promise `stop("Error!")` is evaluated only when `x` is accessed. This happens only when `print(x)` is called.

`print(x)` leads to `x` being evaluated, which evaluates `stop` in the function environment. But, in function environment, the `base::stop()` is masked by a locally defined `stop()` function, which returns `Sys.time()` output.

**Q6.** How many arguments are required when calling `library()`?

**A6.** Going solely by its signature,

```
formals(library)
#> $package
#>
#>
#> $help
#>
#>
#> $pos
#> [1] 2
#>
#> $lib.loc
#> NULL
#>
#> $character.only
#> [1] FALSE
#>
#> $logical.return
#> [1] FALSE
#>
#> $warn.conflicts
#>
#>
#> $quietly
#> [1] FALSE
#>
#> $verbose
#> getOption("verbose")
#>
#> $mask.ok
#>
#>
#> $exclude
#>
#>
#> $include.only
#>
#>
```

```
#> $attach.required
#> missing(include.only)
```

it looks like the following arguments are required:

```
formals(library) %>%
  purrr::discard(is.null) %>%
  purrr::map_lgl(~ .x == "") %>%
  purrr::keep(~ isTRUE(.x)) %>%
  names()
#> [1] "package"        "help"           "warn.conflicts"
#> [4] "mask.ok"        "exclude"        "include.only"
```

But, in reality, only one argument is required: `package`. The function internally checks if the other arguments are missing and adjusts accordingly.

It would have been better if there arguments were `NULL` instead of missing; that would avoid this confusion.

## 6.4 ... (dot-dot-dot) (Exercises 6.6.1)

**Q1.** Explain the following 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
```

**A1.** Let's look at arguments for these functions:

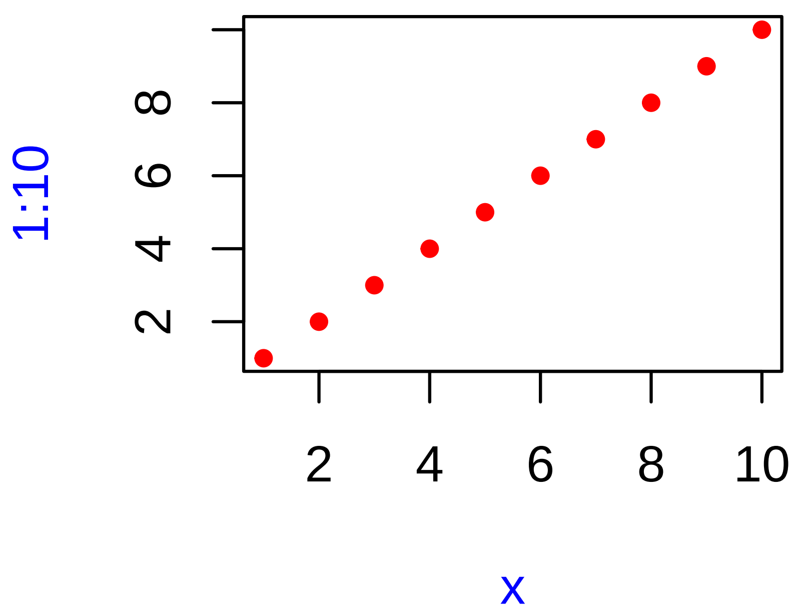
```
str(sum)
#> function (..., na.rm = FALSE)
str(mean)
#> function (x, ...)
```

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.** Explain how to find the documentation for the named arguments in the following function call:

```
plot(1:10, col = "red", pch = 20, xlab = "x", col.lab = "blue")
```




**A2.** Typing `?plot` in the console, we see its documentation, which also shows its signature:

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

Since `...` are passed to `par()`, we can look at `?par` docs:

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

And so on.

The docs for all parameters of interest reside there.

**Q3.** Why does `plot(1:10, col = "red")` only colour the points, not the axes or labels? Read the source code of `plot.default()` to find out.

**A3.** 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()
}
```

## 6.5 Exiting a function (Exercises 6.7.5)

**Q1.** What does `load()` return? Why don't you normally see these values?

**A1.** The `load()` function reloads datasets that were saved using the `save()` function:

```
save(iris, file = "my_iris.rda")
load("my_iris.rda")
```

We normally don't see any value because the function loads the datasets invisibly.

We can change this by setting `verbose = TRUE`:

```
load("my_iris.rda", verbose = TRUE)
#> Loading objects:
#>   iris

# cleanup
unlink("my_iris.rda")
```

**Q2.** What does `write.table()` return? What would be more useful?

**A2.** The `write.table()` writes a data frame to a file and returns a `NULL` invisibly.

```
write.table(BOD, file = "BOD.csv")
```

It would have been more helpful if the function invisibly returned the actual object being written to the file, which could then be further used.

```
# cleanup
unlink("BOD.csv")
```

**Q3.** How does the `chdir` parameter of `source()` compare to `with_dir()`? Why might you prefer one to the other?

**A3.** The `chdir` parameter of `source()` is described as:

> if `TRUE` and `file` is a pathname, the `R` working directory is temporarily changed to the directory containing file for evaluating

That is, `chdir` allows changing working directory temporarily but *only* to the directory containing file being sourced:

While `withr::with_dir()` temporarily changes the current working directory:

```
withr::with_dir
#> function (new, code)
#> {
#>     old <- setwd(dir = new)
#>     on.exit(setwd(old))
#>     force(code)
#> }
#> <bytecode: 0x5616e3f464e8>
#> <environment: namespace:withr>
```

More importantly, its parameters `dir` allows temporarily changing working directory to *any* directory.

**Q4.** Write a function that opens a graphics device, runs the supplied code, and closes the graphics device (always, regardless of whether or not the plotting code works).

**A4.** Here is a function that opens a graphics device, runs the supplied code, and closes the graphics device:

```
with_png_device <- function(filename, code, ...) {
  grDevices::png(filename = filename, ...)
  on.exit(grDevices::dev.off(), add = TRUE)

  force(code)
}
```

**Q5.** We can use `on.exit()` to implement a simple version of `capture.output()`.

```
capture.output2 <- function(code) {
  temp <- tempfile()
  on.exit(file.remove(temp), add = TRUE, after = TRUE)

  sink(temp)
  on.exit(sink(), add = TRUE, after = TRUE)

  force(code)
  readLines(temp)
}

capture.output2(cat("a", "b", "c", sep = "\n"))
#> [1] "a" "b" "c"
```

Compare `capture.output()` to `capture.output2()`. How do the functions differ? What features have I removed to make the key ideas easier to see? How have I rewritten the key ideas so they're easier to understand?

**A5.** The `capture.output()` is significantly more complex, as can be seen by its definition:

```
capture.output
#> function (..., file = NULL, append = FALSE, type = c("output",
 ↪
#>     "message"), split = FALSE)
#> {
#>     type <- match.arg(type)
```

```
#>     rval <- NULL
#>     closeit <- TRUE
#>     if (is.null(file))
#>         file <- textConnection("rval", "w", local = TRUE)
#>     else if (is.character(file))
#>         file <- file(file, if (append)
#>             "a"
#>         else "w")
#>     else if (inherits(file, "connection")) {
#>         if (!isOpen(file))
#>             open(file, if (append)
#>                 "a"
#>             else "w")
#>         else closeit <- FALSE
#>     }
#>     else stop("'file' must be NULL, a character string or a
↪  connection")
#>     sink(file, type = type, split = split)
#>     on.exit({
#>         sink(type = type, split = split)
#>         if (closeit) close(file)
#>     })
#>     for (i in seq_len(...length())) {
#>         out <- withVisible(...elt(i))
#>         if (out$visible)
#>             print(out$value)
#>     }
#>     on.exit()
#>     sink(type = type, split = split)
#>     if (closeit)
#>         close(file)
#>     rval %||% invisible(NULL)
#> }
#> <bytecode: 0x5616e44c5a68>
#> <environment: namespace:utils>
```

Here are few key differences:

- `capture.output()` uses `print()` function to print to console:

```
capture.output(1)
#> [1] "[1] 1"

capture.output2(1)
#> character(0)
```

- `capture.output()` can capture messages as well:

```
capture.output(message("Hi there!"), "a", type = "message")
#> Hi there!
#> [1] "a"
#> character(0)
```

- `capture.output()` takes into account visibility of the expression:

```
capture.output(1, invisible(2), 3)
#> [1] "[1] 1" "[1] 3"
```

## 6.6 Function forms (Exercises 6.8.6)

**Q1.** Rewrite the following code snippets into prefix form:

```
1 + 2 + 3

1 + (2 + 3)

if (length(x) <= 5) x[[5]] else x[[n]]
```

**A1.** Prefix forms for code snippets:

```
# The binary `+`  operator has left to right associative
↪  property.
`+`(`+`(1, 2), 3)

`+`(1, `(`(`+`(2, 3)))

`if`(cond = `<=`(length(x), 5), cons.expr = `[[`(x, 5), alt.expr
↪  = `[[`(x, n))
```

**Q2.** Clarify the following list of odd function calls:

```
x <- sample(replace = TRUE, 20, x = c(1:10, NA))
y <- runif(min = 0, max = 1, 20)
cor(m = "k", y = y, u = "p", x = x)
```

**A2.** These functions don't have dots (`...`) as parameters, so the argument matching takes place in the following steps:

- exact matching for named arguments
- partial matching
- position-based

**Q3.** Explain why the following code fails:

```
modify(get("x"), 1) <- 10
#> Error: target of assignment expands to non-language object
```

**A3.** As provided in the book, the replacement function is defined as:

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
```

Let's re-write the provided code in prefix format to understand why it doesn't work:

```
get("x") <- `modify<-`(x = get("x"), position = 1, value = 10)
```

Although this works:

```
x <- 5
`modify<-`(x = get("x"), position = 1, value = 10)
#> [1] 10
```

The following doesn't because the code above evaluates to:

```
`get<-`("x", 10)
#> Error in `get<-`("x", 10): could not find function "get<-"
```

And there is no `get<-` function in R.

**Q4.** Create a replacement function that modifies a random location in a vector.

**A4.** A replacement function that modifies a random location in a vector:

```
`random_modify<-` <- function(x, value) {
  random_index <- sample(seq_along(x), size = 1)
  x[random_index] <- value
  return(x)
}
```

Let's try it out:

```r
x1 <- rep("a", 10)
random_modify(x1) <- "X"
x1
#>  [1] "a" "a" "a" "a" "X" "a" "a" "a" "a" "a"

x2 <- rep("a", 10)
random_modify(x2) <- "Y"
x2
#>  [1] "a" "a" "a" "a" "a" "Y" "a" "a" "a" "a"

x3 <- rep(0, 15)
random_modify(x3) <- -4
x3
#>  [1]  0  0  0  0 -4  0  0  0  0  0  0  0  0  0  0

x4 <- rep(0, 15)
random_modify(x4) <- -1
x4
#>  [1]  0  0  0  0  0  0  0  0  0  0  0  0 -1  0  0
```

**Q5.** Write your own version of `+` that pastes its inputs together if they are character vectors but behaves as usual otherwise. In other words, make this code work:

```r
1 + 2
#> [1] 3

"a" + "b"
#> [1] "ab"
```

**A5.** Infix operator to re-create the desired output:

```r
`+` <- function(x, y) {
  if (is.character(x) || is.character(y)) {
    paste0(x, y)
  } else {
    base::`+`(x, y)
  }
}

1 + 2
#> [1] 3
```

```r
"a" + "b"
#> [1] "ab"

rm("+", envir = .GlobalEnv)
```

**Q6.** Create a list of all the replacement functions found in the base package. Which ones are primitive functions? (Hint: use `apropos()`.)

**A6.** Replacement functions always have `<-` at the end of their names.

So, using `apropos()`, we can find all replacement functions in search paths and the filter out the ones that don't belong to `{base}` package:

```r
ls_replacement <- apropos("<-", where = TRUE, mode = "function")

base_index <- which(grepl("base", searchpaths()))

ls_replacement <- ls_replacement[which(names(ls_replacement) ==
↪  as.character(base_index))]

unname(ls_replacement)
#>  [1] ".rowNamesDF<-"            "[[<-"
#>  [3] "[[<-.data.frame"          "[[<-.factor"
#>  [5] "[[<-.numeric_version"     "[[<-.POSIXlt"
#>  [7] "[<-"                      "[<-.data.frame"
#>  [9] "[<-.Date"                 "[<-.difftime"
#> [11] "[<-.factor"               "[<-.numeric_version"
#> [13] "[<-.POSIXct"              "[<-.POSIXlt"
#> [15] "@<-"                      "<-"
#> [17] "<<-"                      "$<-"
#> [19] "$<-.data.frame"           "$<-.POSIXlt"
#> [21] "attr<-"                   "attributes<-"
#> [23] "body<-"                   "class<-"
#> [25] "colnames<-"               "comment<-"
#> [27] "diag<-"                   "dim<-"
#> [29] "dimnames<-"               "dimnames<-.data.frame"
#> [31] "Encoding<-"               "environment<-"
#> [33] "formals<-"                "is.na<-"
#> [35] "is.na<-.default"          "is.na<-.factor"
#> [37] "is.na<-.numeric_version" "length<-"
#> [39] "length<-.Date"            "length<-.difftime"
#> [41] "length<-.factor"          "length<-.POSIXct"
#> [43] "length<-.POSIXlt"         "levels<-"
#> [45] "levels<-.factor"          "mode<-"
#> [47] "mostattributes<-"         "names<-"
#> [49] "names<-.POSIXlt"          "oldClass<-"
```

```
#> [51] "parent.env<-"            "regmatches<-"
#> [53] "row.names<-"             "row.names<-.data.frame"
#> [55] "row.names<-.default"     "rownames<-"
#> [57] "split<-"                 "split<-.data.frame"
#> [59] "split<-.default"         "storage.mode<-"
#> [61] "substr<-"                "substring<-"
#> [63] "units<-"                 "units<-.difftime"
```

The primitive replacement functions can be listed using `is.primitive()`:

```
mget(ls_replacement, envir = baseenv()) %>%
  purrr::keep(is.primitive) %>%
  names()
#>  [1] "[[<-"           "[<-"            "@<-"
#>  [4] "<-"             "<<-"            "$<-"
#>  [7] "attr<-"         "attributes<-"   "class<-"
#> [10] "dim<-"          "dimnames<-"     "environment<-"
#> [13] "length<-"       "levels<-"       "names<-"
#> [16] "oldClass<-"     "storage.mode<-"
```

**Q7.** What are valid names for user-created infix functions?

**A7.** As mentioned in the respective section of the book:

> The names of infix functions are more flexible than regular R functions: they can contain any sequence of characters except for `%`.

**Q8.** Create an infix `xor()` operator.

**A8.** Exclusive OR is a logical operation that is `TRUE` if and only if its arguments differ (one is `TRUE`, the other is `FALSE`).

```
lv1 <- c(TRUE, FALSE, TRUE, FALSE)
lv2 <- c(TRUE, TRUE, FALSE, FALSE)

xor(lv1, lv2)
#> [1] FALSE  TRUE  TRUE FALSE
```

We can create infix operator for exclusive OR like so:

```
`%xor%` <- function(x, y) {
  !((x & y) | !(x | y))
}
```

```
lv1 %xor% lv2
#> [1] FALSE  TRUE  TRUE FALSE

TRUE %xor% TRUE
#> [1] FALSE
```

The function is vectorized over its inputs because the underlying logical operators themselves are vectorized.

**Q9.** Create infix versions of the set functions `intersect()`, `union()`, and `setdiff()`. You might call them `%n%`, `%u%`, and `%/%` to match conventions from mathematics.

**A9.** The required infix operators can be created as following:

```
`%n%` <- function(x, y) {
  intersect(x, y)
}

`%u%` <- function(x, y) {
  union(x, y)
}

`%/%` <- function(x, y) {
  setdiff(x, y)
}
```

We can check that the outputs agree with the underlying functions:

```
(x <- c(sort(sample(1:20, 9)), NA))
#>  [1]  4  7  8  9 11 13 15 16 20 NA
(y <- c(sort(sample(3:23, 7)), NA))
#> [1]  9 10 13 15 17 19 20 NA

identical(intersect(x, y), x %n% y)
#> [1] TRUE
identical(union(x, y), x %u% y)
#> [1] TRUE
identical(setdiff(x, y), x %/% y)
#> [1] TRUE
```

## 6.7 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  colorspace    2.1-1   2024-07-26 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  dplyr       * 1.1.4   2023-11-17 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  forcats     * 1.0.0   2023-01-29 [1] RSPM
#>  generics      0.1.3   2022-07-05 [1] RSPM
#>  ggplot2     * 3.5.1   2024-04-23 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  grid          4.4.3   2025-02-28 [3] local
#>  gtable        0.3.6   2024-10-25 [1] RSPM
#>  hms           1.1.3   2023-03-21 [1] RSPM
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  lobstr        1.1.2   2022-06-22 [1] RSPM
#>  lubridate   * 1.9.4   2024-12-08 [1] RSPM
```

```
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  munsell       0.5.1   2024-04-01 [1] RSPM
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  purrr       * 1.0.4   2025-02-05 [1] RSPM
#>  R6            2.6.1   2025-02-15 [1] RSPM
#>  readr       * 2.1.5   2024-01-10 [1] RSPM
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  scales        1.3.0   2023-11-28 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr     * 1.5.1   2023-11-14 [1] RSPM
#>  tibble      * 3.2.1   2023-03-20 [1] RSPM
#>  tidyr       * 1.3.1   2024-01-24 [1] RSPM
#>  tidyselect    1.2.1   2024-03-11 [1] RSPM
#>  tidyverse   * 2.0.0   2023-02-22 [1] RSPM
#>  timechange    0.3.0   2024-01-18 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  tzdb          0.4.0   2023-05-12 [1] RSPM
#>  utf8          1.2.4   2023-10-22 [1] RSPM
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  withr         3.0.2   2024-10-28 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ------------------------------------------------------------
```

# Chapter 7

# Environments

Loading the needed libraries:

```r
library(rlang, warn.conflicts = FALSE)
```

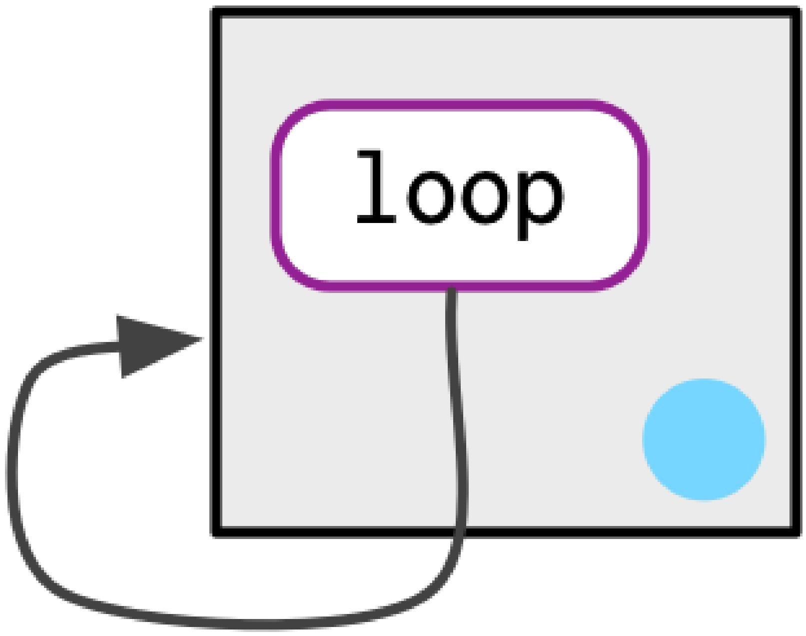
## 7.1 Environment basics (Exercises 7.2.7)

**Q1.** List three ways in which an environment differs from a list.

**A1.** As mentioned in the book, here are a few ways in which environments differ from lists:

| Property | List | Environment |
|---|---|---|
| semantics | value | reference |
| data structure | linear | non-linear |
| duplicated names | allowed | not allowed |
| can have parents? | false | true |
| can contain itself? | false | true |

**Q2.** Create an environment as illustrated by this picture.



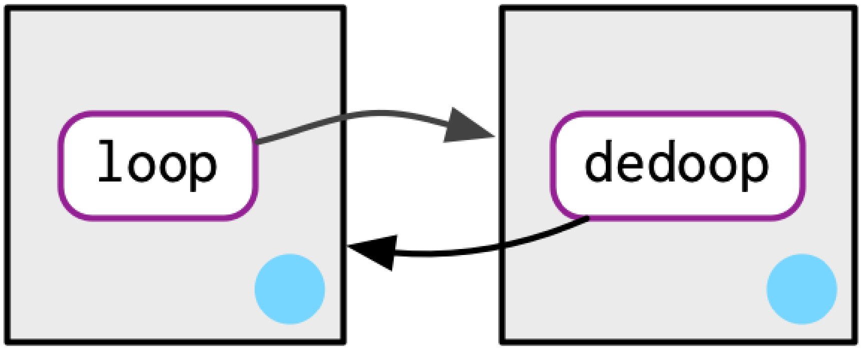

**A2.** Creating the environment illustrated in the picture:

```
library(rlang)

e <- env()
e$loop <- e
env_print(e)
#> <environment: 0x560c602f1048>
#> Parent: <environment: global>
#> Bindings:
#> * loop: <env>
```

The binding `loop` should have the same memory address as the environment `e`:

```
lobstr::ref(e$loop)
#>   [1:0x560c602f1048] <env>
#>   loop = [1:0x560c602f1048]
```

**Q3.** Create a pair of environments as illustrated by this picture.




**A3.** Creating the specified environment:

```
e1 <- env()
e2 <- env()

e1$loop <- e2
e2$deloop <- e1

# following should be the same
lobstr::obj_addrs(list(e1, e2$deloop))
#> [1] "0x560c64a1b388" "0x560c64a1b388"
lobstr::obj_addrs(list(e2, e1$loop))
#> [1] "0x560c64a63ff8" "0x560c64a63ff8"
```

**Q4.** Explain why `e[[1]]` and `e[c("a", "b")]` don't make sense when `e` is an environment.

**A4.** An environment is a non-linear data structure, and has no concept of ordered elements. Therefore, indexing it (e.g. `e[[1]]`) doesn't make sense.

Subsetting a list or a vector returns a subset of the underlying data structure. For example, subsetting a vector returns another vector. But it's unclear what subsetting an environment (e.g. `e[c("a", "b")]`) should return because there is no data structure to contain its returns. It can't be another environment since environments have reference semantics.

**Q5.** Create a version of `env_poke()` that will only bind new names, never re-bind old names. Some programming languages only do this, and are known as single assignment languages.

**A5.** Create a version of `env_poke()` that doesn't allow re-binding old names:

```r
env_poke2 <- function(env, nm, value) {
  if (env_has(env, nm)) {
    abort("Can't re-bind existing names.")
  }

  env_poke(env, nm, value)
}
```

Making sure that it behaves as expected:

```r
e <- env(a = 1, b = 2, c = 3)

# re-binding old names not allowed
env_poke2(e, "b", 4)
#> Error in `env_poke2()`:
#> ! Can't re-bind existing names.

# binding new names allowed
env_poke2(e, "d", 8)
e$d
#> [1] 8
```

Contrast this behavior with the following:

```r
e <- env(a = 1, b = 2, c = 3)

e$b
#> [1] 2

# re-binding old names allowed
env_poke(e, "b", 4)
e$b
#> [1] 4
```

**Q6.** What does this function do? How does it differ from `<<-` and why might you prefer it?

```r
rebind <- function(name, value, env = caller_env()) {
  if (identical(env, empty_env())) {
    stop("Can't find `", name, "`", call. = FALSE)
  } else if (env_has(env, name)) {
    env_poke(env, name, value)
  } else {
```

```
    rebind(name, value, env_parent(env))
  }
}
rebind("a", 10)
#> Error: Can't find `a`
a <- 5
rebind("a", 10)
a
#> [1] 10
```

**A6.** The downside of `<<-` is that it will create a new binding if it doesn't exist in the given environment, which is something that we may not wish:

```
# `x` doesn't exist
exists("x")
#> [1] FALSE

# so `<<-` will create one for us
{
  x <<- 5
}

# in the global environment
env_has(global_env(), "x")
#>    x
#> TRUE
x
#> [1] 5
```

But `rebind()` function will let us know if the binding doesn't exist, which is much safer:

```
rebind <- function(name, value, env = caller_env()) {
  if (identical(env, empty_env())) {
    stop("Can't find `", name, "`", call. = FALSE)
  } else if (env_has(env, name)) {
    env_poke(env, name, value)
  } else {
    rebind(name, value, env_parent(env))
  }
}

# doesn't exist
exists("abc")
```

```r
#> [1] FALSE

# so function will produce an error instead of creating it for us
rebind("abc", 10)
#> Error: Can't find `abc`

# but it will work as expected when the variable already exists
abc <- 5
rebind("abc", 10)
abc
#> [1] 10
```

## 7.2 Recursing over environments (Exercises 7.3.1)

**Q1.** Modify `where()` to return *all* environments that contain a binding for `name`. Carefully think through what type of object the function will need to return.

**A1.** Here is a modified version of `where()` that returns *all* environments that contain a binding for `name`.

Since we anticipate more than one environment, we dynamically update a list each time an environment with the specified binding is found. It is important to initialize to an empty list since that signifies that given binding is not found in any of the environments.

```r
where <- function(name, env = caller_env()) {
  env_list <- list()

  while (!identical(env, empty_env())) {
    if (env_has(env, name)) {
      env_list <- append(env_list, env)
    }

    env <- env_parent(env)
  }

  return(env_list)
}
```

Let's try it out:

```r
where("yyy")
#> list()

x <- 5
where("x")
#> [[1]]
#> <environment: R_GlobalEnv>

where("mean")
#> [[1]]
#> <environment: base>

library(dplyr, warn.conflicts = FALSE)
where("filter")
#> [[1]]
#> <environment: package:dplyr>
#> attr(,"name")
#> [1] "package:dplyr"
#> attr(,"path")
#> [1] "/home/runner/work/_temp/Library/dplyr"
#>
#> [[2]]
#> <environment: package:stats>
#> attr(,"name")
#> [1] "package:stats"
#> attr(,"path")
#> [1] "/opt/R/4.4.3/lib/R/library/stats"
detach("package:dplyr")
```

**Q2.** Write a function called `fget()` that finds only function objects. It should have two arguments, `name` and `env`, and should obey the regular scoping rules for functions: if there's an object with a matching name that's not a function, look in the parent. For an added challenge, also add an `inherits` argument which controls whether the function recurses up the parents or only looks in one environment.

**A2.** Here is a function that recursively looks for function objects:

```r
fget <- function(name, env = caller_env(), inherits = FALSE) {
  # we need only function objects
  f_value <- mget(name,
    envir = env,
    mode = "function",
    inherits = FALSE, # since we have our custom argument
    ifnotfound = list(NULL)
  )
```

```r
  if (!is.null(f_value[[1]])) {
    # success case
    f_value[[1]]
  } else {
    if (inherits && !identical(env, empty_env())) {
      # recursive case
      env <- env_parent(env)
      fget(name, env, inherits = TRUE)
    } else {
      # base case
      stop("No function objects with matching name was found.",
       ↪  call. = FALSE)
    }
  }
}
```

Let's try it out:

```r
fget("mean", inherits = FALSE)
#> Error: No function objects with matching name was found.

fget("mean", inherits = TRUE)
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x560c601f60c0>
#> <environment: namespace:base>

mean <- 5
fget("mean", inherits = FALSE)
#> Error: No function objects with matching name was found.

mean <- function() NULL
fget("mean", inherits = FALSE)
#> function ()
#> NULL
rm("mean")
```

## 7.3 Special environments (Exercises 7.4.5)

**Q1.** How is `search_envs()` different from `env_parents(global_env())`?

**A1.** The `search_envs()` lists a chain of environments currently attached to the search path and contains exported functions from these packages. The search path always ends at the `{base}` package environment. The search path also includes the global environment.

```
search_envs()
#>  [[1]] $ <env: global>
#>  [[2]] $ <env: package:rlang>
#>  [[3]] $ <env: package:magrittr>
#>  [[4]] $ <env: package:stats>
#>  [[5]] $ <env: package:graphics>
#>  [[6]] $ <env: package:grDevices>
#>  [[7]] $ <env: package:utils>
#>  [[8]] $ <env: package:datasets>
#>  [[9]] $ <env: package:methods>
#> [[10]] $ <env: Autoloads>
#> [[11]] $ <env: package:base>
```

The `env_parents()` lists all parent environments up until the empty environment. Of course, the global environment itself is not included in this list.

```
env_parents(global_env())
#>  [[1]] $ <env: package:rlang>
#>  [[2]] $ <env: package:magrittr>
#>  [[3]] $ <env: package:stats>
#>  [[4]] $ <env: package:graphics>
#>  [[5]] $ <env: package:grDevices>
#>  [[6]] $ <env: package:utils>
#>  [[7]] $ <env: package:datasets>
#>  [[8]] $ <env: package:methods>
#>  [[9]] $ <env: Autoloads>
#> [[10]] $ <env: package:base>
#> [[11]] $ <env: empty>
```

**Q2.** Draw a diagram that shows the enclosing environments of this function:

```
f1 <- function(x1) {
  f2 <- function(x2) {
    f3 <- function(x3) {
      x1 + x2 + x3
    }
    f3(3)
  }
  f2(2)
}
f1(1)
```

**A2.** I don't have access to the graphics software used to create diagrams in the book, so I am linking the diagram from the official solutions manual, where you will also find a more detailed description for the figure:

**Q3.** Write an enhanced version of `str()` that provides more information about functions. Show where the function was found and what environment it was defined in.

**A3.** To write the required function, we can first re-purpose the `fget()` function we wrote above to return the environment in which it was found and its enclosing environment:

```
fget2 <- function(name, env = caller_env()) {
  # we need only function objects
  f_value <- mget(name,
    envir = env,
    mode = "function",
    inherits = FALSE,
    ifnotfound = list(NULL)
  )

  if (!is.null(f_value[[1]])) {
    # success case
    list(
      "where" = env,
      "enclosing" = fn_env(f_value[[1]])
    )
  } else {
    if (!identical(env, empty_env())) {
      # recursive case
      env <- env_parent(env)
      fget2(name, env)
    } else {
      # base case
      stop("No function objects with matching name was found.",
       ↪  call. = FALSE)
    }
  }
}
```

Let's try it out:

```
fget2("mean")
#> $where
#> <environment: base>
#>
```

```
#> $enclosing
#> <environment: namespace:base>

mean <- function() NULL
fget2("mean")
#> $where
#> <environment: R_GlobalEnv>
#>
#> $enclosing
#> <environment: R_GlobalEnv>
rm("mean")
```

We can now write the new version of `str()` as a wrapper around this function. We only need to foresee that the users might enter the function name either as a symbol or a string.

```
str_function <- function(.f) {
  fget2(as_string(ensym(.f)))
}
```

Let's first try it with `base::mean()`:

```
str_function(mean)
#> $where
#> <environment: base>
#>
#> $enclosing
#> <environment: namespace:base>

str_function("mean")
#> $where
#> <environment: base>
#>
#> $enclosing
#> <environment: namespace:base>
```

And then with our variant present in the global environment:

```
mean <- function() NULL

str_function(mean)
#> $where
#> <environment: R_GlobalEnv>
```

```r
#>
#> $enclosing
#> <environment: R_GlobalEnv>

str_function("mean")
#> $where
#> <environment: R_GlobalEnv>
#>
#> $enclosing
#> <environment: R_GlobalEnv>

rm("mean")
```

## 7.4 Call stacks (Exercises 7.5.5)

**Q1.** Write a function that lists all the variables defined in the environment in which it was called. It should return the same results as `ls()`.

**A1.** Here is a function that lists all the variables defined in the environment in which it was called:

```r
# let's first remove everything that exists in the global
↪  environment right now
# to test with only newly defined objects
rm(list = ls())
rm(.Random.seed, envir = globalenv())

ls_env <- function(env = rlang::caller_env()) {
  sort(rlang::env_names(env))
}
```

The workhorse here is `rlang::caller_env()`, so let's also have a look at its definition:

```r
rlang::caller_env
#> function (n = 1)
#> {
#>     parent.frame(n + 1)
#> }
#> <bytecode: 0x560c5ff354c0>
#> <environment: namespace:rlang>
```

Let's try it out:

- In global environment:

```
x <- "a"
y <- 1

ls_env()
#> [1] "ls_env" "x"      "y"

ls()
#> [1] "ls_env" "x"      "y"
```

- In function environment:

```
foo <- function() {
  a <- "x"
  b <- 2

  print(ls_env())

  print(ls())
}

foo()
#> [1] "a" "b"
#> [1] "a" "b"
```

## 7.5 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
 ↪  rmarkdown)
```

```
#>  quarto   NA
#>
#> - Packages -------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  crayon        1.5.3   2024-06-20 [1] RSPM
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  dplyr         1.1.4   2023-11-17 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  generics      0.1.3   2022-07-05 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  lobstr        1.1.2   2022-06-22 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  R6            2.6.1   2025-02-15 [1] RSPM
#>  rlang       * 1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tibble        3.2.1   2023-03-20 [1] RSPM
#>  tidyselect    1.2.1   2024-03-11 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
```

```
#>
#> ------------------------------------------------------------
```

# Chapter 8

# Conditions

Attaching the needed libraries:

```
library(rlang, warn.conflicts = FALSE)
library(testthat, warn.conflicts = FALSE)
```

## 8.1 Signalling conditions (Exercises 8.2.4)

---

**Q1.** Write a wrapper around `file.remove()` that throws an error if the file to be deleted does not exist.

**A1.** Let's first create a wrapper function around `file.remove()` that throws an error if the file to be deleted does not exist.

```
fileRemove <- function(...) {
  existing_files <- fs::file_exists(...)

  if (!all(existing_files)) {
    stop(
      cat(
        "The following files to be deleted don't exist:",
        names(existing_files[!existing_files]),
        sep = "\n"
      ),
      call. = FALSE
    )
```

```r
  }

  file.remove(...)
}
```

Let's first create a file that we can delete immediately.

```r
fs::file_create("random.R")
```

The function should fail if there are any other files provided that don't exist:

```r
fileRemove(c("random.R", "XYZ.csv"))
#> The following files to be deleted don't exist:
#> XYZ.csv
#> Error:
```

But it does work as expected when the file exists:

```r
fileRemove("random.R")
#> [1] TRUE
```

---

**Q2.** What does the `appendLF` argument to `message()` do? How is it related to `cat()`?

**A2.** As mentioned in the docs for `message()`, `appendLF` argument decides:

> should messages given as a character string have a newline appended?

- If `TRUE` (default value), a final newline is regarded as part of the message:

```r
foo <- function(appendLF) {
  message("Beetle", appendLF = appendLF)
  message("Juice", appendLF = appendLF)
}

foo(appendLF = TRUE)
#> Beetle
#> Juice
```

- If `FALSE`, messages will be concatenated:

```
foo <- function(appendLF) {
  message("Beetle", appendLF = appendLF)
  message("Juice", appendLF = appendLF)
}

foo(appendLF = FALSE)
#> BeetleJuice
```

On the other hand, `cat()` converts its arguments to character vectors and concatenates them to a single character vector by default:

```
foo <- function() {
  cat("Beetle")
  cat("Juice")
}

foo()
#> BeetleJuice
```

In order to get `message()`-like default behavior for outputs, we can set `sep = "\n"`:

```
foo <- function() {
  cat("Beetle", sep = "\n")
  cat("Juice", sep = "\n")
}

foo()
#> Beetle
#> Juice
```

---

## 8.2 Handling conditions (Exercises 8.4.5)

---

**Q1.** What extra information does the condition generated by `abort()` contain compared to the condition generated by `stop()` i.e. what's the difference between these two objects? Read the help for `?abort` to learn more.

```r
catch_cnd(stop("An error"))
catch_cnd(abort("An error"))
```

**A1.** Compared to `base::stop()`, `rlang::abort()` contains two additional pieces of information:

- `trace`: A traceback capturing the sequence of calls that lead to the current function
- `parent`: Information about another condition used as a parent to create a chained condition.

```r
library(rlang)

stopInfo <- catch_cnd(stop("An error"))
abortInfo <- catch_cnd(abort("An error"))

str(stopInfo)
#> List of 2
#>  $ message: chr "An error"
#>  $ call   : language force(expr)
#>  - attr(*, "class")= chr [1:3] "simpleError" "error"
↪  "condition"

str(abortInfo)
#> List of 5
#>  $ message: chr "An error"
#>  $ trace  :Classes 'rlang_trace', 'rlib_trace', 'tbl' and
↪  'data.frame':  8 obs. of  6 variables:
#>   ..$ call       :List of 8
#>   .. ..$ : language catch_cnd(abort("An error"))
#>   .. ..$ : language
↪  eval_bare(rlang::expr(tryCatch(!!!handlers, {     force(expr)
↪  ...
#>   .. ..$ : language tryCatch(condition = `<fn>`, {
↪  force(expr) ...
#>   .. ..$ : language tryCatchList(expr, classes, parentenv,
↪  handlers)
#>   .. ..$ : language tryCatchOne(expr, names, parentenv,
↪  handlers[[1L]])
#>   .. ..$ : language doTryCatch(return(expr), name, parentenv,
↪  handler)
#>   .. ..$ : language force(expr)
#>   .. ..$ : language abort("An error")
#>   ..$ parent     : int [1:8] 0 1 1 3 4 5 1 0
```

```
#>   ..$ visible    : logi [1:8] FALSE FALSE FALSE FALSE FALSE
↪  FALSE ...
#>   ..$ namespace  : chr [1:8] "rlang" "rlang" "base" "base" ...
#>   ..$ scope      : chr [1:8] "::" "::" "::" "local" ...
#>   ..$ error_frame: logi [1:8] FALSE FALSE FALSE FALSE FALSE
↪  FALSE ...
#>   ..- attr(*, "version")= int 2
#>  $ parent : NULL
#>  $ rlang  :List of 1
#>   ..$ inherit: logi TRUE
#>  $ call   : NULL
#>  - attr(*, "class")= chr [1:3] "rlang_error" "error"
↪  "condition"
```

---

**Q2.** Predict the results of evaluating the following code

```
show_condition <- function(code) {
  tryCatch(
    error = function(cnd) "error",
    warning = function(cnd) "warning",
    message = function(cnd) "message",
    {
      code
      NULL
    }
  )
}

show_condition(stop("!"))
show_condition(10)
show_condition(warning("?!"))
show_condition({
  10
  message("?")
  warning("?!")
})
```

**A2.** Correctly predicted

The first three pieces of code are straightforward:

```r
show_condition <- function(code) {
  tryCatch(
    error = function(cnd) "error",
    warning = function(cnd) "warning",
    message = function(cnd) "message",
    {
      code
      NULL
    }
  )
}

show_condition(stop("!"))
#> [1] "error"
show_condition(10)
#> NULL
show_condition(warning("?!"))
#> [1] "warning"
```

The last piece of code is the challenging one and it illustrates how `tryCatch()` works. From its docs:

> When several handlers are supplied in a single tryCatch then the first one is considered more recent than the second.

```r
show_condition({
  10
  message("?")
  warning("?!")
})
#> [1] "message"
```

---

**Q3.** Explain the results of running this code:

```r
withCallingHandlers(
  message = function(cnd) message("b"),
  withCallingHandlers(
    message = function(cnd) message("a"),
    message("c")
  )
)
```

```
#> b
#> a
#> b
#> c
```

**A3.** The surprising part of this output is the `b` before the last `c`.

This happens because the inner calling handler doesn't handle the message, so it bubbles up to the outer calling handler.

---

**Q4.** Read the source code for `catch_cnd()` and explain how it works.

**A4.** Let's look at the source code for `catch_cnd()`:

```
rlang::catch_cnd
#> function (expr, classes = "condition")
#> {
#>     stopifnot(is_character(classes))
#>     handlers <- rep_named(classes, list(identity))
#>     eval_bare(rlang::expr(tryCatch(!!!handlers, {
#>         force(expr)
#>         return(NULL)
#>     })))
#> }
#> <bytecode: 0x55bf7d05dfb8>
#> <environment: namespace:rlang>
```

As mentioned in the function docs:

> This is a small wrapper around `tryCatch()` that captures any condition signalled while evaluating its argument.

The `classes` argument allows a character vector of condition classes to catch, and the complex tidy evaluation generates the necessary condition (if there is any; otherwise `NULL`).

```
catch_cnd(10)
#> NULL

catch_cnd(abort(message = "an error", class = "class1"))
#> <error/class1>
#> Error:
```

```
#> ! an error
#> ---
#> Backtrace:
#> x
```

---

**Q5.** How could you rewrite `show_condition()` to use a single handler?

**A5.** The source code for `rlang::catch_cond()` gives us a clue as to how we can do this.

Conditions also have a `class` attribute, and we can use it to determine which handler will match the condition.

```
show_condition2 <- function(code) {
  tryCatch(
    condition = function(cnd) {
      if (inherits(cnd, "error")) {
        return("error")
      }
      if (inherits(cnd, "warning")) {
        return("warning")
      }
      if (inherits(cnd, "message")) {
        return("message")
      }
    },
    {
      code
      NULL
    }
  )
}
```

Let's try this new version with the examples used for the original version:

```
show_condition2(stop("!"))
#> [1] "error"
show_condition2(10)
#> NULL
show_condition2(warning("?!"))
#> [1] "warning"
show_condition2({
  10
```

```r
  message("?")
  warning("?!")
})
#> [1] "message"
```

---

## 8.3 Custom conditions (Exercises 8.5.4)

---

**Q1.** Inside a package, it's occasionally useful to check that a package is installed before using it. Write a function that checks if a package is installed (with `requireNamespace("pkg", quietly = FALSE))` and if not, throws a custom condition that includes the package name in the metadata.

**A1.** Here is the desired function:

```r
abort_missing_package <- function(pkg) {
  msg <- glue::glue("Problem loading `{pkg}` package, which is
↪  missing and must be installed.")

  abort("error_missing_package",
    message = msg,
    pkg = pkg
  )
}

check_if_pkg_installed <- function(pkg) {
  if (!requireNamespace(pkg, quietly = TRUE)) {
    abort_missing_package(pkg)
  }

  TRUE
}

check_if_pkg_installed("xyz123")
#> Error in `abort_missing_package()`:
#> ! Problem loading `xyz123` package, which is missing and must
↪  be installed.
check_if_pkg_installed("dplyr")
#> [1] TRUE
```

For a reference, also see the source code for following functions:

- `rlang::is_installed()`
- `insight::check_if_installed()`

---

**Q2.** Inside a package you often need to stop with an error when something is not right. Other packages that depend on your package might be tempted to check these errors in their unit tests. How could you help these packages to avoid relying on the error message which is part of the user interface rather than the API and might change without notice?

**A2.** As an example, let's say that another package developer wanted to use the `check_if_pkg_installed()` function that we just wrote.

So the developer using it in their own package can write a unit test like this:

```
expect_error(
  check_if_pkg_installed("xyz123"),
  "Problem loading `xyz123` package, which is missing and must be
   ↪  installed."
)
```

To dissuade developers from having to rely on error messages to check for errors, we can instead provide a custom condition, which can be used for unit testing instead:

```
e <- catch_cnd(check_if_pkg_installed("xyz123"))

inherits(e, "error_missing_package")
#> [1] TRUE
```

So that the unit test could be:

```
expect_s3_class(e, "error_missing_package")
```

This test wouldn't fail even if we decided to change the exact message.

---

## 8.4 Applications (Exercises 8.6.6)

---

**Q1.** Create `suppressConditions()` that works like `suppressMessages()` and `suppressWarnings()` but suppresses everything. Think carefully about how you should handle errors.

**A1.** To create the desired `suppressConditions()`, we just need to create an equivalent of `suppressWarnings()` and `suppressMessages()` for errors. To suppress the error message, we can handle errors within a `tryCatch()` and return the error object invisibly:

```r
suppressErrors <- function(expr) {
  tryCatch(
    error = function(cnd) invisible(cnd),
    expr
  )
}

suppressConditions <- function(expr) {
  suppressErrors(suppressWarnings(suppressMessages(expr)))
}
```

Let's try out and see if this works as expected:

```r
suppressConditions(1)
#> [1] 1

suppressConditions({
  message("I'm messaging you")
  warning("I'm warning you")
})

suppressConditions({
  stop("I'm stopping this")
})
```

All condition messages are now suppressed, but note that if we assign error object to a variable, we can still extract useful information for debugging:

```r
e <- suppressConditions({
  stop("I'm stopping this")
})

e
#> <simpleError in withCallingHandlers(expr, message =
↪  function(c) if (inherits(c,     classes))
↪  tryInvokeRestart("muffleMessage")): I'm stopping this>
```

---

**Q2.** Compare the following two implementations of `message2error()`. What is the main advantage of `withCallingHandlers()` in this scenario? (Hint: look carefully at the traceback.)

```r
message2error <- function(code) {
  withCallingHandlers(code, message = function(e) stop(e))
}
message2error <- function(code) {
  tryCatch(code, message = function(e) stop(e))
}
```

**A2.** With `withCallingHandlers()`, the condition handler is called from the signaling function itself, and, therefore, provides a more detailed call stack.

```r
message2error1 <- function(code) {
  withCallingHandlers(code, message = function(e) stop("error"))
}

message2error1({
  1
  message("hidden error")
  NULL
})
#> Error in (function (e) : error

traceback()
#> 9: stop("error") at #2
#> 8: (function (e)
#>    stop("error"))(list(message = "hidden error\n",
#>      call = message("hidden error")))
#> 7: signalCondition(cond)
#> 6: doWithOneRestart(return(expr), restart)
#> 5: withOneRestart(expr, restarts[[1L]])
#> 4: withRestarts({
#>        signalCondition(cond)
#>        defaultHandler(cond)
#>    }, muffleMessage = function() NULL)
#> 3: message("hidden error") at #1
#> 2: withCallingHandlers(code,
#>      message = function(e) stop("error")) at #2
#> 1: message2error1({
#>        1
#>        message("hidden error")
```

```
#>         NULL
#>     })
```

With `tryCatch()`, the signalling function terminates when a condition is raised, and so it doesn't provide as detailed call stack.

```
message2error2 <- function(code) {
  tryCatch(code, message = function(e) (stop("error")))
}

message2error2({
  1
  stop("hidden error")
  NULL
})
#> Error in value[[3L]](cond) : error

traceback()
#> 6: stop("error") at #2
#> 5: value[[3L]](cond)
#> 4: tryCatchOne(expr, names, parentenv, handlers[[1L]])
#> 3: tryCatchList(expr, classes, parentenv, handlers)
#> 2: tryCatch(code, message = function(e) (stop("error"))) at #2
#> 1: message2error2({
#>         1
#>         message("hidden error")
#>         NULL
#>     })
```

---

**Q3.** How would you modify the `catch_cnds()` definition if you wanted to recreate the original intermingling of warnings and messages?

**A3.** Actually, you won't have to modify anything about the function defined in the chapter, since it supports this out of the box.

So nothing additional to do here[1]!

```
catch_cnds <- function(expr) {
  conds <- list()
  add_cond <- function(cnd) {
    conds <<- append(conds, list(cnd))
```

[1]The best kind of exercise there is!

```r
    cnd_muffle(cnd)
  }

  withCallingHandlers(
    message = add_cond,
    warning = add_cond,
    expr
  )

  conds
}

catch_cnds({
  inform("a")
  warn("b")
  inform("c")
})
#> [[1]]
#> <message/rlang_message>
#> Message:
#> a
#>
#> [[2]]
#> <warning/rlang_warning>
#> Warning:
#> b
#>
#> [[3]]
#> <message/rlang_message>
#> Message:
#> c
```

---

**Q4.** Why is catching interrupts dangerous? Run this code to find out.

```r
bottles_of_beer <- function(i = 99) {
  message(
    "There are ",
    i,
    " bottles of beer on the wall, ",
    i,
    " bottles of beer."
  )
  while (i > 0) {
```

```r
    tryCatch(
      Sys.sleep(1),
      interrupt = function(err) {
        i <<- i - 1
        if (i > 0) {
          message(
            "Take one down, pass it around, ",
            i,
            " bottle",
            if (i > 1) "s",
            " of beer on the wall."
          )
        }
      }
    )
  }
  message(
    "No more bottles of beer on the wall, ",
    "no more bottles of beer."
  )
}
```

**A4.** Because this function catches the `interrupt` and there is no way to stop `bottles_of_beer()`, because the way you would usually stop it by using `interrupt`!

```r
bottles_of_beer()
#> There are 99 bottles of beer on the wall, 99 bottles of beer.
#> Take one down, pass it around, 98 bottles of beer on the wall.
#> Take one down, pass it around, 97 bottles of beer on the wall.
#> Take one down, pass it around, 96 bottles of beer on the wall.
#> Take one down, pass it around, 95 bottles of beer on the wall.
#> Take one down, pass it around, 94 bottles of beer on the wall.
#> Take one down, pass it around, 93 bottles of beer on the wall.
#> Take one down, pass it around, 92 bottles of beer on the wall.
#> Take one down, pass it around, 91 bottles of beer on the wall.
#> ...
```

In RStudio IDE, you can snap out of this loop by terminating the R session.

This shows why catching `interrupt` is dangerous and can result in poor user experience.

## 8.5 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  brio          1.1.5   2024-04-24 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  desc          1.4.3   2023-12-10 [1] RSPM
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  dplyr         1.1.4   2023-11-17 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  fs            1.6.5   2024-10-30 [1] RSPM
#>  generics      0.1.3   2022-07-05 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  pkgload       1.4.0   2024-06-28 [1] RSPM
```

```
#>  R6            2.6.1   2025-02-15 [1] RSPM
#>  rlang       * 1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  rprojroot     2.0.4   2023-11-05 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  testthat    * 3.2.3   2025-01-13 [1] RSPM
#>  tibble        3.2.1   2023-03-20 [1] RSPM
#>  tidyselect    1.2.1   2024-03-11 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  withr         3.0.2   2024-10-28 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 9

# Functionals

Attaching the needed libraries:

```
library(purrr, warn.conflicts = FALSE)
```

## 9.1 My first functional: `map()` (Exercises 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.** Let's handle the two parts of the question separately.

- `as_mapper()` and `{purrr}`-generated anonymous functions:

Looking at the experimentation below with `map()` and `as_mapper()`, we can see that, depending on the type of the input, `as_mapper()` creates an extractor function using `pluck()`.

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

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

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

```r
#>
#> [[2]]
#> [1] 2
as_mapper(1)
#> function (x, ...)
#> pluck_raw(x, list(1), .default = NULL)
#> <environment: 0x5630c955d0a0>

map(x, list(2, 1))
#> [[1]]
#> NULL
#>
#> [[2]]
#> [1] 3
as_mapper(list(2, 1))
#> function (x, ...)
#> pluck_raw(x, list(2, 1), .default = NULL)
#> <environment: 0x5630c9713f40>

# 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_raw(x, list("m"), .default = NULL)
#> <environment: 0x5630c9b179a8>

# mixing position and name
map(y, list(2, "m"))
#> [[1]]
#> [1] "mo"
#>
#> [[2]]
#> NULL
as_mapper(list(2, "m"))
#> function (x, ...)
```

```
#> pluck_raw(x, list(2, "m"), .default = NULL)
#> <environment: 0x5630c9fcd8d8>

# 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, and these are not valid accessors and so we get `NULL`s in return.

```
map(1:3, ~ runif(2))
#> [[1]]
#> [1] 0.2180892 0.9876342
#>
#> [[2]]
#> [1] 0.3484619 0.3810470
#>
#> [[3]]
#> [1] 0.02098596 0.74972687
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]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
as_mapper(runif(2))
#> function (x, ...)
#> pluck_raw(x, list(0.597890264587477, 0.587997315218672),
↪  .default = NULL)
#> <environment: 0x5630cb2d8670>
```

---

**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.** Using the appropriate `map()` function to:

- 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         11         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.**

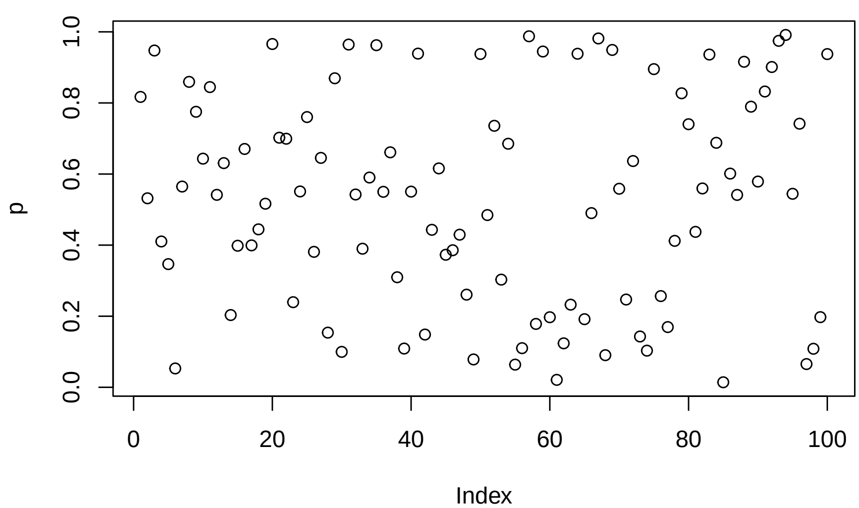
- Extract the *p*-value from each test:

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

(p <- map_dbl(trials, "p.value"))
#>   [1] 0.81695628 0.53177360 0.94750819 0.41026769 0.34655294
#>   [6] 0.05300287 0.56479901 0.85936864 0.77517391 0.64321161
#>  [11] 0.84462914 0.54144946 0.63070476 0.20325827 0.39824435
#>  [16] 0.67052432 0.39932663 0.44437632 0.51645941 0.96578745
#>  [21] 0.70219557 0.69931716 0.23946786 0.55100566 0.76028958
#>  [26] 0.38105366 0.64544126 0.15379307 0.86945196 0.09965658
#>  [31] 0.96425489 0.54239108 0.38985789 0.59019282 0.96247907
#>  [36] 0.54997487 0.66111391 0.30961551 0.10897334 0.55049635
#>  [41] 0.93882405 0.14836866 0.44307287 0.61583610 0.37284284
#>  [46] 0.38559622 0.42935767 0.26059293 0.07831619 0.93768396
```

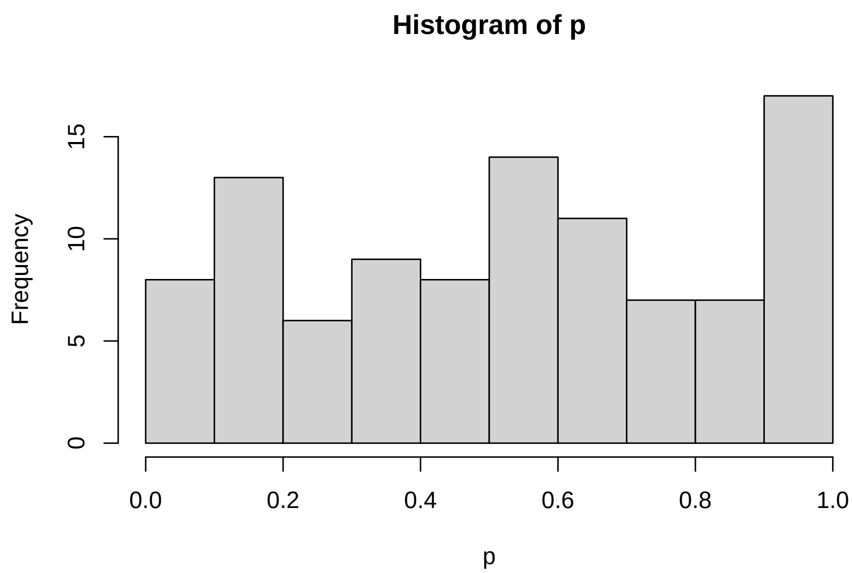
```
#>  [51] 0.48459268 0.73571291 0.30288560 0.68521609 0.06374636
#>  [56] 0.11007808 0.98758443 0.17831882 0.94471538 0.19711729
#>  [61] 0.02094185 0.12370745 0.23247837 0.93842382 0.19160550
#>  [66] 0.49005550 0.98146240 0.09034183 0.94912080 0.55857523
#>  [71] 0.24692070 0.63658206 0.14290966 0.10309770 0.89516449
#>  [76] 0.25660092 0.16943034 0.41199780 0.82721280 0.74017418
#>  [81] 0.43724631 0.55944024 0.93615100 0.68788872 0.01416627
#>  [86] 0.60120497 0.54125910 0.91581929 0.78949327 0.57887371
#>  [91] 0.83217542 0.90108906 0.97474727 0.99129282 0.54436155
#>  [96] 0.74159859 0.06534957 0.10834529 0.19737786 0.93750342
```

- Visualise the extracted $p$-values:

```
plot(p)
```




```
hist(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 `map()`:
#> i In index: 1.
#> Caused by error in `.f()`:
#> ! unused argument (function (.x, .f, ..., .progress = FALSE)
#> {
#>     map_("list", .x, .f, ..., .progress = .progress)
#> })
```

**A5.** This function fails because this call effectively evaluates to the following:

```
map(.x = x, .f = ~ triple(x = .x, map))
```

But `triple()` has only one parameter (`x`), and so the execution fails.

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(.x, ~ triple(.x)))
#> [[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:

```r
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))

bootstraps %>%
  map(~ lm(mpg ~ disp, data = .x)) %>%
  map(summary) %>%
  map_dbl("r.squared")
#>  [1] 0.7864562 0.8110818 0.7956331 0.7632399 0.7967824
#>  [6] 0.7364226 0.7203027 0.6653252 0.7732780 0.6753329
```

---

## 9.2 Map variants (Exercises 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 frame with the same dimensions with the plucked element recycled across rows.

```
head(modify(mtcars, 1))
#>   mpg cyl disp  hp drat   wt  qsec vs am gear carb
#> 1  21   6  160 110  3.9 2.62 16.46  0  1    4    4
#> 2  21   6  160 110  3.9 2.62 16.46  0  1    4    4
#> 3  21   6  160 110  3.9 2.62 16.46  0  1    4    4
#> 4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
#> 5  21   6  160 110  3.9 2.62 16.46  0  1    4    4
#> 6  21   6  160 110  3.9 2.62 16.46  0  1    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.** Let's first rewrite provided code using `iwalk()`:

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

The advantage of using `iwalk()` is that we need to now deal with only a single variable (`cyls`) instead of two (`cyls` and `paths`).

The disadvantage is that the code is difficult to reason about: In `walk2()`, it's explicit what `.x` (`= cyls`) and `.y` (`= paths`) correspond to, while this is not so for `iwalk()` (i.e., `.x = cyls` and `.y = names(cyls)`) with the `.y` argument being "invisible".

---

**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 stored in `trans` as anonymous functions via placeholder `f`, while the names of the columns specified in `mtcars[nm]` are supplied as `var` argument to 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]]))
```

The latter approach can't afford passing arguments to placeholders in an anonymous function.

---

**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 console for every list element.

```
withr::with_tempdir(
  code = {
    ls <- split(mtcars, mtcars$cyl)
    nm <- names(ls)
    map2(ls, nm, write.csv)
  }
)
#> $`4`
#> NULL
#>
```

```
#> $`6`
#> NULL
#>
#> $`8`
#> NULL
```

---

## 9.3 Predicate functionals (Exercises 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.

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

**A2.** The supplied function struggles with inputs of length 0 and 1 because function tries to subscript out-of-bound values.

```r
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:

```r
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:

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

Depending on the function, we can provide a different `init` argument:

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

---

**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.)

**A3.** Implementation of `span()`:

```r
span <- function(x, f) {
  running_lengths <- purrr::map_lgl(x, ~ f(.x)) %>% rle()

  df <- dplyr::tibble(
    "lengths" = running_lengths$lengths,
    "values" = running_lengths$values
  ) %>%
    dplyr::mutate(rowid = dplyr::row_number()) %>%
    dplyr::filter(values)

  # no sequence where condition is `TRUE`
  if (nrow(df) == 0L) {
    return(integer())
  }

  # only single sequence where condition is `TRUE`
  if (nrow(df) == 1L) {
    return((df$rowid):(df$lengths - 1 + df$rowid))
  }
```

```r
  # multiple sequences where condition is `TRUE`; select max one
  if (nrow(df) > 1L) {
    df <- dplyr::filter(df, lengths == max(lengths))
    return((df$rowid):(df$lengths - 1 + df$rowid))
  }
}
```

Testing it once:

```r
span(c(0, 0, 0, 0, 0), is.na)
#> integer(0)
span(c(NA, 0, NA, NA, NA), is.na)
#> [1] 3 4 5
span(c(NA, 0, 0, 0, 0), is.na)
#> [1] 1
span(c(NA, NA, 0, 0, 0), is.na)
#> [1] 1 2
```

Testing it twice:

```r
span(c(3, 1, 2, 4, 5, 6), function(x) x > 3)
#> [1] 2 3 4
span(c(3, 1, 2, 4, 5, 6), function(x) x > 9)
#> integer(0)
span(c(3, 1, 2, 4, 5, 6), function(x) x == 3)
#> [1] 1
span(c(3, 1, 2, 4, 5, 6), function(x) x %in% c(2, 4))
#> [1] 2 3
```

---

**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.

**A4.** Here are implementations for the specified functions:

- Implementing `arg_max()`

```
arg_max <- function(.x, .f) {
  df <- dplyr::tibble(
    original = .x,
    transformed = purrr::map_dbl(.x, .f)
  )

  dplyr::filter(df, transformed ==
↪   max(transformed))[["original"]]
}

arg_max(-10:5, function(x) x^2)
#> [1] -10
arg_max(-5:5, function(x) x^2)
#> [1] -5  5
```

- Implementing `arg_min()`

```
arg_min <- function(.x, .f) {
  df <- dplyr::tibble(
    original = .x,
    transformed = purrr::map_dbl(.x, .f)
  )

  dplyr::filter(df, transformed ==
↪   min(transformed))[["original"]]
}

arg_min(-10:5, function(x) x^2)
#> [1] 0
arg_min(-5:5, function(x) x^2)
#> [1] 0
```

---

**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])
}
```

**A5.** We will use `{purrr}` package to apply this function. Key thing to keep in mind is that a data frame is a list of atomic vectors of equal length.

- Applying function to every column in a data frame: We will use `anscombe` as example since it has all numeric columns.

```
purrr::map_df(head(anscombe), .f = scale01)
#> # A tibble: 6 x 8
#>      x1    x2    x3    x4    y1     y2     y3    y4
#>   <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>  <dbl> <dbl>
#> 1 0.333 0.333 0.333   NaN 0.362 0.897  0.116  0.266
#> 2 0     0     0       NaN 0     0.0345 0      0
#> 3 0.833 0.833 0.833   NaN 0.209 0.552  1      0.633
#> 4 0.167 0.167 0.167   NaN 0.618 0.578  0.0570 1
#> 5 0.5   0.5   0.5     NaN 0.458 1      0.174  0.880
#> 6 1     1     1       NaN 1     0      0.347  0.416
```

- Applying function to every numeric column in a data frame: We will use `iris` as example since not all of its columns are of numeric type.

```
purrr::modify_if(head(iris), .p = is.numeric, .f = scale01)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1        0.625   0.5555556         0.25           0  setosa
#> 2        0.375   0.0000000         0.25           0  setosa
#> 3        0.125   0.2222222         0.00           0  setosa
#> 4        0.000   0.1111111         0.50           0  setosa
#> 5        0.500   0.6666667         0.25           0  setosa
#> 6        1.000   1.0000000         1.00           1  setosa
```

---

## 9.4 Base functionals (Exercises 9.7.3)

---

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

**A1.** Let's prepare an array and apply a function over different margins:

```
(m <- as.array(table(mtcars$cyl, mtcars$am, mtcars$vs)))
#> , ,  = 0
#> 
#> 
#>     auto manual
```

```
#>   4    0      1
#>   6    0      3
#>   8   12      2
#>
#> , ,  = 1
#>
#>
#>     auto manual
#>   4    3      7
#>   6    4      0
#>   8    0      0

# rows
apply(m, 1, function(x) x^2)
#>
#>         4  6   8
#>   [1,]  0  0 144
#>   [2,]  1  9   4
#>   [3,]  9 16   0
#>   [4,] 49  0   0

# columns
apply(m, 2, function(x) x^2)
#>
#>        auto manual
#>   [1,]    0      1
#>   [2,]    0      9
#>   [3,]  144      4
#>   [4,]    9     49
#>   [5,]   16      0
#>   [6,]    0      0

# rows and columns
apply(m, c(1, 2), function(x) x^2)
#> , ,  = auto
#>
#>
#>     4  6   8
#>   0 0  0 144
#>   1 9 16   0
#>
#> , ,  = manual
#>
#>
#>      4 6 8
#>   0  1 9 4
```

```
#>   1 49 0 0
```

As can be seen, `apply()` returns outputs organised first by the margins being operated over, and only then the results.

---

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

**A2.** Let's consider them one-by-one.

- `eapply()`

As mentioned in its documentation:

> `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':
#>
#>     %@%, flatten, flatten_chr, flatten_dbl,
#>     flatten_int, flatten_lgl, flatten_raw, invoke,
#>     splice
#> The following object is masked from 'package:magrittr':
#>
#>     set_names

e <- env("x" = 1, "y" = 2)
rlang::env_print(e)
#> <environment: 0x5630cc44afb8>
#> 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_tree()`.

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

purrr::modify_tree(X, leaf = length)
#> [[1]]
#> [[1]]$a
#> [1] 1
#>
#> [[1]]$b
#> [[1]]$b$c
#> [1] 2
#>
#>
#>
#> $d
#> [1] 1
```

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

**A3.** As mentioned in the suggested reading material:

> A number $x$ is called a fixed point of a function $f$ if $x$ satisfies the equation $f(x) = x$. For some functions $f$ we can locate a fixed point by beginning with an initial guess and applying $f$ repeatedly, $f(x), f(f(x)), f(f(f(x))), ...$ until the value does not change very much. Using this idea, we can devise a procedure fixed-point that takes as inputs a function and an initial guess and produces an approximation to a fixed point of the function.

Let's first implement a fixed-point algorithm:

```
close_enough <- function(x1, x2, tolerance = 0.001) {
  if (abs(x1 - x2) < tolerance) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}

find_fixed_point <- function(.f, .guess, tolerance = 0.001) {
  .next <- .f(.guess)
  is_close_enough <- close_enough(.next, .guess, tolerance =
 ↪  tolerance)

  if (is_close_enough) {
    return(.next)
  } else {
    find_fixed_point(.f, .next, tolerance)
  }
}
```

Let's check if it works as expected:

```
find_fixed_point(cos, 1.0)
#> [1] 0.7387603

# cos(x) = x
cos(find_fixed_point(cos, 1.0))
#> [1] 0.7393039
```

We will solve only one exercise from the reading material. Rest are beyond the scope of this solution manual.

> Show that the golden ratio $\phi$ is a fixed point of the transformation $x \mapsto 1 + 1/x$, and use this fact to compute $\phi$ by means of the fixed-point procedure.

```
golden_ratio_f <- function(x) 1 + (1 / x)

find_fixed_point(golden_ratio_f, 1.0)
#> [1] 1.618182
```

---

## 9.5 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  dplyr         1.1.4   2023-11-17 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  generics      0.1.3   2022-07-05 [1] RSPM
```

```
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  purrr       * 1.0.4   2025-02-05 [1] RSPM
#>  R6            2.6.1   2025-02-15 [1] RSPM
#>  rlang       * 1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tibble        3.2.1   2023-03-20 [1] RSPM
#>  tidyselect    1.2.1   2024-03-11 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  withr         3.0.2   2024-10-28 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ------------------------------------------------------------
```

# Chapter 10

# Function factories

Attaching the needed libraries:

```
library(rlang, warn.conflicts = FALSE)
library(ggplot2, warn.conflicts = FALSE)
```

## 10.1 Factory fundamentals (Exercises 10.2.6)

---

**Q1.** The definition of `force()` is simple:

```
force
#> function (x)
#> x
#> <bytecode: 0x56173e8a3840>
#> <environment: namespace:base>
```

Why is it better to `force(x)` instead of just `x`?

**A1.** Due to lazy evaluation, argument to a function won't be evaluated until its value is needed. But sometimes we may want to have eager evaluation, and using `force()` makes this intent clearer.

---

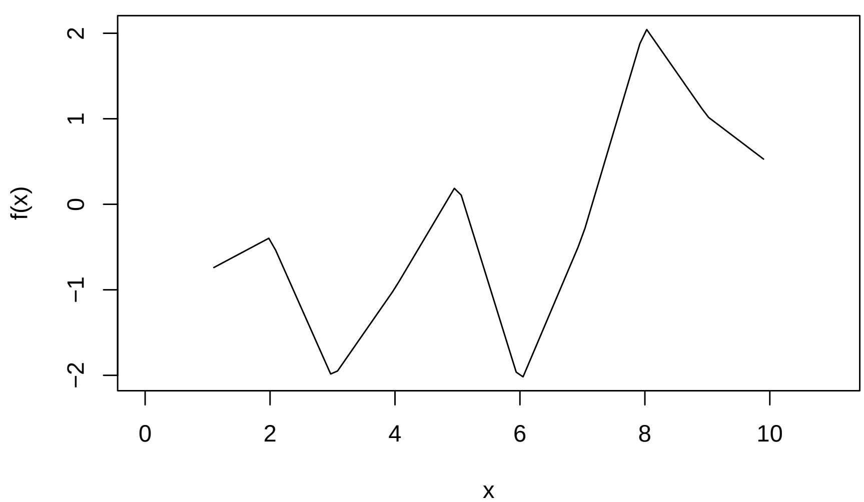
**Q2.** Base R contains two function factories, `approxfun()` and `ecdf()`. Read their documentation and experiment to figure out what the functions do and what they return.

**A2.** About the two function factories-

- `approxfun()`

This function factory returns a function performing the linear (or constant) interpolation.

```
x <- 1:10
y <- rnorm(10)
f <- approxfun(x, y)
f
#> function (v)
#> .approxfun(x, y, v, method, yleft, yright, f, na.rm)
#> <bytecode: 0x561742b9ff30>
#> <environment: 0x561742b9f5c8>
f(x)
#>  [1] -0.7786629 -0.3894764 -2.0337983 -0.9823731  0.2478901
#>  [6] -2.1038646 -0.3814180  2.0749198  1.0271384  0.4730142
curve(f(x), 0, 11)
```




- `ecdf()`

This function factory computes an empirical cumulative distribution function.

```
x <- rnorm(12)
f <- ecdf(x)
f
#> Empirical CDF
#> Call: ecdf(x)
#>  x[1:12] = -1.8793, -1.3221, -1.2392,  ..., 1.1604, 1.7956
f(seq(-2, 2, by = 0.1))
#>  [1] 0.00000000 0.00000000 0.08333333 0.08333333 0.08333333
#>  [6] 0.08333333 0.08333333 0.16666667 0.25000000 0.25000000
#> [11] 0.33333333 0.33333333 0.33333333 0.41666667 0.41666667
#> [16] 0.41666667 0.41666667 0.50000000 0.58333333 0.58333333
#> [21] 0.66666667 0.75000000 0.75000000 0.75000000 0.75000000
#> [26] 0.75000000 0.75000000 0.75000000 0.75000000 0.83333333
#> [31] 0.83333333 0.83333333 0.91666667 0.91666667 0.91666667
#> [36] 0.91666667 0.91666667 0.91666667 1.00000000 1.00000000
#> [41] 1.00000000
```

---

**Q3.** Create a function `pick()` that takes an index, `i`, as an argument and returns a function with an argument `x` that subsets `x` with `i`.

```
pick(1)(x)
# should be equivalent to
x[[1]]

lapply(mtcars, pick(5))
# should be equivalent to
lapply(mtcars, function(x) x[[5]])
```

**A3.** To write desired function, we just need to make sure that the argument `i` is eagerly evaluated.

```
pick <- function(i) {
  force(i)
  function(x) x[[i]]
}
```

Testing it with specified test cases:

```
x <- list("a", "b", "c")
identical(x[[1]], pick(1)(x))
#> [1] TRUE
```

```r
identical(
  lapply(mtcars, pick(5)),
  lapply(mtcars, function(x) x[[5]])
)
#> [1] TRUE
```

---

**Q4.** Create a function that creates functions that compute the $i^{th}$ central moment of a numeric vector. You can test it by running the following code:

```r
m1 <- moment(1)
m2 <- moment(2)
x <- runif(100)
stopifnot(all.equal(m1(x), 0))
stopifnot(all.equal(m2(x), var(x) * 99 / 100))
```

**A4.** The following function satisfied the specified requirements:

```r
moment <- function(k) {
  force(k)

  function(x) (sum((x - mean(x))^k)) / length(x)
}
```

Testing it with specified test cases:

```r
m1 <- moment(1)
m2 <- moment(2)
x <- runif(100)

stopifnot(all.equal(m1(x), 0))
stopifnot(all.equal(m2(x), var(x) * 99 / 100))
```

---

**Q5.** What happens if you don't use a closure? Make predictions, then verify with the code below.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}
```

**A5.** In case closures are not used in this context, the counts are stored in a global variable, which can be modified by other processes or even deleted.

```
new_counter2()
#> [1] 1

new_counter2()
#> [1] 2

new_counter2()
#> [1] 3

i <- 20
new_counter2()
#> [1] 21
```

---

**Q6.** What happens if you use `<-` instead of `<<-`? Make predictions, then verify with the code below.

```
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

**A6.** In this case, the function will always return `1`.

```
new_counter3()
#> function ()
#> {
#>     i <- i + 1
#>     i
#> }
```

```
#> <environment: 0x5617428d5d28>

new_counter3()
#> function ()
#> {
#>     i <- i + 1
#>     i
#> }
#> <bytecode: 0x561742b0dfb0>
#> <environment: 0x56174299df30>
```

---

## 10.2 Graphical factories (Exercises 10.3.4)

---

**Q1.** Compare and contrast `ggplot2::label_bquote()` with `scales::number_format()`.

**A1.** To compare and contrast, let's first look at the source code for these functions:

- `ggplot2::label_bquote()`

```
ggplot2::label_bquote
#> function (rows = NULL, cols = NULL, default)
#> {
#>     cols_quoted <- substitute(cols)
#>     rows_quoted <- substitute(rows)
#>     call_env <- env_parent()
#>     fun <- function(labels) {
#>         quoted <- resolve_labeller(rows_quoted, cols_quoted,
#>             labels)
#>         if (is.null(quoted)) {
#>             return(label_value(labels))
#>         }
#>         evaluate <- function(...) {
#>             params <- list(...)
#>             params <- as_environment(params, call_env)
#>             eval(substitute(bquote(expr, params), list(expr =
#>   quoted)))
#>         }
#>         list(inject(mapply(evaluate, !!!labels, SIMPLIFY =
#>   FALSE)))
```

```
#>     }
#>     structure(fun, class = "labeller")
#> }
#> <bytecode: 0x561742e03680>
#> <environment: namespace:ggplot2>
```

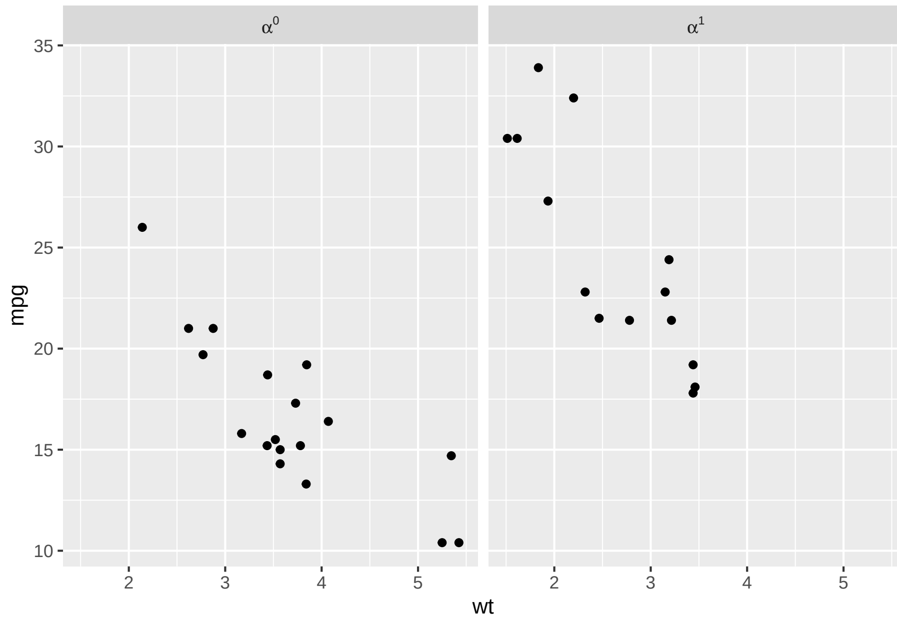
- `scales::number_format()`

```
scales::number_format
#> function (accuracy = NULL, scale = 1, prefix = "", suffix =
 ↪  "",
#>     big.mark = " ", decimal.mark = ".", style_positive =
 ↪  c("none",
#>         "plus", "space"), style_negative = c("hyphen",
 ↪  "minus",
#>         "parens"), scale_cut = NULL, trim = TRUE, ...)
#> {
#>     force_all(accuracy, scale, prefix, suffix, big.mark,
 ↪  decimal.mark,
#>         style_positive, style_negative, scale_cut, trim, ...)
#>     function(x) {
#>         number(x, accuracy = accuracy, scale = scale, prefix =
 ↪  prefix,
#>             suffix = suffix, big.mark = big.mark, decimal.mark
 ↪  = decimal.mark,
#>             style_positive = style_positive, style_negative =
 ↪  style_negative,
#>             scale_cut = scale_cut, trim = trim, ...)
#>     }
#> }
#> <bytecode: 0x561743172b88>
#> <environment: namespace:scales>
```

Both of these functions return formatting functions used to style the facets labels and other labels to have the desired format in `{ggplot2}` plots.

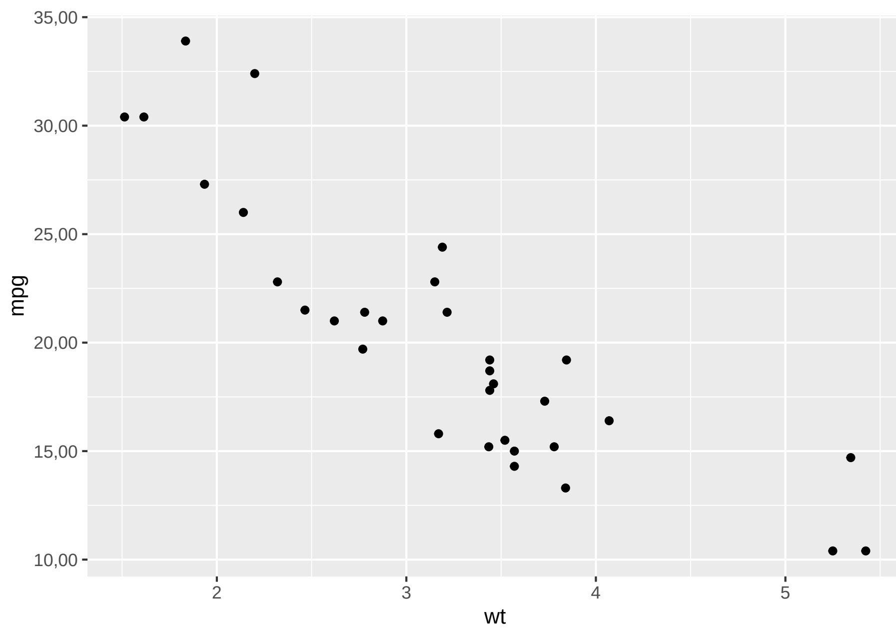
For example, using plotmath expression in the facet label:

```
library(ggplot2)

p <- ggplot(mtcars, aes(wt, mpg)) +
  geom_point()
p + facet_grid(. ~ vs, labeller = label_bquote(cols =
 ↪  alpha^.(vs)))
```




Or to display axes labels in the desired format:

```
library(scales)

ggplot(mtcars, aes(wt, mpg)) +
  geom_point() +
  scale_y_continuous(labels = number_format(accuracy = 0.01,
   ↪  decimal.mark = ","))
```




The `ggplot2::label_bquote()` adds an additional class to the returned function.

The `scales::number_format()` function is a simple pass-through method that forces evaluation of all its parameters and passes them on to the underlying `scales::number()` function.

---

## 10.3 Statistical factories (Exercises 10.4.4)

---

**Q1.** In `boot_model()`, why don't I need to force the evaluation of `df` or `model`?

**A1.** We don't need to force the evaluation of `df` or `model` because these arguments are automatically evaluated by `lm()`:

```
boot_model <- function(df, formula) {
  mod <- lm(formula, data = df)
  fitted <- unname(fitted(mod))
  resid <- unname(resid(mod))
  rm(mod)
```

```r
  function() {
    fitted + sample(resid)
  }
}
```

---

**Q2.** Why might you formulate the Box-Cox transformation like this?

```r
boxcox3 <- function(x) {
  function(lambda) {
    if (lambda == 0) {
      log(x)
    } else {
      (x^lambda - 1) / lambda
    }
  }
}
```

**A2.** To see why we formulate this transformation like above, we can compare it to the one mentioned in the book:

```r
boxcox2 <- function(lambda) {
  if (lambda == 0) {
    function(x) log(x)
  } else {
    function(x) (x^lambda - 1) / lambda
  }
}
```

Let's have a look at one example with each:

```r
boxcox2(1)
#> function (x)
#> (x^lambda - 1)/lambda
#> <environment: 0x561743ec7d48>

boxcox3(mtcars$wt)
#> function (lambda)
#> {
#>     if (lambda == 0) {
#>         log(x)
#>     }
```

```
#>     else {
#>         (x^lambda - 1)/lambda
#>     }
#> }
#> <environment: 0x561743f33598>
```

As can be seen:

- in `boxcox2()`, we can vary `x` for the same value of `lambda`, while
- in `boxcox3()`, we can vary `lambda` for the same vector.

Thus, `boxcox3()` can be handy while exploring different transformations across inputs.

---

**Q3.** Why don't you need to worry that `boot_permute()` stores a copy of the data inside the function that it generates?

**A3.** If we look at the source code generated by the function factory, we notice that the exact data frame (`mtcars`) is not referenced:

```
boot_permute <- function(df, var) {
  n <- nrow(df)
  force(var)

  function() {
    col <- df[[var]]
    col[sample(n, replace = TRUE)]
  }
}

boot_permute(mtcars, "mpg")
#> function ()
#> {
#>     col <- df[[var]]
#>     col[sample(n, replace = TRUE)]
#> }
#> <environment: 0x5617444a8dc0>
```

This is why we don't need to worry about a copy being made because the `df` in the function environment points to the memory address of the data frame. We can confirm this by comparing their memory addresses:

```r
boot_permute_env <- rlang::fn_env(boot_permute(mtcars, "mpg"))
rlang::env_print(boot_permute_env)
#> <environment: 0x561744a33e48>
#> Parent: <environment: global>
#> Bindings:
#> * n: <int>
#> * df: <df[,11]>
#> * var: <chr>

identical(
  lobstr::obj_addr(boot_permute_env$df),
  lobstr::obj_addr(mtcars)
)
#> [1] TRUE
```

We can also check that the values of these bindings are the same as what we entered into the function factory:

```r
identical(boot_permute_env$df, mtcars)
#> [1] TRUE
identical(boot_permute_env$var, "mpg")
#> [1] TRUE
```

---

**Q4.** How much time does `ll_poisson2()` save compared to `ll_poisson1()`? Use `bench::mark()` to see how much faster the optimisation occurs. How does changing the length of `x` change the results?

**A4.** Let's first compare the performance of these functions with the example in the book:

```r
ll_poisson1 <- function(x) {
  n <- length(x)

  function(lambda) {
    log(lambda) * sum(x) - n * lambda - sum(lfactorial(x))
  }
}

ll_poisson2 <- function(x) {
  n <- length(x)
  sum_x <- sum(x)
  c <- sum(lfactorial(x))
```

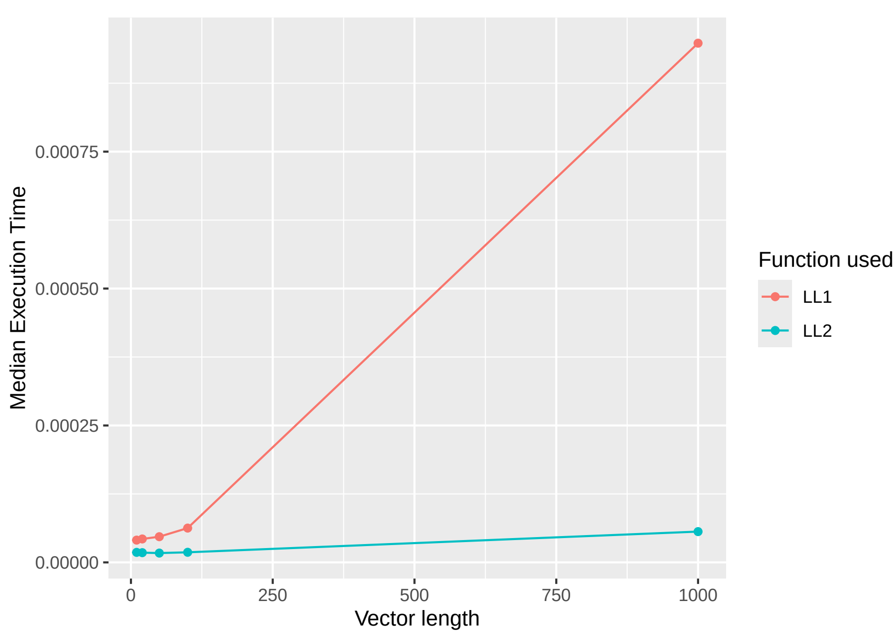
```
  function(lambda) {
    log(lambda) * sum_x - n * lambda - c
  }
}

x1 <- c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)

bench::mark(
  "LL1" = optimise(ll_poisson1(x1), c(0, 100), maximum = TRUE),
  "LL2" = optimise(ll_poisson2(x1), c(0, 100), maximum = TRUE)
)
#> # 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 LL1          27.7us   28.9us    33444.    12.8KB     36.8
#> 2 LL2          14.8us   15.5us    61542.        0B     36.9
```

As can be seen, the second version is much faster than the first version.

We can also vary the length of the vector and confirm that across a wide range of vector lengths, this performance advantage is observed.

```
generate_ll_benches <- function(n) {
  x_vec <- sample.int(n, n)

  bench::mark(
    "LL1" = optimise(ll_poisson1(x_vec), c(0, 100), maximum =
     ↪  TRUE),
    "LL2" = optimise(ll_poisson2(x_vec), c(0, 100), maximum =
     ↪  TRUE)
  )[1:4] %>%
    dplyr::mutate(length = n, .before = expression)
}

(df_bench <- purrr::map_dfr(
  .x = c(10, 20, 50, 100, 1000),
  .f = ~ generate_ll_benches(n = .x)
))
#> # A tibble: 10 x 5
#>    length expression      min   median `itr/sec`
#>     <dbl> <bch:expr> <bch:tm> <bch:tm>     <dbl>
#>  1     10 LL1          39.3us   40.7us    24120.
#>  2     10 LL2          17.7us   18.3us    53708.
#>  3     20 LL1          41.5us   42.8us    23006.
#>  4     20 LL2          17.1us   17.8us    55303.
```

```
#>  5     50 LL1           45.7us   46.9us    20999.
#>  6     50 LL2           16.4us     17us    57673.
#>  7    100 LL1           60.8us   62.6us    14942.
#>  8    100 LL2           17.7us   18.4us    51986.
#>  9   1000 LL1          830.4us    948us     1070.
#> 10   1000 LL2           54.7us   56.2us    17370.

ggplot(
  df_bench,
  aes(
    x = as.numeric(length),
    y = median,
    group = as.character(expression),
    color = as.character(expression)
  )
) +
  geom_point() +
  geom_line() +
  labs(
    x = "Vector length",
    y = "Median Execution Time",
    colour = "Function used"
  )
```



---

## 10.4 Function factories + functionals (Exercises 10.5.1)

**Q1.** Which of the following commands is equivalent to `with(x, f(z))`?

```
(a) `x$f(x$z)`.
(b) `f(x$z)`.
(c) `x$f(z)`.
(d) `f(z)`.
(e) It depends.
```

**A1.** It depends on whether `with()` is used with a data frame or a list.

```r
f <- mean
z <- 1
x <- list(f = mean, z = 1)

identical(with(x, f(z)), x$f(x$z))
#> [1] TRUE

identical(with(x, f(z)), f(x$z))
#> [1] TRUE

identical(with(x, f(z)), x$f(z))
#> [1] TRUE

identical(with(x, f(z)), f(z))
#> [1] TRUE
```

---

**Q2.** Compare and contrast the effects of `env_bind()` vs. `attach()` for the following code.

**A2.** Let's compare and contrast the effects of `env_bind()` vs. `attach()`.

- `attach()` adds `funs` to the search path. Since these functions have the same names as functions in `{base}` package, the attached names mask the ones in the `{base}` package.

```
funs <- list(
  mean = function(x) mean(x, na.rm = TRUE),
  sum = function(x) sum(x, na.rm = TRUE)
)

attach(funs)
#> The following objects are masked from package:base:
#>
#>     mean, sum

mean
#> function (x)
#> mean(x, na.rm = TRUE)
head(search())
#> [1] ".GlobalEnv"       "funs"              "package:scales"
#> [4] "package:ggplot2"  "package:rlang"     "package:magrittr"

mean <- function(x) stop("Hi!")
mean
#> function (x)
#> stop("Hi!")
head(search())
#> [1] ".GlobalEnv"       "funs"              "package:scales"
#> [4] "package:ggplot2"  "package:rlang"     "package:magrittr"

detach(funs)
```

- `env_bind()` adds the functions in `funs` to the global environment, instead of masking the names in the `{base}` package.

```
env_bind(globalenv(), !!!funs)
mean
#> function (x)
#> mean(x, na.rm = TRUE)

mean <- function(x) stop("Hi!")
mean
#> function (x)
#> stop("Hi!")
env_unbind(globalenv(), names(funs))
```

Note that there is no `"funs"` in this output.

---

## 10.5 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bench         1.1.4   2025-01-16 [1] RSPM
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  colorspace    2.1-1   2024-07-26 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  dplyr         1.1.4   2023-11-17 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  farver        2.1.2   2024-05-13 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  generics      0.1.3   2022-07-05 [1] RSPM
#>  ggplot2     * 3.5.1   2024-04-23 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  grid          4.4.3   2025-02-28 [3] local
#>  gtable        0.3.6   2024-10-25 [1] RSPM
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  labeling      0.4.3   2023-08-29 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  lobstr        1.1.2   2022-06-22 [1] RSPM
```

```
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  munsell       0.5.1   2024-04-01 [1] RSPM
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  profmem       0.6.0   2020-12-13 [1] RSPM
#>  purrr         1.0.4   2025-02-05 [1] RSPM
#>  R6            2.6.1   2025-02-15 [1] RSPM
#>  rlang       * 1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  scales      * 1.3.0   2023-11-28 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tibble        3.2.1   2023-03-20 [1] RSPM
#>  tidyselect    1.2.1   2024-03-11 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utf8          1.2.4   2023-10-22 [1] RSPM
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  withr         3.0.2   2024-10-28 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 11

# Function operators

Attaching the needed libraries:

```
library(purrr, warn.conflicts = FALSE)
```

## 11.1 Existing function operators (Exercises 11.2.3)

---

**Q1.** Base R provides a function operator in the form of `Vectorize()`. What does it do? When might you use it?

**A1.** `Vectorize()` function creates a function that vectorizes the action of the provided function over specified arguments (i.e., it acts on each element of the vector). We will see its utility by solving a problem that otherwise would be difficult to solve.

The problem is to find indices of matching numeric values for the given threshold by creating a hybrid of the following functions:

- `%in%` (which doesn't provide any way to provide tolerance when comparing numeric values),
- `dplyr::near()` (which is vectorized element-wise and thus expects two vectors of equal length)

```
which_near <- function(x, y, tolerance) {
  # Vectorize `dplyr::near()` function only over the `y`
   ↪  argument.
  # `Vectorize()` is a function operator and will return a
   ↪  function.
  customNear <- Vectorize(dplyr::near, vectorize.args = c("y"),
 ↪  SIMPLIFY = FALSE)

  # Apply the vectorized function to vector arguments and then
   ↪  check where the
  # comparisons are equal (i.e. `TRUE`) using `which()`.
  #
  # Use `compact()` to remove empty elements from the resulting
   ↪  list.
  index_list <- purrr::compact(purrr::map(customNear(x, y, tol =
 ↪  tolerance), which))

  # If there are any matches, return the indices as an atomic
   ↪  vector of integers.
  if (length(index_list) > 0L) {
    index_vector <- purrr::simplify(index_list, "integer")
    return(index_vector)
  }

  # If there are no matches
  return(integer(0L))
}
```

Let's use it:

```
x1 <- c(2.1, 3.3, 8.45, 8, 6)
x2 <- c(6, 8.40, 3)

which_near(x1, x2, tolerance = 0.1)
#> [1] 5 3
```

Note that we needed to create a new function for this because neither of the existing functions do what we want.

```
which(x1 %in% x2)
#> [1] 5

which(dplyr::near(x1, x2, tol = 0.1))
#> Warning in x - y: longer object length is not a multiple of
```

```
#> shorter object length
#> integer(0)
```

We solved a complex task here using the `Vectorize()` function!

---

**Q2.** Read the source code for `possibly()`. How does it work?

**A2.** Let's have a look at the source code for this function:

```
possibly
#> function (.f, otherwise = NULL, quiet = TRUE)
#> {
#>     .f <- as_mapper(.f)
#>     force(otherwise)
#>     check_bool(quiet)
#>     function(...) {
#>         tryCatch(.f(...), error = function(e) {
#>             if (!quiet)
#>                 message("Error: ", conditionMessage(e))
#>             otherwise
#>         })
#>     }
#> }
#> <bytecode: 0x56444ca6b3d0>
#> <environment: namespace:purrr>
```

Looking at this code, we can see that `possibly()`:

- uses `tryCatch()` for error handling
- has a parameter `otherwise` to specify default value in case an error occurs
- has a parameter `quiet` to suppress error message (if needed)

---

**Q3.** Read the source code for `safely()`. How does it work?

**A3.** Let's have a look at the source code for this function:

```
safely
#> function (.f, otherwise = NULL, quiet = TRUE)
#> {
```

```
#>     .f <- as_mapper(.f)
#>     force(otherwise)
#>     check_bool(quiet)
#>     function(...) capture_error(.f(...), otherwise, quiet)
#> }
#> <bytecode: 0x56444cc54890>
#> <environment: namespace:purrr>

purrr:::capture_error
#> function (code, otherwise = NULL, quiet = TRUE)
#> {
#>     tryCatch(list(result = code, error = NULL), error =
↪  function(e) {
#>         if (!quiet)
#>             message("Error: ", conditionMessage(e))
#>         list(result = otherwise, error = e)
#>     })
#> }
#> <bytecode: 0x56444cc8d1f0>
#> <environment: namespace:purrr>
```

Looking at this code, we can see that `safely()`:

- uses a list to save both the results (if the function executes successfully) and the error (if it fails)
- uses `tryCatch()` for error handling
- has a parameter `otherwise` to specify default value in case an error occurs
- has a parameter `quiet` to suppress error message (if needed)

---

## 11.2 Case study: Creating your own function operators (Exercises 11.3.1)

---

**Q1.** Weigh the pros and cons of `download.file %>% dot_every(10) %>% delay_by(0.1)` versus `download.file %>% delay_by(0.1) %>% dot_every(10)`.

**A1.** Although both of these chains of piped operations produce the same number of dots and would need the same amount of time, there is a subtle difference in how they do this.

- `download.file %>% dot_every(10) %>% delay_by(0.1)`

Here, the printing of the dot is also delayed, and the first dot is printed when the 10th URL download starts.

- `download.file %>% delay_by(0.1) %>% dot_every(10)`

Here, the first dot is printed after the 9th download is finished, and the 10th download starts after a short delay.

---

**Q2.** Should you memoise `download.file()`? Why or why not?

**A2.** Since `download.file()` is meant to download files from the Internet, memoising it is not recommended for the following reasons:

- Memoization is helpful when giving the same input the function returns the same output. This is not necessarily the case for webpages since they constantly change, and you may continue to "download" an outdated version of the webpage.
- Memoization works by caching results, which can take up a significant amount of memory.

---

**Q3.** Create a function operator that reports whenever a file is created or deleted in the working directory, using `dir()` and `setdiff()`. What other global function effects might you want to track?

**A3.** First, let's create helper functions to compare and print added or removed filenames:

```
print_multiple_entries <- function(header, entries) {
  message(paste0(header, ":\n"), paste0(entries, collapse =
   ↪  "\n"))
}

file_comparator <- function(old, new) {
  if (setequal(old, new)) {
    return()
  }

  removed <- setdiff(old, new)
```

```
  added <- setdiff(new, old)

  if (length(removed) > 0L) print_multiple_entries("- File
   ↪  removed", removed)
  if (length(added) > 0L) print_multiple_entries("- File added",
   ↪  added)
}
```

We can then write a function operator and use it to create functions that will do the necessary tracking:

```
dir_tracker <- function(f) {
  force(f)
  function(...) {
    old_files <- dir()
    on.exit(file_comparator(old_files, dir()), add = TRUE)

    f(...)
  }
}

file_creation_tracker <- dir_tracker(file.create)
file_deletion_tracker <- dir_tracker(file.remove)
```

Let's try it out:

```
file_creation_tracker(c("a.txt", "b.txt"))
#> - File added:
#> a.txt
#> b.txt
#> [1] TRUE TRUE

file_deletion_tracker(c("a.txt", "b.txt"))
#> - File removed:
#> a.txt
#> b.txt
#> [1] TRUE TRUE
```

Other global function effects we might want to track:

- working directory
- environment variables
- connections

- library paths
- graphics devices
- etc.

---

**Q4.** Write a function operator that logs a timestamp and message to a file every time a function is run.

**A4.** The following function operator logs a timestamp and message to a file every time a function is run:

```
# helper function to write to a file connection
write_line <- function(filepath, ...) {
  cat(..., "\n", sep = "", file = filepath, append = TRUE)
}

# function operator
logger <- function(f, filepath) {
  force(f)
  force(filepath)

  write_line(filepath, "Function created at: ",
   ↪  as.character(Sys.time()))

  function(...) {
    write_line(filepath, "Function called at:  ",
     ↪  as.character(Sys.time()))
    f(...)
  }
}

# check that the function works as expected with a tempfile
withr::with_tempfile("logfile", code = {
  logged_runif <- logger(runif, logfile)

  Sys.sleep(sample.int(10, 1))
  logged_runif(1)

  Sys.sleep(sample.int(10, 1))
  logged_runif(2)

  Sys.sleep(sample.int(10, 1))
  logged_runif(3)

  cat(readLines(logfile), sep = "\n")
```

```
})
#> Function created at: 2025-03-16 00:14:54.002037
#> Function called at:  2025-03-16 00:14:59.008844
#> Function called at:  2025-03-16 00:15:04.014166
#> Function called at:  2025-03-16 00:15:12.022491
```

---

**Q5.** Modify `delay_by()` so that instead of delaying by a fixed amount of time, it ensures that a certain amount of time has elapsed since the function was last called. That is, if you called `g <- delay_by(1, f); g(); Sys.sleep(2); g()` there shouldn't be an extra delay.

**A5.** Modified version of the function meeting the specified requirements:

```
delay_by_atleast <- function(f, amount) {
  force(f)
  force(amount)

  # the last time the function was run
  last_time <- NULL

  function(...) {
    if (!is.null(last_time)) {
      wait <- (last_time - Sys.time()) + amount
      if (wait > 0) Sys.sleep(wait)
    }

    # update the time in the parent frame for the next run when
    ↪  the function finishes
    on.exit(last_time <<- Sys.time())

    f(...)
  }
}
```

---

## 11.3 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info ------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
 ↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages ----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  dplyr         1.1.4   2023-11-17 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  generics      0.1.3   2022-07-05 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  purrr       * 1.0.4   2025-02-05 [1] RSPM
#>  R6            2.6.1   2025-02-15 [1] RSPM
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
```

```
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tibble        3.2.1   2023-03-20 [1] RSPM
#>  tidyselect    1.2.1   2024-03-11 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  withr         3.0.2   2024-10-28 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

Chapter 12

# Base Types

No exercises.

# Chapter 13

# S3

Attaching the needed libraries:

```
library(sloop, warn.conflicts = FALSE)
library(dplyr, warn.conflicts = FALSE)
library(purrr, warn.conflicts = FALSE)
```

## 13.1 Basics (Exercises 13.2.1)

---

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

**A1.** The difference between `t.test()` and `t.data.frame()` is the following:

- `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` objects.

We can also confirm these function types using `ftype()`:

```
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 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
```

Given the number of `.`s in this name, it is quite confusing to figure out what is the name of the generic and the name of the class.

---

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

```r
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.** The difference in behaviour in the specified calls.

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

```r
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:

```r
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"
```

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`.

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

```r
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`.

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

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

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

---

## 13.2 Classes (Exercises 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.** A data frame is built on top of a named list of atomic vectors and has attributes for row names:

```r
unclass(data.frame())
#> named list()
#> attr(,"row.names")
#> integer(0)
```

The restriction imposed on individual elements is that they need to have the same length. Additionally, the names need to be syntactically valid and unique.

```r
new_data_frame <- function(x = list(), row.names = character()) {
  # row names should be character
  if (!all(is.character(row.names))) {
    stop("Row name should be of `chracter` type.", call. = FALSE)
  }

  # all elements should have the same length
  unique_element_lengths <- unique(purrr::map_int(x, length))
  if (length(unique_element_lengths) > 1L) {
    stop("All list elements in `x` should have same length.",
    ↪  call. = FALSE)
  }

  # if not provided, generate row names
  # this is necessary if there is at least one element in the
  ↪  list
  if (length(x) > 0L && length(row.names) == 0L) {
    row.names <- .set_row_names(unique_element_lengths)
  }

  structure(x, class = "data.frame", row.names = row.names)
}
```

Let's try it out:

```r
new_data_frame(list("x" = 1, "y" = c(2, 3)))
#> Error: All list elements in `x` should have same length.

new_data_frame(list("x" = 1, "y" = c(2)), row.names = 1L)
#> Error: Row name should be of `chracter` type.

new_data_frame(list())
#> data frame with 0 columns and 0 rows

new_data_frame(list("x" = 1, "y" = 2))
#>   x y
```

```
#> 1 1 2

new_data_frame(list("x" = 1, "y" = 2), row.names = "row-1")
#>       x y
#> row-1 1 2
```

---

**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?

**A2.** When one or more `values` is not found in `levels`, those values are converted to `NA` in `base::factor()`:

```
base::factor(c("a", "b", "c"), levels = c("a", "c"))
#> [1] a    <NA> c
#> Levels: a c
```

In the new constructor, we can throw an error to inform the user:

```
new_factor <- function(x = integer(), levels = character()) {
  stopifnot(is.integer(x))
  stopifnot(is.character(levels))

  structure(
    x,
    levels = levels,
    class = "factor"
  )
}

validate_factor <- function(x) {
  values <- unclass(x)
  levels <- attr(x, "levels")

  if (!all(!is.na(values) & values > 0)) {
    stop(
      "All `x` values must be non-missing and greater than zero",
      call. = FALSE
    )
  }

  if (length(levels) < max(values)) {
    stop(
```

```r
      "There must be at least as many `levels` as possible values
       ↪  in `x`",
      call. = FALSE
    )
  }

  x
}

create_factor <- function(x = character(), levels = unique(x)) {
  ind <- match(x, levels)

  if (any(is.na(ind))) {
    missing_values <- x[which(is.na(match(x, levels)))]

    stop(
      paste0(
        "Following values from `x` are not present in
         ↪  `levels`:\n",
        paste0(missing_values, collapse = "\n")
      ),
      call. = FALSE
    )
  }

  validate_factor(new_factor(ind, levels))
}
```

Let's try it out:

```r
create_factor(c("a", "b", "c"), levels = c("a", "c"))
#> Error: Following values from `x` are not present in `levels`:
#> b

create_factor(c("a", "b", "c"), levels = c("a", "b", "c"))
#> [1] a b c
#> Levels: a b c
```

---

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

**A3.** The source code for `factor()` can be read here.

There are a number ways in which the base version is more flexible.

- It allows labeling the values:

```r
x <- c("a", "b", "b")
levels <- c("a", "b", "c")
labels <- c("one", "two", "three")

factor(x, levels = levels, labels = labels)
#> [1] one two two
#> Levels: one two three
```

- It checks that the levels are not duplicated.

```r
x <- c("a", "b", "b")
levels <- c("a", "b", "b")

factor(x, levels = levels)
#> Error in `levels<-`(`*tmp*`, value = as.character(levels)):
↪  factor level [3] is duplicated

create_factor(x, levels = levels)
#> [1] a b b
#> Levels: a b b
#> Warning in print.factor(x): duplicated level [3] in factor
```

- The `levels` argument can be `NULL`.

```r
x <- c("a", "b", "b")

factor(x, levels = NULL)
#> [1] <NA> <NA> <NA>
#> Levels:

create_factor(x, levels = NULL)
#> Error: Following values from `x` are not present in `levels`:
#> a
#> b
#> b
```

**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.

**A4.** Categorical variables are typically encoded as dummy variables in regression models and by default each level is compared with the first factor level. Contrats provide a flexible way for such comparisons.

You can set the `"contrasts"` attribute for a factor using `stats::C()`.

Alternatively, you can set the `"contrasts"` attribute using matrix (`?contrasts`):

> [Contrasts] can be a matrix with one row for each level of the factor or a suitable function like contr.poly or a character string giving the name of the function

The constructor provided in the book:

```
new_factor <- function(x = integer(), levels = character()) {
  stopifnot(is.integer(x))
  stopifnot(is.character(levels))

  structure(
    x,
    levels = levels,
    class = "factor"
  )
}
```

Here is how it can be updated to also support contrasts:

```
new_factor <- function(x = integer(),
                       levels = character(),
                       contrasts = NULL) {
  stopifnot(is.integer(x))
  stopifnot(is.character(levels))

  if (!is.null(contrasts)) {
    stopifnot(is.matrix(contrasts) && is.numeric(contrasts))
  }

  structure(
    x,
    levels = levels,
    class = "factor",
    contrasts = contrasts
  )
}
```

**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?

**A5.** `utils::as.roman()` converts Indo-Arabic numerals to Roman numerals. Removing its class also reveals that it is implemented using the base type `integer`:

```r
as.roman(1)
#> [1] I

typeof(unclass(as.roman(1)))
#> [1] "integer"
```

Therefore, we can create a simple constructor to create a new instance of this class:

```r
new_roman <- function(x = integer()) {
  stopifnot(is.integer(x))

  structure(x, class = "roman")
}
```

The docs mention the following:

> Only numbers between 1 and 3899 have a unique representation as roman numbers, and hence others result in as.roman(NA).

```r
as.roman(10000)
#> [1] <NA>
```

Therefore, we can warn the user and then return `NA` in a validator function:

```r
validate_new_roman <- function(x) {
  int_values <- unclass(x)

  if (any(int_values < 1L | int_values > 3899L)) {
    warning(
      "Integer should be between 1 and 3899. Returning `NA`
       ↪  otherwise.",
      call. = FALSE
    )
  }

  x
}
```

The helper function can coerce the entered input to integer type for convenience:

```r
roman <- function(x = integer()) {
  x <- as.integer(x)

  validate_new_roman(new_roman(x))
}
```

Let's try it out:

```r
roman(1)
#> [1] I

roman(c(5, 20, 100, 150, 100000))
#> Warning: Integer should be between 1 and 3899. Returning
#> `NA` otherwise.
#> [1] V    XX   C    CL   <NA>
```

## 13.3 Generics and methods (Exercises 13.4.4)

**Q1.** Read the source code for `t()` and `t.test()` and confirm that `t.test()` is an S3 generic and not an S3 method. What happens if you create an object with class `test` and call `t()` with it? Why?

```r
x <- structure(1:10, class = "test")
t(x)
```

**A1.** Looking at source code of these functions, we can see that both of these are generic, and we can confirm the same using `{sloop}`:

```r
t
#> function (x)
#> UseMethod("t")
#> <bytecode: 0x555816ea2b68>
#> <environment: namespace:base>
sloop::is_s3_generic("t")
#> [1] TRUE

t.test
#> function (x, ...)
#> UseMethod("t.test")
#> <bytecode: 0x555813676fa8>
#> <environment: namespace:stats>
sloop::is_s3_generic("t.test")
#> [1] TRUE
```

Looking at the `S3` dispatch, we can see that since R can't find `S3` method for `test` class for generic function `t()`, it dispatches the default method, which converts the structure to a matrix:

```
x <- structure(1:10, class = "test")
t(x)
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,]    1    2    3    4    5    6    7    8    9    10
#> attr(,"class")
#> [1] "test"
s3_dispatch(t(x))
#>    t.test
#> => t.default
```

The same behaviour can be observed with a vector:

```
t(1:10)
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,]    1    2    3    4    5    6    7    8    9    10
```

**Q2.** What generics does the `table` class have methods for?

**A2.** The `table` class have methods for the following generics:

```
s3_methods_class("table")
#> # A tibble: 11 x 4
#>    generic       class visible source
#>    <chr>         <chr> <lgl>   <chr>
#>  1 [             table TRUE    base
#>  2 aperm         table TRUE    base
#>  3 as_tibble     table FALSE   registered S3method
#>  4 as.data.frame table TRUE    base
#>  5 Axis          table FALSE   registered S3method
#>  6 lines         table FALSE   registered S3method
#>  7 plot          table FALSE   registered S3method
#>  8 points        table FALSE   registered S3method
#>  9 print         table TRUE    base
#> 10 summary       table TRUE    base
#> 11 tail          table FALSE   registered S3method
```

**Q3.** What generics does the `ecdf` class have methods for?

**A3.** The `ecdf` class have methods for the following generics:

```r
s3_methods_class("ecdf")
#> # A tibble: 4 x 4
#>   generic  class visible source
#>   <chr>    <chr> <lgl>   <chr>
#> 1 plot     ecdf  TRUE    stats
#> 2 print    ecdf  FALSE   registered S3method
#> 3 quantile ecdf  FALSE   registered S3method
#> 4 summary  ecdf  FALSE   registered S3method
```

**Q4.** Which base generic has the greatest number of defined methods?

**A4.** To answer this question, first, let's list all functions base has and only retain the generics.

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

# extracting only generics
genFuns <- names(funs) %>%
  purrr::keep(~ sloop::is_s3_generic(.x))
```

Now it's a simple matter of counting number of methods per generic and ordering the data frame in descending order of this count:

```r
purrr::map_dfr(
  genFuns,
  ~ s3_methods_generic(.)
) %>%
  dplyr::group_by(generic) %>%
  dplyr::tally() %>%
  dplyr::arrange(desc(n))
#> # A tibble: 123 x 2
#>    generic           n
#>    <chr>         <int>
#>  1 print           273
#>  2 format          124
#>  3 [                50
#>  4 summary          39
#>  5 as.character     32
#>  6 as.data.frame    32
#>  7 plot             31
#>  8 [[               22
#>  9 [<-              17
#> 10 $                15
```

```
#> # i 113 more rows
```

This reveals that the base generic function with most methods is `print()`.

**Q5.** Carefully read the documentation for `UseMethod()` and explain why the following code returns the results that it does. What two usual rules of function evaluation does `UseMethod()` violate?

```
g <- function(x) {
  x <- 10
  y <- 10
  UseMethod("g")
}
g.default <- function(x) c(x = x, y = y)
x <- 1
y <- 1
g(x)
#> x y
#> 1 1
```

**A5.** If called directly, `g.default()` method takes `x` value from argument and `y` from the global environment:

```
g.default(x)
#> x y
#> 1 1
```

But, if `g()` function is called, it takes the `x` from argument, but comes from function environment:

```
g(x)
#> x y
#> 1 1
```

The docs for `?UseMethod()` clarify why this is the case:

> Any local variables defined before the call to UseMethod are retained

That is, when `UseMethod()` calls `g.default()`, variables defined inside the generic are also available to `g.default()` method. The arguments supplied to the function are passed on as is, however, and cannot be affected by code inside the generic.

Two rules of function evaluation violated by `UseMethod()`:

- Name masking
- A fresh start

**Q6.** What are the arguments to `[`? Why is this a hard question to answer?

**A6.** It is difficult to say how many formal arguments the subsetting `[` operator has because it is a generic function with methods for vectors, matrices, arrays, lists, etc., and these different methods have different number of arguments:

```
s3_methods_generic("[") %>%
  dplyr::filter(source == "base")
#> # A tibble: 17 x 4
#>    generic class           visible source
#>    <chr>   <chr>           <lgl>   <chr>
#>  1 [       AsIs            TRUE    base
#>  2 [       data.frame      TRUE    base
#>  3 [       Date            TRUE    base
#>  4 [       difftime        TRUE    base
#>  5 [       Dlist           TRUE    base
#>  6 [       DLLInfoList     TRUE    base
#>  7 [       factor          TRUE    base
#>  8 [       hexmode         TRUE    base
#>  9 [       listof          TRUE    base
#> 10 [       noquote         TRUE    base
#> 11 [       numeric_version TRUE    base
#> 12 [       octmode         TRUE    base
#> 13 [       POSIXct         TRUE    base
#> 14 [       POSIXlt         TRUE    base
#> 15 [       simple.list     TRUE    base
#> 16 [       table           TRUE    base
#> 17 [       warnings        TRUE    base
```

We can sample a few of them to see the wide variation in the number of formal arguments:

```
# table
names(formals(`[.table`))
#> [1] "x"    "i"    "j"    "..."  "drop"

# Date
names(formals(`[.Date`))
#> [1] "x"    "..."  "drop"

# data frame
names(formals(`[.data.frame`))
#> [1] "x"    "i"    "j"    "drop"
```

```r
# etc.
```

## 13.4 Object styles (Exercises 13.5.1)

**Q1.** Categorise the objects returned by `lm()`, `factor()`, `table()`, `as.Date()`, `as.POSIXct()` `ecdf()`, `ordered()`, `I()` into the styles described above.

**A1.** Objects returned by these functions can be categorized as follows:

- Vector style objects (`length` represents no. of observations)

`factor()`

```r
factor_obj <- factor(c("a", "b"))
length(factor_obj)
#> [1] 2
length(unclass(factor_obj))
#> [1] 2
```

`table()`

```r
tab_object <- table(mtcars$am)
length(tab_object)
#> [1] 2
length(unlist(tab_object))
#> [1] 2
```

`as.Date()`

```r
date_obj <- as.Date("02/27/92", "%m/%d/%y")
length(date_obj)
#> [1] 1
length(unclass(date_obj))
#> [1] 1
```

`as.POSIXct()`

```r
posix_obj <- as.POSIXct(1472562988, origin = "1960-01-01")
length(posix_obj)
#> [1] 1
length(unclass(posix_obj))
#> [1] 1
```

`ordered()`

```
ordered_obj <- ordered(factor(c("a", "b")))
length(ordered_obj)
#> [1] 2
length(unclass(ordered_obj))
#> [1] 2
```

- Record style objects (equi-length vectors to represent object components)

None.

- Dataframe style objects (Record style but two-dimensions)

None.

- Scalar objects (a list to represent a single thing)

`lm()` (represent one regression model)

```
lm_obj <- lm(wt ~ mpg, mtcars)
length(lm_obj)
#> [1] 12
length(unclass(lm_obj))
#> [1] 12
```

`ecdf()` (represents one distribution)

```
ecdf_obj <- ecdf(rnorm(12))
length(ecdf_obj)
#> [1] 1
length(unclass(ecdf_obj))
#> [1] 1
```

`I()` is special: It just adds a new class to the object to indicate that it should be treated *as is*.

```
x <- ecdf(rnorm(12))
class(x)
#> [1] "ecdf"     "stepfun"  "function"
class(I(x))
#> [1] "AsIs"     "ecdf"     "stepfun"  "function"
```

Therefore, the object style would be the same as the superclass' object style.

**Q2.** What would a constructor function for `lm` objects, `new_lm()`, look like? Use `?lm` and experimentation to figure out the required fields and their types.

**A2.** The `lm` object is a scalar object, i.e. this object contains a named list of atomic vectors of varying lengths and types to represent a single thing (a regression model).

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

typeof(mod)
#> [1] "list"

attributes(mod)
#> $names
#>  [1] "coefficients"  "residuals"     "effects"
#>  [4] "rank"          "fitted.values" "assign"
#>  [7] "qr"            "df.residual"   "xlevels"
#> [10] "call"          "terms"         "model"
#>
#> $class
#> [1] "lm"

purrr::map_chr(unclass(mod), typeof)
#>  coefficients     residuals       effects          rank
#>      "double"      "double"      "double"     "integer"
#> fitted.values        assign            qr   df.residual
#>      "double"     "integer"        "list"     "integer"
#>       xlevels          call         terms         model
#>        "list"    "language"    "language"        "list"

purrr::map_int(unclass(mod), length)
#>  coefficients     residuals       effects          rank
#>             2            32            32             1
#> fitted.values        assign            qr   df.residual
#>            32             2             5             1
#>       xlevels          call         terms         model
#>             0             3             3             2
```

Based on this information, we can write a new constructor for this object:

```
new_lm <- function(coefficients,
                   residuals,
                   effects,
                   rank,
```

```
                   fitted.values,
                   assign,
                   qr,
                   df.residual,
                   xlevels,
                   call,
                   terms,
                   model) {
  stopifnot(
    is.double(coefficients),
    is.double(residuals),
    is.double(effects),
    is.integer(rank),
    is.double(fitted.values),
    is.integer(assign),
    is.list(qr),
    is.integer(df.residual),
    is.list(xlevels),
    is.language(call),
    is.language(terms),
    is.list(model)
  )

  structure(
    list(
      coefficients  = coefficients,
      residuals     = residuals,
      effects       = effects,
      rank          = rank,
      fitted.values = fitted.values,
      assign        = assign,
      qr            = qr,
      df.residual   = df.residual,
      xlevels       = xlevels,
      call          = call,
      terms         = terms,
      model         = model
    ),
    class = "lm"
  )
}
```

## 13.5 Inheritance (Exercises 13.6.3)

**Q1.** How does `[.Date` support subclasses? How does it fail to support subclasses?

**A1.** The `[.Date` method is defined as follows:

```
sloop::s3_get_method("[.Date")
#> function (x, ..., drop = TRUE)
#> {
#>     .Date(NextMethod("["), oldClass(x))
#> }
#> <bytecode: 0x555811bcf6c8>
#> <environment: namespace:base>
```

The `.Date` function looks like this:

```
.Date
#> function (xx, cl = "Date")
#> `class<-`(xx, cl)
#> <bytecode: 0x5558135cb6f8>
#> <environment: namespace:base>
```

Here, `oldClass` is the same as `class()`.

Therefore, by reading this code, we can surmise that:

- `[.Date` supports subclasses by preserving the class of the input.
- `[.Date` fails to support subclasses by not preserving the attributes of the input.

For example,

```
x <- structure(Sys.Date(), name = "myName", class = c("subDate",
 ↪  "Date"))

# `$name` is gone
attributes(x[1])
#> $class
#> [1] "subDate" "Date"

x[1]
#> [1] "2025-03-16"
```

**Q2.** R has two classes for representing date time data, `POSIXct` and `POSIXlt`, which both inherit from `POSIXt`. Which generics have different behaviours for the two classes? Which generics share the same behaviour?

**A2.** First, let's demonstrate that `POSIXct` and `POSIXlt` are indeed subclasses and `POSIXt` is the superclass.

```
dt_lt <- as.POSIXlt(Sys.time(), "GMT")
class(dt_lt)
#> [1] "POSIXlt" "POSIXt"

dt_ct <- as.POSIXct(Sys.time(), "GMT")
class(dt_ct)
#> [1] "POSIXct" "POSIXt"

dt_t <- structure(dt_ct, class = "POSIXt")
class(dt_t)
#> [1] "POSIXt"
```

Remember that the way `S3` method dispatch works, if a generic has a method for superclass, then that method is also inherited by the subclass.

We can extract a vector of all generics supported by both sub- and super-classes:

```
(t_generics <- s3_methods_class("POSIXt")$generic)
#>  [1] "-"            "+"            "all.equal"
#>  [4] "as.character" "Axis"         "cut"
#>  [7] "diff"         "hist"         "is.numeric"
#> [10] "julian"       "Math"         "months"
#> [13] "Ops"          "pretty"       "quantile"
#> [16] "quarters"     "round"        "seq"
#> [19] "str"          "trunc"        "weekdays"

(lt_generics <- s3_methods_class("POSIXlt")$generic)
#>  [1] "["             "[["            "[[<-"
#>  [4] "[<-"           "$<-"           "anyNA"
#>  [7] "as.data.frame" "as.Date"       "as.double"
#> [10] "as.list"       "as.matrix"     "as.POSIXct"
#> [13] "as.vector"     "c"             "duplicated"
#> [16] "format"        "is.finite"     "is.infinite"
#> [19] "is.na"         "is.nan"        "length"
#> [22] "length<-"      "mean"          "mtfrm"
#> [25] "names"         "names<-"       "print"
#> [28] "rep"           "sort"          "summary"
#> [31] "Summary"       "unique"        "weighted.mean"
#> [34] "xtfrm"
```

```
(ct_generics <- s3_methods_class("POSIXct")$generic)
#>  [1] "["              "[["             "[<-"
#>  [4] "as.data.frame" "as.Date"        "as.list"
#>  [7] "as.POSIXlt"    "c"              "format"
#> [10] "length<-"      "mean"           "mtfrm"
#> [13] "print"         "range"          "rep"
#> [16] "split"         "summary"        "Summary"
#> [19] "weighted.mean" "xtfrm"
```

Methods which are specific to the subclasses:

```
union(lt_generics, ct_generics)
#>  [1] "["              "[["             "[[<-"
#>  [4] "[<-"            "$<-"            "anyNA"
#>  [7] "as.data.frame" "as.Date"        "as.double"
#> [10] "as.list"       "as.matrix"      "as.POSIXct"
#> [13] "as.vector"     "c"              "duplicated"
#> [16] "format"        "is.finite"      "is.infinite"
#> [19] "is.na"         "is.nan"         "length"
#> [22] "length<-"      "mean"           "mtfrm"
#> [25] "names"         "names<-"        "print"
#> [28] "rep"           "sort"           "summary"
#> [31] "Summary"       "unique"         "weighted.mean"
#> [34] "xtfrm"         "as.POSIXlt"     "range"
#> [37] "split"
```

Let's see an example:

```
s3_dispatch(is.na(dt_lt))
#> => is.na.POSIXlt
#>    is.na.POSIXt
#>    is.na.default
#>  * is.na (internal)

s3_dispatch(is.na(dt_ct))
#>    is.na.POSIXct
#>    is.na.POSIXt
#>    is.na.default
#> => is.na (internal)

s3_dispatch(is.na(dt_t))
#>    is.na.POSIXt
#>    is.na.default
#> => is.na (internal)
```

Methods which are inherited by subclasses from superclass:

```
setdiff(t_generics, union(lt_generics, ct_generics))
#>  [1] "-"            "+"            "all.equal"
#>  [4] "as.character" "Axis"         "cut"
#>  [7] "diff"         "hist"         "is.numeric"
#> [10] "julian"       "Math"         "months"
#> [13] "Ops"          "pretty"       "quantile"
#> [16] "quarters"     "round"        "seq"
#> [19] "str"          "trunc"        "weekdays"
```

Let's see one example generic:

```
s3_dispatch(is.numeric(dt_lt))
#>    is.numeric.POSIXlt
#> => is.numeric.POSIXt
#>    is.numeric.default
#>  * is.numeric (internal)

s3_dispatch(is.numeric(dt_ct))
#>    is.numeric.POSIXct
#> => is.numeric.POSIXt
#>    is.numeric.default
#>  * is.numeric (internal)

s3_dispatch(is.numeric(dt_t))
#> => is.numeric.POSIXt
#>    is.numeric.default
#>  * is.numeric (internal)
```

**Q3.** What do you expect this code to return? What does it actually return? Why?

```
generic2 <- function(x) UseMethod("generic2")
generic2.a1 <- function(x) "a1"
generic2.a2 <- function(x) "a2"
generic2.b <- function(x) {
  class(x) <- "a1"
  NextMethod()
}

generic2(structure(list(), class = c("b", "a2")))
```

**A3.** Naively, we would expect for this code to return `"a1"`, but it actually returns `"a2"`:

```
generic2 <- function(x) UseMethod("generic2")
generic2.a1 <- function(x) "a1"
generic2.a2 <- function(x) "a2"
generic2.b <- function(x) {
  class(x) <- "a1"
  NextMethod()
}

generic2(structure(list(), class = c("b", "a2")))
#> [1] "a2"
```

S3 dispatch explains why:

```
sloop::s3_dispatch(generic2(structure(list(), class = c("b",
 ↪  "a2"))))
#> => generic2.b
#> -> generic2.a2
#>    generic2.default
```

As mentioned in the book, the `UseMethod()` function

> tracks the list of potential next methods with a special variable, which means that modifying the object that's being dispatched upon will have no impact on which method gets called next.

This special variable is `.Class`:

> `.Class` is a character vector of classes used to find the next method. `NextMethod` adds an attribute "previous" to `.Class` giving the `.Class` last used for dispatch, and shifts `.Class` along to that used for dispatch.

So, we can print `.Class` to confirm that adding a new class to `x` indeed doesn't change `.Class`, and therefore dispatch occurs on `"a2"` class:

```
generic2.b <- function(x) {
  message(paste0("before: ", paste0(.Class, collapse = ", ")))
  class(x) <- "a1"
  message(paste0("after: ", paste0(.Class, collapse = ", ")))

  NextMethod()
}
```

```r
invisible(generic2(structure(list(), class = c("b", "a2"))))
#> before: b, a2
#> after: b, a2
```

## 13.6 Dispatch details (Exercises 13.7.5)

**Q1.** Explain the differences in dispatch below:

```r
length.integer <- function(x) 10

x1 <- 1:5
class(x1)
#> [1] "integer"
s3_dispatch(length(x1))
#>  * length.integer
#>    length.numeric
#>    length.default
#> => length (internal)

x2 <- structure(x1, class = "integer")
class(x2)
#> [1] "integer"
s3_dispatch(length(x2))
#> => length.integer
#>    length.default
#>  * length (internal)
```

**A1.** The differences in the dispatch are due to classes of arguments:

```r
s3_class(x1)
#> [1] "integer" "numeric"

s3_class(x2)
#> [1] "integer"
```

`x1` has an implicit class `integer` but it inherits from `numeric`, while `x2` is explicitly assigned the class `integer`.

**Q2.** What classes have a method for the `Math` group generic in base R? Read the source code. How do the methods work?

**A2.** The following classes have a method for the `Math` group generic in base R:

```
s3_methods_generic("Math") %>%
  dplyr::filter(source == "base")
#> # A tibble: 5 x 4
#>   generic class      visible source
#>   <chr>   <chr>      <lgl>   <chr>
#> 1 Math    data.frame TRUE    base
#> 2 Math    Date       TRUE    base
#> 3 Math    difftime   TRUE    base
#> 4 Math    factor     TRUE    base
#> 5 Math    POSIXt     TRUE    base
```

Reading source code for a few of the methods:

`Math.factor()` and `Math.Date()` provide only error message:

```
Math.factor <- function(x, ...) {
  stop(gettextf("%s not meaningful for factors",
  ↪  sQuote(.Generic)))
}

Math.Date <- function(x, ...) {
  stop(gettextf("%s not defined for \"Date\" objects", .Generic),
    domain = NA
  )
}
```

`Math.data.frame()` is defined as follows (except the first line of code, which I have deliberately added):

```
Math.data.frame <- function(x, ...) {
  message(paste0("Environment variable `.Generic` set to: ",
  ↪  .Generic))

  mode.ok <- vapply(x, function(x) {
    is.numeric(x) || is.logical(x) || is.complex(x)
  }, NA)

  if (all(mode.ok)) {
    x[] <- lapply(X = x, FUN = .Generic, ...)
    return(x)
  } else {
    vnames <- names(x)
    if (is.null(vnames)) vnames <- seq_along(x)
    stop(
      "non-numeric-alike variable(s) in data frame: ",
```

```
      paste(vnames[!mode.ok], collapse = ", ")
    )
  }
}
```

As can be surmised from the code: the method checks that all elements are of the same and expected type.

If so, it applies the generic (tracked via the environment variable `.Generic`) to each element of the list of atomic vectors that makes up a data frame:

```
df1 <- data.frame(x = 1:2, y = 3:4)
sqrt(df1)
#> Environment variable `.Generic` set to: sqrt
#>          x        y
#> 1 1.000000 1.732051
#> 2 1.414214 2.000000
```

If not, it produces an error:

```
df2 <- data.frame(x = c(TRUE, FALSE), y = c("a", "b"))
abs(df2)
#> Environment variable `.Generic` set to: abs
#> Error in Math.data.frame(df2): non-numeric-alike variable(s)
↪   in data frame: y
```

**Q3.** `Math.difftime()` is more complicated than I described. Why?

**A3.** `Math.difftime()` source code looks like the following:

```
Math.difftime <- function(x, ...) {
  switch(.Generic,
    "abs" = ,
    "sign" = ,
    "floor" = ,
    "ceiling" = ,
    "trunc" = ,
    "round" = ,
    "signif" = {
      units <- attr(x, "units")
      .difftime(NextMethod(), units)
    },
    ### otherwise :
    stop(gettextf("'%s' not defined for \"difftime\" objects",
     ↪  .Generic),
```

```
      domain = NA
    )
  )
}
```

This group generic is a bit more complicated because it produces an error for some generics, while it works for others.

## 13.7 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  codetools     0.2-20  2024-03-31 [3] CRAN (R 4.4.3)
#>  compiler      4.4.3   2025-02-28 [3] local
#>  crayon        1.5.3   2024-06-20 [1] RSPM
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  dplyr       * 1.1.4   2023-11-17 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  generics      0.1.3   2022-07-05 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
```

```
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  purrr       * 1.0.4   2025-02-05 [1] RSPM
#>  R6            2.6.1   2025-02-15 [1] RSPM
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  sloop       * 1.0.1   2019-02-17 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tibble        3.2.1   2023-03-20 [1] RSPM
#>  tidyselect    1.2.1   2024-03-11 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utf8          1.2.4   2023-10-22 [1] RSPM
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 14

# R6

Loading the needed libraries:

```
library(R6)
```

## 14.1 Classes and methods (Exercises 14.2.6)

**Q1.** Create a bank account R6 class that stores a balance and allows you to deposit and withdraw money. Create a subclass that throws an error if you attempt to go into overdraft. Create another subclass that allows you to go into overdraft, but charges you a fee. Create the superclass and make sure it works as expected.

**A1.** First, let's create a bank account R6 class that stores a balance and allows you to deposit and withdraw money:

```
library(R6)

bankAccount <- R6::R6Class(
  "bankAccount",
  public = list(
    balance = 0,
    initialize = function(balance) {
      self$balance <- balance
    },
    deposit = function(amount) {
      self$balance <- self$balance + amount
      message(paste0("Current balance is: ", self$balance))
      invisible(self)
```

```
    },
    withdraw = function(amount) {
      self$balance <- self$balance - amount
      message(paste0("Current balance is: ", self$balance))
      invisible(self)
    }
  )
)
```

Let's try it out:

```
indra <- bankAccount$new(balance = 100)

indra$deposit(20)
#> Current balance is: 120

indra$withdraw(10)
#> Current balance is: 110
```

Create a subclass that errors if you attempt to overdraw:

```
bankAccountStrict <- R6::R6Class(
  "bankAccountStrict",
  inherit = bankAccount,
  public = list(
    withdraw = function(amount) {
      if (self$balance - amount < 0) {
        stop(
          paste0("Can't withdraw more than your current balance:
           ↪  ", self$balance),
          call. = FALSE
        )
      }

      super$withdraw(amount)
    }
  )
)
```

Let's try it out:

```
Pritesh <- bankAccountStrict$new(balance = 100)
```

```
Pritesh$deposit(20)
#> Current balance is: 120

Pritesh$withdraw(150)
#> Error: Can't withdraw more than your current balance: 120
```

Now let's create a subclass that charges a fee if account is overdrawn:

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

      if (self$balance) {
        self$balance <- self$balance - 10
        message("You're withdrawing more than your current
         ↪  balance. You will be charged a fee of 10 euros.")
      }
    }
  )
)
```

Let's try it out:

```
Mangesh <- bankAccountFee$new(balance = 100)

Mangesh$deposit(20)
#> Current balance is: 120

Mangesh$withdraw(150)
#> Current balance is: -30
#> You're withdrawing more than your current balance. You will be
↪  charged a fee of 10 euros.
```

**Q2.** Create an R6 class that represents a shuffled deck of cards. You should be able to draw cards from the deck with `$draw(n)`, and return all cards to the deck and reshuffle with `$reshuffle()`. Use the following code to make a vector of cards.

```
suit <- c(" ", " ", " ", " ")
value <- c("A", 2:10, "J", "Q", "K")
cards <- paste0(rep(value, 4), suit)
```

**A2.** Let's create needed class that represents a shuffled deck of cards:

```
suit <- c(" ", " ", " ", " ")
value <- c("A", 2:10, "J", "Q", "K")
cards <- paste(rep(value, 4), suit)

Deck <- R6::R6Class(
  "Deck",
  public = list(
    initialize = function(deck) {
      private$cards <- sample(deck)
    },
    draw = function(n) {
      if (n > length(private$cards)) {
        stop(
          paste0("Can't draw more than remaining number of cards:
           ↪   ", length(private$cards)),
          call. = FALSE
        )
      }

      drawn_cards <- sample(private$cards, n)
      private$cards <- private$cards[-which(private$cards %in%
 ↪   drawn_cards)]
      message(paste0("Remaining number of cards: ",
        ↪   length(private$cards)))

      return(drawn_cards)
    },
    reshuffle = function() {
      private$cards <- sample(private$cards)
      invisible(self)
    }
  ),
  private = list(
    cards = NULL
  )
)
```

Let's try it out:

```
myDeck <- Deck$new(cards)

myDeck$draw(4)
#> Remaining number of cards: 48
```

```
#> [1] "2  "  "10  " "9  "  "3  "

myDeck$reshuffle()$draw(5)
#> Remaining number of cards: 43
#> [1] "6  "  "10  " "2  "  "A  "  "8  "

myDeck$draw(50)
#> Error: Can't draw more than remaining number of cards: 43
```

**Q3.** Why can't you model a bank account or a deck of cards with an S3 class?

**A3.** We can't model a bank account or a deck of cards with an `S3` class because instances of these classes are *immutable*.

On the other hand, `R6` classes encapsulate data and represent its *state*, which can change over the course of object's lifecycle. In other words, these objects are *mutable* and well-suited to model a bank account.

**Q4.** Create an R6 class that allows you to get and set the current time zone. You can access the current time zone with `Sys.timezone()` and set it with `Sys.setenv(TZ = "newtimezone")`. When setting the time zone, make sure the new time zone is in the list provided by `OlsonNames()`.

**A4.** Here is an `R6` class that manages the current time zone:

```
CurrentTimeZone <- R6::R6Class("CurrentTimeZone",
  public = list(
    setTimeZone = function(tz) {
      stopifnot(tz %in% OlsonNames())
      Sys.setenv(TZ = tz)
    },
    getTimeZone = function() {
      Sys.timezone()
    }
  )
)
```

Let's try it out:

```
myCurrentTimeZone <- CurrentTimeZone$new()

myCurrentTimeZone$getTimeZone()
#> [1] "UTC"

myCurrentTimeZone$setTimeZone("Asia/Kolkata")
myCurrentTimeZone$getTimeZone()
```

```
#> [1] "Asia/Kolkata"

myCurrentTimeZone$setTimeZone("Europe/Berlin")
```

**Q5.** Create an R6 class that manages the current working directory. It should have `$get()` and `$set()` methods.

**A5.** Here is an `R6` class that manages the current working directory:

```
ManageDirectory <- R6::R6Class("ManageDirectory",
  public = list(
    setWorkingDirectory = function(dir) {
      setwd(dir)
    },
    getWorkingDirectory = function() {
      getwd()
    }
  )
)
```

Let's create an instance of this class and check if the methods work as expected:

```
myDirManager <- ManageDirectory$new()

# current working directory
myDirManager$getWorkingDirectory()

# change and check if that worked
myDirManager$setWorkingDirectory("..")
myDirManager$getWorkingDirectory()

# revert this change
myDirManager$setWorkingDirectory("/Advanced-R-exercises")
```

**Q6.** Why can't you model the time zone or current working directory with an S3 class?

**A6.** Same as answer to **Q3**:

Objects that represent these real-life entities need to be mutable and `S3` class instances are not mutable.

**Q7.** What base type are R6 objects built on top of? What attributes do they have?

**A7.** Let's create an example class and create instance of that class:

```r
Example <- R6::R6Class("Example")
myExample <- Example$new()
```

The `R6` objects are built on top of environment:

```r
typeof(myExample)
#> [1] "environment"

rlang::env_print(myExample)
#> <environment: 0x55fec2028768> [L]
#> Parent: <environment: empty>
#> Class: Example, R6
#> Bindings:
#> * .__enclos_env__: <env>
#> * clone: <fn> [L]
```

And it has only `class` attribute, which is a character vector with the `"R6"` being the last element and the superclasses being other elements:

```r
attributes(myExample)
#> $class
#> [1] "Example" "R6"
```

## 14.2 Controlling access (Exercises 14.3.3)

**Q1.** Create a bank account class that prevents you from directly setting the account balance, but you can still withdraw from and deposit to. Throw an error if you attempt to go into overdraft.

**A1.** Here is a bank account class that satisfies the specified requirements:

```r
SafeBankAccount <- R6::R6Class(
  classname = "SafeBankAccount",
  public = list(
    deposit = function(deposit_amount) {
      private$.balance <- private$.balance + deposit_amount
      print(paste("Current balance:", private$.balance))

      invisible(self)
    },
    withdraw = function(withdrawal_amount) {
      if (withdrawal_amount > private$.balance) {
```

```
        stop("You can't withdraw more than your current
         ↪  balance.", call. = FALSE)
      }

      private$.balance <- private$.balance - withdrawal_amount
      print(paste("Current balance:", private$.balance))

      invisible(self)
    }
  ),
  private = list(
    .balance = 0
  )
)
```

Let's check if it works as expected:

```
mySafeBankAccount <- SafeBankAccount$new()

mySafeBankAccount$deposit(100)
#> [1] "Current balance: 100"

mySafeBankAccount$withdraw(50)
#> [1] "Current balance: 50"

mySafeBankAccount$withdraw(100)
#> Error: You can't withdraw more than your current balance.
```

**Q2.** Create a class with a write-only `$password` field. It should have `$check_password(password)` method that returns `TRUE` or `FALSE`, but there should be no way to view the complete password.

**A2.** Here is an implementation of the class with the needed properties:

```
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() {
      cat("Password: XXXX")

      # for method chaining
      invisible(self)
    }
  ),
  private = list(
    .password = NULL
  )
)

myCheck <- checkCredentials$new()

myCheck$set_password("1234")
print(myCheck)
#> Password: XXXX

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"
```

**Q3.** Extend the `Rando` class with another active binding that allows you to access the previous random value. Ensure that active binding is the only way to access the value.

**A3.** Here is a modified version of the `Rando` class to meet the specified requirements:

```r
Rando <- R6::R6Class("Rando",
  active = list(
    random = function(value) {
      if (missing(value)) {
        newValue <- runif(1)
        private$.previousRandom <- private$.currentRandom
        private$.currentRandom <- newValue
        return(private$.currentRandom)
      } else {
        stop("Can't set `$random`", call. = FALSE)
      }
    },
    previousRandom = function(value) {
      if (missing(value)) {
        if (is.null(private$.previousRandom)) {
          message("No random value has been generated yet.")
        } else {
          return(private$.previousRandom)
        }
      } else {
        stop("Can't set `$previousRandom`", call. = FALSE)
      }
    }
  ),
  private = list(
    .currentRandom = NULL,
    .previousRandom = NULL
  )
)
```

Let's try it out:

```r
myRando <- Rando$new()

# first time
myRando$random
#> [1] 0.5549124
myRando$previousRandom
#> No random value has been generated yet.
#> NULL

# second time
myRando$random
#> [1] 0.3482785
myRando$previousRandom
```

```
#> [1] 0.5549124

# third time
myRando$random
#> [1] 0.2187275
myRando$previousRandom
#> [1] 0.3482785
```

**Q4.** Can subclasses access private fields/methods from their parent? Perform an experiment to find out.

**A4.** Unlike common OOP in other languages (e.g. C++), R6 subclasses (or derived classes) 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"
```

## 14.3 Reference semantics (Exercises 14.4.4)

**Q1.** Create a class that allows you to write a line to a specified file. You should open a connection to the file in `$initialize()`, append a line using `cat()` in `$append_line()`, and close the connection in `$finalize()`.

**A1.** Here is a class that allows you to write a line to a specified file:

```
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) max used (Mb)
#> Ncells  689553 36.9    1411312 75.4  1411312 75.4
#> Vcells 1293884  9.9    8388608 64.0  2539021 19.4
```

## 14.4 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       Europe/Berlin
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  R6          * 2.6.1   2025-02-15 [1] RSPM
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
```

```
#>  tools          4.4.3   2025-02-28 [3] local
#>  utils        * 4.4.3   2025-02-28 [3] local
#>  vctrs          0.6.5   2023-12-01 [1] RSPM
#>  xfun           0.51    2025-02-19 [1] RSPM
#>  yaml           2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 15

# S4

## 15.1 Basics (Exercises 15.2.1)

---

**Q1.** `lubridate::period()` returns an S4 class. What slots does it have? What class is each slot? What accessors does it provide?

**A1.** Let's first create an instance of `Period` class:

```
library(lubridate)
x <- lubridate::period(c(2, 43, 6), c("hour", "second",
 ↪  "minute"))
x
#> [1] "2H 6M 43S"
```

It has the following slots:

```
slotNames(x)
#> [1] ".Data"  "year"   "month"  "day"    "hour"   "minute"
```

Additionally, the base type of each slot (`numeric`) can be seen in `str()` output:

```
str(x)
#> Formal class 'Period' [package "lubridate"] with 6 slots
#>   ..@ .Data : num 43
#>   ..@ year  : num 0
#>   ..@ month : num 0
```

```
#>   ..@ day    : num 0
#>   ..@ hour   : num 2
#>   ..@ minute: num 6
```

The `{lubridate}` package provides accessors for all slots:

```
year(x)
#> [1] 0
month(x)
#> [1] 0
day(x)
#> [1] 0
hour(x)
#> [1] 2
minute(x)
#> [1] 6
second(x)
#> [1] 43
```

---

**Q2.** What other ways can you find help for a method? Read `?"?"` and summarise the details.

**A2.** The `"?"` operator allows access to documentation in three ways. To demonstrate different ways to access documentation, let's define a new `S4` class.

```
pow <- function(x, exp) c(x, exp)
setGeneric("pow")
#> [1] "pow"
setMethod("pow", c("numeric", "numeric"), function(x, exp) x^exp)
```

Ways to access documentation:

- The general documentation for a generic can be found with `?topic`:

```
?pow
```

- The expression `type?topic` will look for the overall documentation methods for the function `f`.

```
?pow # produces the function documentation

methods?pow # looks for the overall methods documentation
```

---

## 15.2 Classes (Exercises 15.3.6)

---

**Q1.** Extend the Person class with fields to match `utils::person()`. Think about what slots you will need, what class each slot should have, and what you'll need to check in your validity method.

**A1.** The code below extends the `Person` class described in the book to match more closely with `utils::person()`.

```
setClass("Person",
  slots = c(
    age     = "numeric",
    given   = "character",
    family  = "character",
    middle  = "character",
    email   = "character",
    role    = "character",
    comment = "character"
  ),
  prototype = list(
    age     = NA_real_,
    given   = NA_character_,
    family  = NA_character_,
    middle  = NA_character_,
    email   = NA_character_,
    role    = NA_character_,
    comment = NA_character_
  )
)

# Helper function to create an instance of the `Person` class
Person <- function(given,
                   family,
                   middle = NA_character_,
                   age = NA_real_,
```

```
                   email = NA_character_,
                   role = NA_character_,
                   comment = NA_character_) {
  age <- as.double(age)

  new("Person",
    age     = age,
    given   = given,
    family  = family,
    middle  = middle,
    email   = email,
    role    = role,
    comment = comment
  )
}

# Validator to ensure that each slot is of length one and that
↪  the specified
# role is one of the possible roles
setValidity("Person", function(object) {
  invalid_length <- NULL
  slot_lengths <- c(
    length(object@age),
    length(object@given),
    length(object@middle),
    length(object@family),
    length(object@email),
    length(object@comment)
  )

  if (any(slot_lengths > 1L)) {
    invalid_length <- "\nFollowing slots must be of length 1:\n
↪   @age, @given, @family, @middle, @email, @comment"
  }

  possible_roles <- c(
    NA_character_,
    "aut",
    "com",
    "cph",
    "cre",
    "ctb",
    "ctr",
    "dtc",
    "fnd",
    "rev",
```

```
    "ths",
    "trl"
  )

  if (any(!object@role %in% possible_roles)) {
    invalid_length <- paste(
      invalid_length,
      "\nSlot @role(s) must be one of the following:\n",
      paste(possible_roles, collapse = ", ")
    )
  }

  if (!is.null(invalid_length)) {
    return(invalid_length)
  } else {
    return(TRUE)
  }
})
#> Class "Person" [in ".GlobalEnv"]
#>
#> Slots:
#>
#> Name:         age     given    family    middle     email
#> Class:    numeric character character character character
#>
#> Name:       role   comment
#> Class: character character
```

Let's make sure that validation works as expected:

```
# length of first argument not 1
Person(c("Indrajeet", "Surendra"), "Patil")
#> Error in validObject(.Object): invalid class "Person" object:
#> Following slots must be of length 1:
#>  @age, @given, @family, @middle, @email, @comment

# role not recognized
Person("Indrajeet", "Patil", role = "xyz")
#> Error in validObject(.Object): invalid class "Person" object:
 ↪
#> Slot @role(s) must be one of the following:
#>  NA, aut, com, cph, cre, ctb, ctr, dtc, fnd, rev, ths, trl

# all okay
Person("Indrajeet", "Patil", role = c("aut", "cph"))
```

```
#> An object of class "Person"
#> Slot "age":
#> [1] NA
#>
#> Slot "given":
#> [1] "Indrajeet"
#>
#> Slot "family":
#> [1] "Patil"
#>
#> Slot "middle":
#> [1] NA
#>
#> Slot "email":
#> [1] NA
#>
#> Slot "role":
#> [1] "aut" "cph"
#>
#> Slot "comment":
#> [1] NA
```

---

**Q2.** What happens if you define a new S4 class that doesn't have any slots? (Hint: read about virtual classes in `?setClass`.)

**A2.** If you define a new `S4` class that doesn't have any slots, it will create *virtual* classes:

```
setClass("Empty")

isVirtualClass("Empty")
#> [1] TRUE
```

You can't create an instance of this class:

```
new("Empty")
#> Error in new("Empty"): trying to generate an object from a
↪  virtual class ("Empty")
```

So how is this useful? As mentioned in `?setClass` docs:

> Classes exist for which no actual objects can be created, the virtual classes.
>
> The most common and useful form of virtual class is the class union, a virtual class that is defined in a call to `setClassUnion()` rather than a call to `setClass()`.

So virtual classes can still be inherited:

```
setClass("Nothing", contains = "Empty")
```

In addition to not specifying any slots, here is another way to create virtual classes:

> Calls to `setClass()` will also create a virtual class, either when only the Class argument is supplied (no slots or superclasses) or when the `contains=` argument includes the special class name `"VIRTUAL"`.

---

**Q3.** Imagine you were going to reimplement factors, dates, and data frames in S4. Sketch out the `setClass()` calls that you would use to define the classes. Think about appropriate `slots` and `prototype`.

**A3.** The reimplementation of following classes in `S4` might have definitions like the following.

- `factor`

For simplicity, we won't provide all options that `factor()` provides. Note that `x` has pseudo-class `ANY` to accept objects of any type.

```
setClass("Factor",
  slots = c(
    x       = "ANY",
    levels  = "character",
    ordered = "logical"
  ),
  prototype = list(
    x       = character(),
    levels  = character(),
    ordered = FALSE
  )
)
```

```r
new("Factor", x = letters[1:3], levels = LETTERS[1:3])
#> An object of class "Factor"
#> Slot "x":
#> [1] "a" "b" "c"
#>
#> Slot "levels":
#> [1] "A" "B" "C"
#>
#> Slot "ordered":
#> [1] FALSE

new("Factor", x = 1:3, levels = letters[1:3])
#> An object of class "Factor"
#> Slot "x":
#> [1] 1 2 3
#>
#> Slot "levels":
#> [1] "a" "b" "c"
#>
#> Slot "ordered":
#> [1] FALSE

new("Factor", x = c(TRUE, FALSE, TRUE), levels = c("x", "y",
↪  "x"))
#> An object of class "Factor"
#> Slot "x":
#> [1]  TRUE FALSE  TRUE
#>
#> Slot "levels":
#> [1] "x" "y" "x"
#>
#> Slot "ordered":
#> [1] FALSE
```

- `Date`

Just like the base-R version, this will have only integer values.

```r
setClass("Date2",
  slots = list(
    data = "integer"
  ),
  prototype = list(
    data = integer()
```

```
  )
)

new("Date2", data = 1342L)
#> An object of class "Date2"
#> Slot "data":
#> [1] 1342
```

- `data.frame`

The tricky part is supporting the `...` argument of `data.frame()`. For this, we can let the users pass a (named) list.

```
setClass("DataFrame",
  slots = c(
    data      = "list",
    row.names = "character"
  ),
  prototype = list(
    data      = list(),
    row.names = character(0L)
  )
)

new("DataFrame", data = list(x = c("a", "b"), y = c(1L, 2L)))
#> An object of class "DataFrame"
#> Slot "data":
#> $x
#> [1] "a" "b"
#>
#> $y
#> [1] 1 2
#>
#>
#> Slot "row.names":
#> character(0)
```

---

## 15.3 Generics and methods (Exercises 15.4.5)

---

**Q1.** Add `age()` accessors for the `Person` class.

**A1.** We first should define a generic and then a method for our class:

```
Indra <- Person("Indrajeet", "Patil", role = c("aut", "cph"), age
↪  = 34)

setGeneric("age", function(x) standardGeneric("age"))
#> [1] "age"
setMethod("age", "Person", function(x) x@age)

age(Indra)
#> [1] 34
```

---

**Q2.** In the definition of the generic, why is it necessary to repeat the name of the generic twice?

**A2.** Let's look at the generic we just defined; the generic name `"age"` is repeated twice.

```
setGeneric(name = "age", def = function(x)
↪  standardGeneric("age"))
```

This is because:

- the `"age"` passed to argument `name` provides the name for the generic
- the `"age"` passed to argument `def` supplies the method dispatch

This is reminiscent of how we defined `S3` generic, where we also had to repeat the name twice:

```
age <- function(x) {
  UseMethod("age")
}
```

---

**Q3.** Why does the `show()` method defined in Section Show method use `is(object)[[1]]`? (Hint: try printing the employee subclass.)

**A3.** Because we wish to define `show()` method for a specific class, we need to disregard the other super-/sub-classes.

Always using the first element ensures that the method will be defined for the class in question:

```r
Alice <- new("Employee")

is(Alice)
#> [1] "Employee" "Person"

is(Alice)[[1]]
#> [1] "Employee"
```

---

**Q4.** What happens if you define a method with different argument names to the generic?

**A4.** Let's experiment with the method we defined in **Q1.** to study this behavior.

The original method that worked as expected since the argument name between generic and method matched:

```r
setMethod("age", "Person", function(x) x@age)
```

If this is not the case, we either get a warning or get an error depending on which and how many arguments have been specified:

```r
setMethod("age", "Person", function(object) object@age)
#> Warning: For function 'age', signature 'Person': argument
#> in method definition changed from (object) to (x)

setMethod("age", "Person", function(object, x) object@age)
#> Error in rematchDefinition(definition, fdef, mnames, fnames,
↪  signature): methods can add arguments to the generic 'age'
↪  only if '...' is an argument to the generic

setMethod("age", "Person", function(...) ...elt(1)@age)
#> Warning: For function 'age', signature 'Person': argument
#> in method definition changed from (...) to (x)

setMethod("age", "Person", function(x, ...) x@age)
#> Error in rematchDefinition(definition, fdef, mnames, fnames,
↪  signature): methods can add arguments to the generic 'age'
↪  only if '...' is an argument to the generic
```

---

## 15.4 Method dispatch (Exercises 15.5.5)

---

**Q1.** Draw the method graph for `f(, )`.

**A1.** I don't how to prepare the visual illustrations used in the book, so I am linking to the illustration in the official solution manual:

---

**Q2.** Draw the method graph for `f(, , )`.

**A2.** I don't have access to the software used to prepare the visual illustrations used in the book, so I am linking to the illustration in the official solution manual:

---

**Q3.** Take the last example which shows multiple dispatch over two classes that use multiple inheritance. What happens if you define a method for all terminal classes? Why does method dispatch not save us much work here?

**A3.** Because one class has distance of 2 to all terminal nodes and the other four have distance of 1 to two terminal nodes each, this will introduce ambiguity.

Method dispatch not save us much work here because to resolve this ambiguity we have to define five more methods (one per class combination).

---

## 15.5 `S4` and `S3` (Exercises 15.6.3)

---

**Q1.** What would a full `setOldClass()` definition look like for an ordered factor (i.e. add `slots` and `prototype` the definition above)?

**A1.** We can register the old-style/`S3` `ordered` class to a formally defined class using `setOldClass()`.

```
setClass("factor",
  contains = "integer",
  slots = c(
    levels = "character"
  ),
  prototype = structure(
    integer(),
    levels = character()
  )
)
setOldClass("factor", S4Class = "factor")
#> Warning in rm(list = what, pos = classWhere): object
#> '.__C__factor' not found

setClass("Ordered",
  contains = "factor",
  slots = c(
    levels  = "character",
    ordered = "logical"
  ),
  prototype = structure(
    integer(),
    levels  = character(),
    ordered = logical()
  )
)

setOldClass("ordered", S4Class = "Ordered")
```

Let's use it to see if it works as expected.

```
x <- new("Ordered", 1L:4L, levels = letters[1:4], ordered = TRUE)

x
#> Object of class "Ordered"
#> [1] a b c d
#> Levels: a b c d
#> Slot "ordered":
#> [1] TRUE

str(x)
#> Formal class 'Ordered' [package ".GlobalEnv"] with 4 slots
#>   ..@ .Data   : int [1:4] 1 2 3 4
#>   ..@ levels  : chr [1:4] "a" "b" "c" "d"
#>   ..@ ordered : logi TRUE
```

```
#>   ..@ .S3Class: chr "factor"

class(x)
#> [1] "Ordered"
#> attr(,"package")
#> [1] ".GlobalEnv"
```

---

**Q2.** Define a `length` method for the `Person` class.

**A2.** Because our `Person` class can be used to create objects that represent multiple people, let's say the `length()` method returns how many persons are in the object.

```
Friends <- new("Person", name = c("Vishu", "Aditi"))
```

We can define an `S3` method for this class:

```
length.Person <- function(x) length(x@name)

length(Friends)
#> [1] 2
```

Alternatively, we can also write `S4` method:

```
setMethod("length", "Person", function(x) length(x@name))

length(Friends)
#> [1] 2
```

---

## 15.6 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
```

```
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
 ↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  generics      0.1.3   2022-07-05 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  lubridate   * 1.9.4   2024-12-08 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  timechange    0.3.0   2024-01-18 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
```

```
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ---------------------------------------------------------
```

# Chapter 16

# Trade-offs

No exercises.

# Chapter 17

# Big Picture

No exercises.

# Chapter 18

# Expressions

Attaching the needed libraries:

```r
library(rlang, warn.conflicts = FALSE)
library(lobstr, warn.conflicts = FALSE)
```

## 18.1 Abstract syntax trees (Exercises 18.2.4)

**Q1.** Reconstruct the code represented by the trees below:

```
#> f
#>  g
#>    h
#> `+`
#>  `+`
#>   1
#>   2
#> 3
#> `*`
#>  `(`
#>    `+`
#>     x
#>     y
#> z
```

**A1.** Below is the reconstructed code.

```
f(g(h()))
1 + 2 + 3
(x + y) * z
```

We can confirm it by drawing ASTs for them:

```
ast(f(g(h())))
#> f
#>   g
#>     h

ast(1 + 2 + 3)
#> `+`
#>   `+`
#>    1
#>    2
#> 3

ast((x + y) * z)
#> `*`
#>   `(`
#>     `+`
#>      x
#>      y
#> z
```

**Q2.** Draw the following trees by hand and then check your answers with `ast()`.

```
f(g(h(i(1, 2, 3))))
f(1, g(2, h(3, i())))
f(g(1, 2), h(3, i(4, 5)))
```

**A2.** Successfully drawn by hand. Checking using `ast()`:

```
ast(f(g(h(i(1, 2, 3)))))
#> f
#>   g
#>     h
#>       i
#>        1
#>        2
#>        3
```

```
ast(f(1, g(2, h(3, i()))))
#> f
#> 1
#>  g
#>   2
#>    h
#>     3
#>      i

ast(f(g(1, 2), h(3, i(4, 5))))
#> f
#>  g
#>   1
#>   2
#>  h
#>   3
#>    i
#>     4
#>     5
```

**Q3.** What's happening with the ASTs below? (Hint: carefully read `?"^"`.)

```
ast(`x` + `y`)
#>  `+`
#> x
#> y
ast(x**y)
#>  `^`
#> x
#> y
ast(1 -> x)
#>  `<-`
#> x
#> 1
```

**A3.** The `str2expression()` helps make sense of these ASTs.

The non-syntactic names are parsed to names. Thus, backticks have been removed in the AST.

```
str2expression("`x` + `y`")
#> expression(x + y)
```

As mentioned in the docs for `^`:

> ** is translated in the parser to ^

```
str2expression("x**y")
#> expression(x^y)
```

The rightward assignment is parsed to leftward assignment:

```
str2expression("1 -> x")
#> expression(x <- 1)
```

**Q4.** What is special about the AST below?

```
ast(function(x = 1, y = 2) {})
#>  `function`
#>   x = 1
#>    y = 2
#>    `{`
#>   NULL
```

**A4.** As mentioned in this section:

> Like all objects in R, functions can also possess any number of additional `attributes()`. One attribute used by base R is `srcref`, short for source reference. It points to the source code used to create the function. The `srcref` is used for printing because, unlike `body()`, it contains code comments and other formatting.

Therefore, the last leaf in this AST, although not specified in the function call, represents source reference attribute.

**Q5.** What does the call tree of an `if` statement with multiple `else if` conditions look like? Why?

**A5.** There is nothing special about this tree. It just shows the nested loop structure inherent to code with `if` and multiple `else if` statements.

```
ast(if (FALSE) 1 else if (FALSE) 2 else if (FALSE) 3 else 4)
#>  `if`
#>  FALSE
#>  1
#>   `if`
#>    FALSE
#>    2
#>     `if`
```

```
#>       FALSE
#>       3
#>       4
```

## 18.2 Expressions (Exercises 18.3.5)

**Q1.** Which two of the six types of atomic vector can't appear in an expression? Why? Similarly, why can't you create an expression that contains an atomic vector of length greater than one?

**A1.** Out of the six types of atomic vectors, the two that can't appear in an expression are: complex and raw.

Complex numbers are created via a **function call** (using `+`), as can be seen by its AST:

```
x_complex <- expr(1 + 1i)
typeof(x_complex)
#> [1] "language"

ast(1 + 1i)
#>  `+`
#>  1
#>  1i
```

Similarly, for raw vectors (using `raw()`):

```
x_raw <- expr(raw(2))
typeof(x_raw)
#> [1] "language"

ast(raw(2))
#>  raw
#>  2
```

Contrast this with other atomic vectors:

```
x_int <- expr(2L)
typeof(x_int)
#> [1] "integer"

ast(2L)
#> 2L
```

For the same reason, you can't you create an expression that contains an atomic vector of length greater than one since that itself is a function call that uses `c()` function:

```
x_vec <- expr(c(1, 2))
typeof(x_vec)
#> [1] "language"

ast(c(1, 2))
#>  c
#>  1
#>  2
```

**Q2.** What happens when you subset a call object to remove the first element? e.g. `expr(read.csv("foo.csv", header = TRUE))[-1]`. Why?

**A2.** A captured function call like the following creates a call object:

```
expr(read.csv("foo.csv", header = TRUE))
#> read.csv("foo.csv", header = TRUE)

typeof(expr(read.csv("foo.csv", header = TRUE)))
#> [1] "language"
```

As mentioned in the respective section:

> The first element of the call object is the function position.

Therefore, when the first element in the call object is removed, the next one moves in the function position, and we get the observed output:

```
expr(read.csv("foo.csv", header = TRUE))[-1]
#> "foo.csv"(header = TRUE)
```

**Q3.** Describe the differences between the following call objects.

```
x <- 1:10
call2(median, x, na.rm = TRUE)
call2(expr(median), x, na.rm = TRUE)
call2(median, expr(x), na.rm = TRUE)
call2(expr(median), expr(x), na.rm = TRUE)
```

**A4.** The differences in the constructed call objects are due to the different *type* of arguments supplied to first two parameters in the `call2()` function.

Types of arguments supplied to `.fn`:

```r
typeof(median)
#> [1] "closure"
typeof(expr(median))
#> [1] "symbol"
```

Types of arguments supplied to the dynamic dots:

```r
x <- 1:10
typeof(x)
#> [1] "integer"
typeof(expr(x))
#> [1] "symbol"
```

The following outputs can be understood using the following properties:

- when `.fn` argument is a `closure`, that function is inlined in the constructed function call
- when `x` is not a symbol, its value is passed to the function call

```r
x <- 1:10

call2(median, x, na.rm = TRUE)
#> (function (x, na.rm = FALSE, ...)
#> UseMethod("median"))(1:10, na.rm = TRUE)

call2(expr(median), x, na.rm = TRUE)
#> median(1:10, na.rm = TRUE)

call2(median, expr(x), na.rm = TRUE)
#> (function (x, na.rm = FALSE, ...)
#> UseMethod("median"))(x, na.rm = TRUE)

call2(expr(median), expr(x), na.rm = TRUE)
#> median(x, na.rm = TRUE)
```

Importantly, all of the constructed call objects evaluate to give the same result:

```r
x <- 1:10

eval(call2(median, x, na.rm = TRUE))
#> [1] 5.5

eval(call2(expr(median), x, na.rm = TRUE))
```

```
#> [1] 5.5

eval(call2(median, expr(x), na.rm = TRUE))
#> [1] 5.5

eval(call2(expr(median), expr(x), na.rm = TRUE))
#> [1] 5.5
```

**Q4.** `call_standardise()` doesn't work so well for the following calls. Why? What makes `mean()` special?

```
call_standardise(quote(mean(1:10, na.rm = TRUE)))
#> Warning: `call_standardise()` is deprecated as of rlang 0.4.11
#> This warning is displayed once every 8 hours.
#> mean(x = 1:10, na.rm = TRUE)
call_standardise(quote(mean(n = T, 1:10)))
#> mean(x = 1:10, n = T)
call_standardise(quote(mean(x = 1:10, , TRUE)))
#> mean(x = 1:10, , TRUE)
```

**A4.** This is because of the ellipsis in `mean()` function signature:

```
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x56303fdd80c0>
#> <environment: namespace:base>
```

As mentioned in the respective section:

> If the function uses `...` it's not possible to standardise all arguments.

`mean()` is an S3 generic and the dots are passed to underlying S3 methods.

So, the output can be improved using a specific method. For example:

```
call_standardise(quote(mean.default(n = T, 1:10)))
#> mean.default(x = 1:10, na.rm = T)
```

**Q5.** Why does this code not make sense?

```
x <- expr(foo(x = 1))
names(x) <- c("x", "y")
```

**A5.** This doesn't make sense because the first position in a call object is reserved for function (function position), and so assigning names to this element will just be ignored by R:

```
x <- expr(foo(x = 1))
x
#> foo(x = 1)

names(x) <- c("x", "y")
x
#> foo(y = 1)
```

**Q6.** Construct the expression `if(x > 1) "a" else "b"` using multiple calls to `call2()`. How does the code structure reflect the structure of the AST?

**A6.** Using multiple calls to construct the required expression:

```
x <- 5
call_obj1 <- call2(">", expr(x), 1)
call_obj1
#> x > 1

call_obj2 <- call2("if", cond = call_obj1, cons.expr = "a",
 ↪  alt.expr = "b")
call_obj2
#> if (x > 1) "a" else "b"
```

This construction follows from the prefix form of this expression, revealed by its AST:

```
ast(if (x > 1) "a" else "b")
#>  `if`
#>   `>`
#>    x
#>    1
#>   "a"
#>   "b"
```

## 18.3 Parsing and grammar (Exercises 18.4.4)

**Q1.** R uses parentheses in two slightly different ways as illustrated by these two calls:

```
f((1))
`(`(1 + 1)
```

Compare and contrast the two uses by referencing the AST.

**A1.** Let's first have a look at the AST:

```
ast(f((1)))
#> f
#>   `(`
#>    1
ast(`(`(1 + 1))
#> `(`
#>   `+`
#>    1
#>    1
```

As, you can see `(` is being used in two separate ways:

- As a function in its own right ``"`(`"``
- As part of the prefix syntax (`f()`)

This is why, in the AST for `f((1))`, we see only one ``"`(`"`` (the first use case), and not for `f()`, which is part of the function syntax (the second use case).

**Q2.** `=` can also be used in two ways. Construct a simple example that shows both uses.

**A2.** Here is a simple example illustrating how `=` can also be used in two ways:

- for assignment
- for named arguments in function calls

```
m <- mean(x = 1)
```

We can also have a look at its AST:

```
ast({
  m <- mean(x = 1)
})
#> `{`
#>  `<-`
#>   m
#>    mean
#>     x = 1
```

**Q3.** Does `-2^2` yield 4 or -4? Why?

**A3.** The expression `-2^2` evaluates to -4 because the operator `^` has higher precedence than the unary `-` operator:

```
-2^2
#> [1] -4
```

The same can also be seen by its AST:

```
ast(-2^2)
#> `-`
#>  `^`
#>   2
#>   2
```

A less confusing way to write this would be:

```
-(2^2)
#> [1] -4
```

**Q4.** What does `!1 + !1` return? Why?

**A3.** The expression `!1 + !1` evaluates to FALSE.

This is because the `!` operator has higher precedence than the unary `+` operator. Thus, `!1` evaluates to `FALSE`, which is added to `1 + FALSE`, which evaluates to `1`, and then logically negated to `!1`, or `FALSE`.

This can be easily seen by its AST:

```
ast(!1 + !1)
#> `!`
#>  `+`
#>   1
#>    `!`
#>     1
```

**Q5.** Why does `x1 <- x2 <- x3 <- 0` work? Describe the two reasons.

**A5.** There are two reasons why the following works as expected:

```
x1 <- x2 <- x3 <- 0
```

- The `<-` operator is right associative.

Therefore, the order of assignment here is:

```
(x3 <- 0)
(x2 <- x3)
(x1 <- x2)
```

- The `<-` operator invisibly returns the assigned value.

```
(x <- 1)
#> [1] 1
```

This is easy to surmise from its AST:

```
ast(x1 <- x2 <- x3 <- 0)
#>   `<-`
#>   x1
#>    `<-`
#>     x2
#>      `<-`
#>       x3
#>       0
```

**Q6.** Compare the ASTs of `x + y %+% z` and `x ^ y %+% z`. What have you learned about the precedence of custom infix functions?

**A6.** Looking at the ASTs for these expressions,

```
ast(x + y %+% z)
#>   `+`
#>   x
#>    `%+%`
#>     y
#>     z

ast(x^y %+% z)
```

```
#>   `%+%`
#>    `^`
#>    x
#>    y
#>   z
```

we can say that the custom infix operator `%+%` has:

- higher precedence than the `+` operator
- lower precedence than the `^` operator

**Q7.** What happens if you call `parse_expr()` with a string that generates multiple expressions? e.g. `parse_expr("x + 1; y + 1")`

**A7.** It produced an error:

```
parse_expr("x + 1; y + 1")
#> Error in `parse_expr()`:
#> ! `x` must contain exactly 1 expression, not 2.
```

This is expected based on the docs:

> parse_expr() returns one expression. If the text contains more than one expression (separated by semicolons or new lines), an error is issued.

We instead need to use `parse_exprs()`:

```
parse_exprs("x + 1; y + 1")
#> [[1]]
#> x + 1
#> 
#> [[2]]
#> y + 1
```

**Q8.** What happens if you attempt to parse an invalid expression? e.g. `"a +"` or `"f())"`.

**A8.** An invalid expression produces an error:

```
parse_expr("a +")
#> Error in parse(text = x, keep.source = FALSE): <text>:2:0:
↪  unexpected end of input
```

```
#> 1: a +
#>     ^

parse_expr("f())")
#> Error in parse(text = x, keep.source = FALSE): <text>:1:4:
↪  unexpected ')'
#> 1: f())
#>        ^
```

Since the underlying `parse()` function produces an error:

```
parse(text = "a +")
#> Error in parse(text = "a +"): <text>:2:0: unexpected end of
↪  input
#> 1: a +
#>     ^

parse(text = "f())")
#> Error in parse(text = "f())"): <text>:1:4: unexpected ')'
#> 1: f())
#>        ^
```

**Q9.** `deparse()` produces vectors when the input is long. For example, the following call produces a vector of length two:

```
expr <- expr(g(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))
deparse(expr)
```

What does `expr_text()` do instead?

**A9.** The only difference between `deparse()` and `expr_text()` is that the latter turns the (possibly multi-line) expression into a single string.

```
expr <- expr(g(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))

deparse(expr)
#> [1] "g(a + b + c + d + e + f + g + h + i + j + k + l + m + n +
↪  o + "
#> [2] "    p + q + r + s + t + u + v + w + x + y + z)"

expr_text(expr)
#> [1] "g(a + b + c + d + e + f + g + h + i + j + k + l + m + n +
↪  o + \n    p + q + r + s + t + u + v + w + x + y + z)"
```

**Q10.** `pairwise.t.test()` assumes that `deparse()` always returns a length one character vector. Can you construct an input that violates this expectation? What happens?

**A10** Since R 4.0, it is not possible to violate this expectation since the new implementation produces a single string no matter the input:

> New function `deparse1()` produces one string, wrapping `deparse()`, to be used typically in `deparse1(substitute(*))`

## 18.4 Walking AST with recursive functions (Exercises 18.5.3)

**Q1.** `logical_abbr()` returns `TRUE` for `T(1, 2, 3)`. How could you modify `logical_abbr_rec()` so that it ignores function calls that use `T` or `F`?

**A1.** To avoid function calls that use `T` or `F`, we just need to ignore the function position in call objects:

Let's try it out:

```
logical_abbr_rec(expr(T(1, 2, 3)))
#> [1] FALSE

logical_abbr_rec(expr(F(1, 2, 3)))
#> [1] FALSE

logical_abbr_rec(expr(T))
#> [1] TRUE

logical_abbr_rec(expr(F))
#> [1] TRUE
```

**Q2.** `logical_abbr()` works with expressions. It currently fails when you give it a function. Why? How could you modify `logical_abbr()` to make it work? What components of a function will you need to recurse over?

```
logical_abbr(function(x = TRUE) {
  g(x + T)
})
```

**A2.** Surprisingly, `logical_abbr()` currently doesn't fail with closures:

To see why, let's see what type of object is produced when we capture user provided closure:

```
print_enexpr <- function(.f) {
  print(typeof(enexpr(.f)))
  print(is.call(enexpr(.f)))
}

print_enexpr(function(x = TRUE) {
  g(x + T)
})
#> [1] "language"
#> [1] TRUE
```

Given that closures are converted to `call` objects, it is not a surprise that the function works:

```
logical_abbr(function(x = TRUE) {
  g(x + T)
})
#> [1] TRUE
```

The function only fails if it can't find any negative case. For example, instead of returning `FALSE`, this produces an error for reasons that remain (as of yet) elusive to me:

```
logical_abbr(function(x = TRUE) {
  g(x + TRUE)
})
#> [1] FALSE
```

**Q3.** Modify `find_assign` to also detect assignment using replacement functions, i.e. `names(x) <- y`.

**A3.** Although both simple assignment (`x <- y`) and assignment using replacement functions (`names(x) <- y`) have `<-` operator in their call, in the latter case, `names(x)` will be a call object and not a symbol:

```
expr1 <- expr(names(x) <- y)
as.list(expr1)
#> [[1]]
#> `<-`
#>
#> [[2]]
#> names(x)
#>
#> [[3]]
```

```
#> y
typeof(expr1[[2]])
#> [1] "language"

expr2 <- expr(x <- y)
as.list(expr2)
#> [[1]]
#> `<-`
#>
#> [[2]]
#> x
#>
#> [[3]]
#> y
typeof(expr2[[2]])
#> [1] "symbol"
```

That's how we can detect this kind of assignment by checking if the second element of the expression is a `symbol` or `language` type object.

```
expr_type <- function(x) {
  if (is_syntactic_literal(x)) {
    "constant"
  } else if (is.symbol(x)) {
    "symbol"
  } else if (is.call(x)) {
    "call"
  } else if (is.pairlist(x)) {
    "pairlist"
  } else {
    typeof(x)
  }
}

switch_expr <- function(x, ...) {
  switch(expr_type(x),
    ...,
    stop("Don't know how to handle type ", typeof(x), call. =
     ↪  FALSE)
  )
}

flat_map_chr <- function(.x, .f, ...) {
  purrr::flatten_chr(purrr::map(.x, .f, ...))
}
```

```
extract_symbol <- function(x) {
  if (is_symbol(x[[2]])) {
    as_string(x[[2]])
  } else {
    extract_symbol(as.list(x[[2]]))
  }
}

find_assign_call <- function(x) {
  if (is_call(x, "<-") && is_symbol(x[[2]])) {
    lhs <- as_string(x[[2]])
    children <- as.list(x)[-1]
  } else if (is_call(x, "<-") && is_call(x[[2]])) {
    lhs <- extract_symbol(as.list(x[[2]]))
    children <- as.list(x)[-1]
  } else {
    lhs <- character()
    children <- as.list(x)
  }

  c(lhs, flat_map_chr(children, find_assign_rec))
}

find_assign_rec <- function(x) {
  switch_expr(x,
    # Base cases
    constant = ,
    symbol = character(),

    # Recursive cases
    pairlist = flat_map_chr(x, find_assign_rec),
    call = find_assign_call(x)
  )
}

find_assign <- function(x) find_assign_rec(enexpr(x))
```

Let's try it out:

```
find_assign(names(x))
#> character(0)

find_assign(names(x) <- y)
#> [1] "x"
```

```
find_assign(names(f(x)) <- y)
#> [1] "x"

find_assign(names(x) <- y <- z <- NULL)
#> [1] "x" "y" "z"

find_assign(a <- b <- c <- 1)
#> [1] "a" "b" "c"

find_assign(system.time(x <- print(y <- 5)))
#> [1] "x" "y"
```

**Q4.** Write a function that extracts all calls to a specified function.

**A4.** Here is a function that extracts all calls to a specified function:

```
find_function_call <- function(x, .f) {
  if (is_call(x)) {
    if (is_call(x, .f)) {
      list(x)
    } else {
      purrr::map(as.list(x), ~ find_function_call(.x, .f)) %>%
        purrr::compact() %>%
        unlist(use.names = FALSE)
    }
  }
}

# example-1: with infix operator `:`
find_function_call(expr(mean(1:2)), ":")
#> [[1]]
#> 1:2

find_function_call(expr(sum(mean(1:2))), ":")
#> [[1]]
#> 1:2

find_function_call(expr(list(1:5, 4:6, 3:9)), ":")
#> [[1]]
#> 1:5
#>
#> [[2]]
#> 4:6
#>
#> [[3]]
```

```
#> 3:9

find_function_call(expr(list(1:5, sum(4:6), mean(3:9))), ":")
#> [[1]]
#> 1:5
#>
#> [[2]]
#> 4:6
#>
#> [[3]]
#> 3:9

# example-2: with assignment operator `<-`
find_function_call(expr(names(x)), "<-")
#> NULL

find_function_call(expr(names(x) <- y), "<-")
#> [[1]]
#> names(x) <- y

find_function_call(expr(names(f(x)) <- y), "<-")
#> [[1]]
#> names(f(x)) <- y

find_function_call(expr(names(x) <- y <- z <- NULL), "<-")
#> [[1]]
#> names(x) <- y <- z <- NULL

find_function_call(expr(a <- b <- c <- 1), "<-")
#> [[1]]
#> a <- b <- c <- 1

find_function_call(expr(system.time(x <- print(y <- 5))), "<-")
#> [[1]]
#> x <- print(y <- 5)
```

## 18.5 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
```

```
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  crayon        1.5.3   2024-06-20 [1] RSPM
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  lobstr      * 1.1.2   2022-06-22 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  purrr         1.0.4   2025-02-05 [1] RSPM
#>  rlang       * 1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
```

```
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 19

# Quasiquotation

Attaching the needed libraries:

```
library(rlang)
library(purrr)
library(lobstr)
library(dplyr)
library(ggplot2)
```

## 19.1 Motivation (Exercises 19.2.2)

---

**Q1.** For each function in the following base R code, identify which arguments are quoted and which are evaluated.

```
library(MASS)

mtcars2 <- subset(mtcars, cyl == 4)

with(mtcars2, sum(vs))
sum(mtcars2$am)

rm(mtcars2)
```

**A1.** To identify which arguments are quoted and which are evaluated, we can use the trick mentioned in the book:

> If you're ever unsure about whether an argument is quoted or evaluated, try executing the code outside of the function. If it doesn't work or does something different, then that argument is quoted.

- `library(MASS)`

The `package` argument in `library()` is quoted:

```
library(MASS)

MASS
#> Error: object 'MASS' not found
```

- `subset(mtcars, cyl == 4)`

The argument `x` is evaluated, while the argument `subset` is quoted.

```
mtcars2 <- subset(mtcars, cyl == 4)

invisible(mtcars)

cyl == 4
#> Error: object 'cyl' not found
```

- `with(mtcars2, sum(vs))`

The argument `data` is evaluated, while `expr` argument is quoted.

```
with(mtcars2, sum(vs))
#> [1] 10

invisible(mtcars2)

sum(vs)
#> Error: object 'vs' not found
```

- `sum(mtcars2$am)`

The argument `...` is evaluated.

```
sum(mtcars2$am)
#> [1] 8

mtcars2$am
#>  [1] 1 0 0 1 1 1 0 1 1 1 1
```

- `rm(mtcars2)`

The trick we are using so far won't work here since trying to print `mtcars2` will always fail after `rm()` has made a pass at it.

```
rm(mtcars2)
```

We can instead look at the docs for `...`:

> … the objects to be removed, as names (unquoted) or character strings (quoted).

Thus, this argument is not evaluated, but rather quoted.

---

**Q2.** For each function in the following tidyverse code, identify which arguments are quoted and which are evaluated.

```
library(dplyr)
library(ggplot2)

by_cyl <- mtcars %>%
  group_by(cyl) %>%
  summarise(mean = mean(mpg))

ggplot(by_cyl, aes(cyl, mean)) +
  geom_point()
```

**A2.** As seen in the answer for **Q1.**, `library()` quotes its first argument:

```
library(dplyr)
library(ggplot2)
```

In the following code:

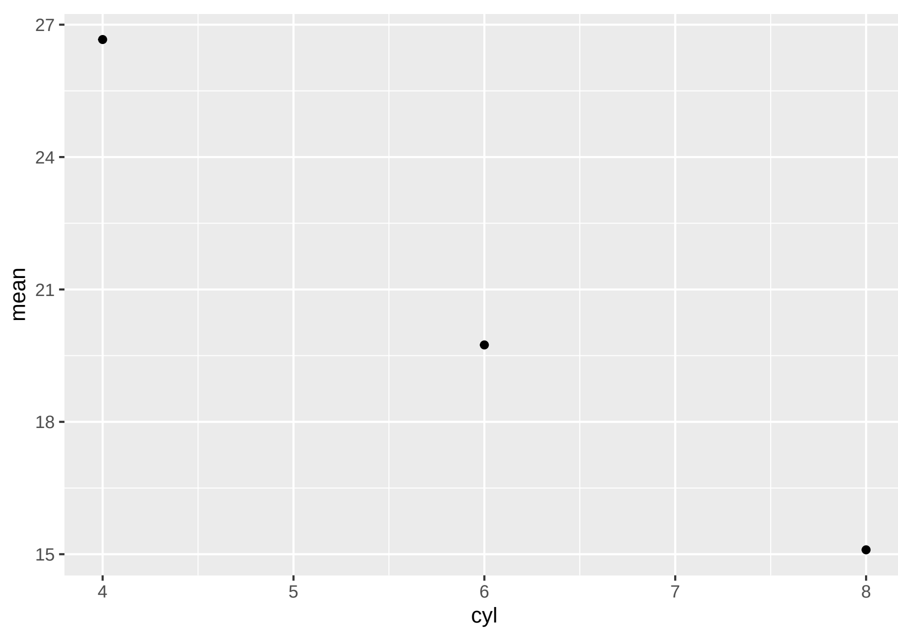
- `%>%` (lazily) evaluates its argument
- `group_by()` and `summarise()` quote their arguments

```
by_cyl <- mtcars %>%
  group_by(cyl) %>%
  summarise(mean = mean(mpg))
```

In the following code:

- `ggplot()` evaluates the `data` argument
- `aes()` quotes its arguments

```
ggplot(by_cyl, aes(cyl, mean)) +
  geom_point()
```




---

## 19.2 Quoting (Exercises 19.3.6)

---

**Q1.** How is `expr()` implemented? Look at its source code.

**A1.** Looking at the source code, we can see that `expr()` is a simple wrapper around `enexpr()`, and captures and returns the user-entered expressions:

```
rlang::expr
#> function (expr)
#> {
#>     enexpr(expr)
#> }
#> <bytecode: 0x5644b8055d58>
#> <environment: namespace:rlang>
```

For example:

```
x <- expr(x <- 1)
x
#> x <- 1
```

In its turn, `enexpr()` calls native code:

```
rlang::enexpr
#> function (arg)
#> {
#>     .Call(ffi_enexpr, substitute(arg), parent.frame())
#> }
#> <bytecode: 0x5644b3d395b8>
#> <environment: namespace:rlang>
```

---

**Q2.** Compare and contrast the following two functions. Can you predict the output before running them?

```
f1 <- function(x, y) {
  exprs(x = x, y = y)
}
f2 <- function(x, y) {
  enexprs(x = x, y = y)
}
f1(a + b, c + d)
f2(a + b, c + d)
```

**A2.** The `exprs()` captures and returns the expressions specified by the developer instead of their values:

```
f1 <- function(x, y) {
  exprs(x = x, y = y)
}

f1(a + b, c + d)
#> $x
#> x
#>
#> $y
#> y
```

On the other hand, `enexprs()` captures the user-entered expressions and returns their values:

```
f2 <- function(x, y) {
  enexprs(x = x, y = y)
}

f2(a + b, c + d)
#> $x
#> a + b
#>
#> $y
#> c + d
```

---

**Q3.** What happens if you try to use `enexpr()` with an expression (i.e. `enexpr(x + y)`? What happens if `enexpr()` is passed a missing argument?

**A3.** If you try to use `enexpr()` with an expression, it fails because it works only with `symbol`.

```
enexpr(x + y)
#> Error in `enexpr()`:
#> ! `arg` must be a symbol
```

If `enexpr()` is passed a missing argument, it returns a missing argument:

```
arg <- missing_arg()

enexpr(arg)

is_missing(enexpr(arg))
#> [1] TRUE
```

---

**Q4.** How are `exprs(a)` and `exprs(a = )` different? Think about both the input and the output.

**A4.** The key difference between `exprs(a)` and `exprs(a = )` is that the former will return an unnamed list, while the latter will return a named list. This is because the former is interpreted as an unnamed argument, while the latter a named argument.

```r
exprs(a)
#> [[1]]
#> a

exprs(a = )
#> $a
```

In both cases, `a` is treated as a symbol:

```r
map_lgl(exprs(a), is_symbol)
#>
#> TRUE

map_lgl(exprs(a = ), is_symbol)
#>    a
#> TRUE
```

But, the argument is missing only in the latter case, since only the name but no corresponding value is provided:

```r
map_lgl(exprs(a), is_missing)
#>
#> FALSE

map_lgl(exprs(a = ), is_missing)
#>    a
#> TRUE
```

---

**Q5.** What are other differences between `exprs()` and `alist()`? Read the documentation for the named arguments of `exprs()` to find out.

**A5.** Here are some additional differences between `exprs()` and `alist()`.

- Names: If the inputs are not named, `exprs()` provides a way to name them automatically using `.named` argument.

```
alist("x" = 1, TRUE, "z" = expr(x + y))
#> $x
#> [1] 1
#>
#> [[2]]
#> [1] TRUE
#>
#> $z
#> expr(x + y)

exprs("x" = 1, TRUE, "z" = expr(x + y), .named = TRUE)
#> $x
#> [1] 1
#>
#> $`TRUE`
#> [1] TRUE
#>
#> $z
#> expr(x + y)
```

- Ignoring empty arguments: The `.ignore_empty` argument in `exprs()` gives you a much finer control over what to do with the empty arguments, while `alist()` doesn't provide a way to ignore such arguments.

```
alist("x" = 1, , TRUE, )
#> $x
#> [1] 1
#>
#> [[2]]
#>
#>
#> [[3]]
#> [1] TRUE
#>
#> [[4]]

exprs("x" = 1, , TRUE, , .ignore_empty = "trailing")
#> $x
#> [1] 1
#>
#> [[2]]
#>
```

```
#>
#> [[3]]
#> [1] TRUE

exprs("x" = 1, , TRUE, , .ignore_empty = "none")
#> $x
#> [1] 1
#>
#> [[2]]
#>
#>
#> [[3]]
#> [1] TRUE
#>
#> [[4]]

exprs("x" = 1, , TRUE, , .ignore_empty = "all")
#> $x
#> [1] 1
#>
#> [[2]]
#> [1] TRUE
```

- Names injection: Using `.unquote_names` argument in `exprs()`, we can inject a name for the argument.

```
alist(foo := bar)
#> [[1]]
#> `:=`(foo, bar)

exprs(foo := bar, .unquote_names = FALSE)
#> [[1]]
#> `:=`(foo, bar)

exprs(foo := bar, .unquote_names = TRUE)
#> $foo
#> bar
```

---

**Q6.** The documentation for `substitute()` says:

> Substitution takes place by examining each component of the parse tree as follows:

- If it is not a bound symbol in `env`, it is unchanged.
- If it is a promise object (i.e., a formal argument to a function) the expression slot of the promise replaces the symbol.
- If it is an ordinary variable, its value is substituted, unless `env` is .GlobalEnv in which case the symbol is left unchanged.

Create examples that illustrate each of the above cases.

**A6.** See below examples that illustrate each of the above-mentioned cases.

> If it is not a bound symbol in `env`, it is unchanged.

Symbol `x` is not bound in `env`, so it remains unchanged.

```
substitute(x + y, env = list(y = 2))
#> x + 2
```

> If it is a promise object (i.e., a formal argument to a function) the expression slot of the promise replaces the symbol.

```
msg <- "old"
delayedAssign("myVar", msg) # creates a promise
substitute(myVar)
#> myVar
msg <- "new!"
myVar
#> [1] "new!"
```

> If it is an ordinary variable, its value is substituted, unless `env` is .GlobalEnv in which case the symbol is left unchanged.

```
substitute(x + y, env = env(x = 2, y = 1))
#> 2 + 1

x <- 2
y <- 1
substitute(x + y, env = .GlobalEnv)
#> x + y
```

---

## 19.3 Unquoting (Exercises 19.4.8)

---

**Q1.** Given the following components:

```
xy <- expr(x + y)
xz <- expr(x + z)
yz <- expr(y + z)
abc <- exprs(a, b, c)
```

Use quasiquotation to construct the following calls:

```
(x + y) / (y + z)
-(x + z)^(y + z)
(x + y) + (y + z) - (x + y)
atan2(x + y, y + z)
sum(x + y, x + y, y + z)
sum(a, b, c)
mean(c(a, b, c), na.rm = TRUE)
foo(a = x + y, b = y + z)
```

**A1.** Using quasiquotation to construct the specified calls:

```
xy <- expr(x + y)
xz <- expr(x + z)
yz <- expr(y + z)
abc <- exprs(a, b, c)

expr((!!xy) / (!!yz))
#> (x + y)/(y + z)

expr(-(!!xz)^(!!yz))
#> -(x + z)^(y + z)

expr(((!!xy)) + (!!yz) - (!!xy))
#> (x + y) + (y + z) - (x + y)

call2("atan2", expr(!!xy), expr(!!yz))
#> atan2(x + y, y + z)

call2("sum", expr(!!xy), expr(!!xy), expr(!!yz))
#> sum(x + y, x + y, y + z)
```

```r
call2("sum", !!!abc)
#> sum(a, b, c)

expr(mean(c(!!!abc), na.rm = TRUE))
#> mean(c(a, b, c), na.rm = TRUE)

call2("foo", a = expr(!!xy), b = expr(!!yz))
#> foo(a = x + y, b = y + z)
```

---

**Q2.** The following two calls print the same, but are actually different:

```r
(a <- expr(mean(1:10)))
#> mean(1:10)
(b <- expr(mean(!!(1:10))))
#> mean(1:10)
identical(a, b)
#> [1] FALSE
```

What's the difference? Which one is more natural?

**A2.** We can see the difference between these two expression if we convert them to lists:

```r
as.list(expr(mean(1:10)))
#> [[1]]
#> mean
#>
#> [[2]]
#> 1:10

as.list(expr(mean(!!(1:10))))
#> [[1]]
#> mean
#>
#> [[2]]
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

As can be seen, the second element of `a` is a `call` object, while that in `b` is an integer vector:

```
waldo::compare(a, b)
#> `old[[2]]` is a call
#> `new[[2]]` is an integer vector (1, 2, 3, 4, 5, ...)
```

The same can also be noticed in ASTs for these expressions:

```
ast(expr(mean(1:10)))
#> expr
#>  mean
#>    `:`
#>     1
#>     10

ast(expr(mean(!!(1:10))))
#> expr
#>  mean
#>   <inline integer>
```

The first call is more natural, since the second one inlines a vector directly into the call, something that is rarely done.

---

## 19.4 . . . (dot-dot-dot) (Exercises 19.6.5)

---

**Q1.** One way to implement `exec()` is shown below. Describe how it works. What are the key ideas?

```
exec <- function(f, ..., .env = caller_env()) {
  args <- list2(...)
  do.call(f, args, envir = .env)
}
```

**A1.** The keys ideas that underlie this implementation of `exec()` function are the following:

- It constructs a call using function `f` and its argument `...`, and evaluates the call in the environment `.env`.

- It uses dynamic dots via `list2()`, which means that you can splice arguments using `!!!`, you can inject names using `:=`, and trailing commas are not a problem.

Here is an example:

```
vec <- c(1:5, NA)
args_list <- list(trim = 0, na.rm = TRUE)

exec(mean, vec, !!!args_list, , .env = caller_env())
#> [1] 3

rm("exec")
```

---

**Q2.** Carefully read the source code for `interaction()`, `expand.grid()`, and `par()`. Compare and contrast the techniques they use for switching between dots and list behaviour.

**A2.** Source code reveals the following comparison table:

| Function | Capture the dots | Handle list input |
|---|---|---|
| `interaction()` | `args <- list(...)` | `length(args) == 1L && is.list(args[[1L]])` |
| `expand.grid()` | `args <- list(...)` | `length(args) == 1L && is.list(args[[1L]])` |
| `par()` | `args <- list(...)` | `length(args) == 1L && (is.list(args[[1L]]) \|\| is.null(args[[1L]]))` |

All functions capture the dots in a list.

Using these dots, the functions check:

- if a list was entered as an argument by checking the number of arguments
- if the count is 1, by checking if the argument is a list

---

**Q3.** Explain the problem with this definition of `set_attr()`

```
set_attr <- function(x, ...) {
  attr <- rlang::list2(...)
  attributes(x) <- attr
  x
}
set_attr(1:10, x = 10)
#> Error in attributes(x) <- attr: attributes must be named
```

**A3.** The `set_attr()` function signature has a parameter called `x`, and additionally it uses dynamic dots to pass multiple arguments to specify additional attributes for `x`.

But, as shown in the example, this creates a problem when the attribute is itself named `x`. Naming the arguments won't help either:

```
set_attr <- function(x, ...) {
  attr <- rlang::list2(...)
  attributes(x) <- attr
  x
}
set_attr(x = 1:10, x = 10)
#> Error in set_attr(x = 1:10, x = 10): formal argument "x"
↪  matched by multiple actual arguments
```

We can avoid these issues by renaming the parameter:

```
set_attr <- function(.x, ...) {
  attr <- rlang::list2(...)
  attributes(.x) <- attr
  .x
}

set_attr(.x = 1:10, x = 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
#> attr(,"x")
#> [1] 10
```

---

## 19.5 Case studies (Exercises 19.7.5)

---

**Q1.** In the linear-model example, we could replace the `expr()` in `reduce(summands, ~ expr(!!.x + !!.y))` with `call2()`: `reduce(summands, call2, "+")`. Compare and contrast the two approaches. Which do you think is easier to read?

**A1.** We can rewrite the `linear()` function from this chapter using `call2()` as follows:

```
linear <- function(var, val) {
  var <- ensym(var)
  coef_name <- map(seq_along(val[-1]), ~ expr((!!var)[[!!.x]]))

  summands <- map2(val[-1], coef_name, ~ expr((!!.x * !!.y)))
  summands <- c(val[[1]], summands)

  reduce(summands, ~ call2("+", .x, .y))
}

linear(x, c(10, 5, -4))
#> 10 + (5 * x[[1L]]) + (-4 * x[[2L]])
```

I personally find the version with `call2()` to be much more readable since the `!!` syntax is a bit esoteric.

---

**Q2.** Re-implement the Box-Cox transform defined below using unquoting and `new_function()`:

```
bc <- function(lambda) {
  if (lambda == 0) {
    function(x) log(x)
  } else {
    function(x) (x^lambda - 1) / lambda
  }
}
```

**A2.** Re-implementation of the Box-Cox transform using unquoting and `new_function()`:

```
bc_new <- function(lambda) {
  lambda <- enexpr(lambda)

  if (!!lambda == 0) {
```

```
    new_function(
      exprs(x = ),
      expr(log(x))
    )
  } else {
    new_function(
      exprs(x = ),
      expr((x^(!!lambda) - 1) / (!!lambda))
    )
  }
}
```

Let's try it out to see if it produces the same output as before:

```
bc(0)(1)
#> [1] 0
bc_new(0)(1)
#> [1] 0

bc(2)(2)
#> [1] 1.5
bc_new(2)(2)
#> [1] 1.5
```

---

**Q3.** Re-implement the simple `compose()` defined below using quasiquotation and `new_function()`:

```
compose <- function(f, g) {
  function(...) f(g(...))
}
```

**A3.** Following is a re-implementation of `compose()` using quasiquotation and `new_function()`:

```
compose_new <- function(f, g) {
  f <- enexpr(f) # or ensym(f)
  g <- enexpr(g) # or ensym(g)

  new_function(
    exprs(... = ),
    expr((!!f)((!!g)(...)))
```

```
  )
}
```

Checking that the new version behaves the same way as the original version:

```
not_null <- compose(`!`, is.null)
not_null(4)
#> [1] TRUE

not_null2 <- compose_new(`!`, is.null)
not_null2(4)
#> [1] TRUE
```

---

## 19.6 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  colorspace    2.1-1   2024-07-26 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  crayon        1.5.3   2024-06-20 [1] RSPM
```

```
#>  datasets     * 4.4.3   2025-02-28 [3] local
#>  diffobj        0.3.5   2021-10-05 [1] RSPM
#>  digest         0.6.37  2024-08-19 [1] RSPM
#>  dplyr        * 1.1.4   2023-11-17 [1] RSPM
#>  emoji          16.0.0  2024-10-28 [1] RSPM
#>  evaluate       1.0.3   2025-01-10 [1] RSPM
#>  farver         2.1.2   2024-05-13 [1] RSPM
#>  fastmap        1.2.0   2024-05-15 [1] RSPM
#>  generics       0.1.3   2022-07-05 [1] RSPM
#>  ggplot2      * 3.5.1   2024-04-23 [1] RSPM
#>  glue           1.8.0   2024-09-30 [1] RSPM
#>  graphics     * 4.4.3   2025-02-28 [3] local
#>  grDevices    * 4.4.3   2025-02-28 [3] local
#>  grid           4.4.3   2025-02-28 [3] local
#>  gtable         0.3.6   2024-10-25 [1] RSPM
#>  htmltools      0.5.8.1 2024-04-04 [1] RSPM
#>  knitr          1.49    2024-11-08 [1] RSPM
#>  labeling       0.4.3   2023-08-29 [1] RSPM
#>  lifecycle      1.0.4   2023-11-07 [1] RSPM
#>  lobstr       * 1.1.2   2022-06-22 [1] RSPM
#>  magrittr     * 2.0.3   2022-03-30 [1] RSPM
#>  MASS         * 7.3-64  2025-01-04 [3] CRAN (R 4.4.3)
#>  methods      * 4.4.3   2025-02-28 [3] local
#>  munsell        0.5.1   2024-04-01 [1] RSPM
#>  pillar         1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig      2.0.3   2019-09-22 [1] RSPM
#>  purrr        * 1.0.4   2025-02-05 [1] RSPM
#>  R6             2.6.1   2025-02-15 [1] RSPM
#>  rlang        * 1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown      2.29    2024-11-04 [1] RSPM
#>  scales         1.3.0   2023-11-28 [1] RSPM
#>  sessioninfo    1.2.3   2025-02-05 [1] RSPM
#>  stats        * 4.4.3   2025-02-28 [3] local
#>  stringi        1.8.4   2024-05-06 [1] RSPM
#>  stringr        1.5.1   2023-11-14 [1] RSPM
#>  tibble         3.2.1   2023-03-20 [1] RSPM
#>  tidyselect     1.2.1   2024-03-11 [1] RSPM
#>  tools          4.4.3   2025-02-28 [3] local
#>  utils        * 4.4.3   2025-02-28 [3] local
#>  vctrs          0.6.5   2023-12-01 [1] RSPM
#>  waldo          0.6.1   2024-11-07 [1] RSPM
#>  withr          3.0.2   2024-10-28 [1] RSPM
#>  xfun           0.51    2025-02-19 [1] RSPM
#>  yaml           2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
```

```
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 20

# Evaluation

Attaching the needed libraries:

```r
library(rlang)
```

## 20.1 Evaluation basics (Exercises 20.2.4)

---

**Q1.** Carefully read the documentation for `source()`. What environment does it use by default? What if you supply `local = TRUE`? How do you provide a custom environment?

**A1.** The parameter `local` for `source()` decides the environment in which the parsed expressions are evaluated.

By default `local = FALSE`, this corresponds to the user's workspace (the global environment, i.e.).

```r
withr::with_tempdir(
  code = {
    f <- tempfile()
    writeLines("rlang::env_print()", f)
    foo <- function() source(f, local = FALSE)
    foo()
  }
)
#> <environment: global>
#> Parent: <environment: package:rlang>
```

```
#> Bindings:
#> * .Random.seed: <int>
#> * foo: <fn>
#> * emojis: <chr>
#> * f: <chr>
```

If `local = TRUE`, then the environment from which `source()` is called will be used.

```
withr::with_tempdir(
  code = {
    f <- tempfile()
    writeLines("rlang::env_print()", f)
    foo <- function() source(f, local = TRUE)
    foo()
  }
)
#> <environment: 0x55a7616fae88>
#> Parent: <environment: global>
```

To specify a custom environment, the `sys.source()` function can be used, which provides an `envir` parameter.

---

**Q2.** Predict the results of the following lines of code:

```
eval(expr(eval(expr(eval(expr(2 + 2))))))
eval(eval(expr(eval(expr(eval(expr(2 + 2)))))))
expr(eval(expr(eval(expr(eval(expr(2 + 2)))))))
```

**A2.** Correctly predicted

```
eval(expr(eval(expr(eval(expr(2 + 2))))))
#> [1] 4

eval(eval(expr(eval(expr(eval(expr(2 + 2)))))))
#> [1] 4

expr(eval(expr(eval(expr(eval(expr(2 + 2)))))))
#> eval(expr(eval(expr(eval(expr(2 + 2))))))
```

---

**Q3.** Fill in the function bodies below to re-implement `get()` using `sym()` and `eval()`, and `assign()` using `sym()`, `expr()`, and `eval()`. Don't worry about the multiple ways of choosing an environment that `get()` and `assign()` support; assume that the user supplies it explicitly.

```
# name is a string
get2 <- function(name, env) {}
assign2 <- function(name, value, env) {}
```

**A3.** Here are the required re-implementations:

- `get()`

```
get2 <- function(name, env = caller_env()) {
  name <- sym(name)
  eval(name, env)
}

x <- 2

get2("x")
#> [1] 2
get("x")
#> [1] 2

y <- 1:4
assign("y[1]", 2)

get2("y[1]")
#> [1] 2
get("y[1]")
#> [1] 2
```

- `assign()`

```
assign2 <- function(name, value, env = caller_env()) {
  name <- sym(name)
  eval(expr(!!name <- !!value), env)
}

assign("y1", 4)
y1
#> [1] 4
```

```r
assign2("y2", 4)
y2
#> [1] 4
```

---

**Q4.** Modify `source2()` so it returns the result of *every* expression, not just the last one. Can you eliminate the for loop?

**A4.** We can use `purrr::map()` to iterate over every expression and return result of every expression:

```r
source2 <- function(path, env = caller_env()) {
  file <- paste(readLines(path, warn = FALSE), collapse = "\n")
  exprs <- parse_exprs(file)
  purrr::map(exprs, ~ eval(.x, env))
}

withr::with_tempdir(
  code = {
    f <- tempfile(fileext = ".R")
    writeLines("1 + 1; 2 + 4", f)
    source2(f)
  }
)
#> [[1]]
#> [1] 2
#>
#> [[2]]
#> [1] 6
```

---

**Q5.** We can make `base::local()` slightly easier to understand by spreading out over multiple lines:

```r
local3 <- function(expr, envir = new.env()) {
  call <- substitute(eval(quote(expr), envir))
  eval(call, envir = parent.frame())
}
```

Explain how `local()` works in words. (Hint: you might want to `print(call)` to help understand what `substitute()` is doing, and read the documentation to remind yourself what environment `new.env()` will inherit from.)

**A5.** In order to figure out how this function works, let's add the suggested `print(call)`:

```
local3 <- function(expr, envir = new.env()) {
  call <- substitute(eval(quote(expr), envir))
  print(call)

  eval(call, envir = parent.frame())
}

local3({
  x <- 10
  y <- 200
  x + y
})
#> eval(quote({
#>     x <- 10
#>     y <- 200
#>     x + y
#> }), new.env())
#> [1] 210
```

As docs for `substitute()` mention:

> Substituting and quoting often cause confusion when the argument is expression(…). The result is a call to the expression constructor function and needs to be evaluated with eval to give the actual expression object.

Thus, to get the actual expression object, quoted expression needs to be evaluated using `eval()`:

```
is_expression(eval(quote({
  x <- 10
  y <- 200
  x + y
}), new.env()))
#> [1] TRUE
```

Finally, the generated `call` is evaluated in the caller environment. So the final function call looks like the following:

```
# outer environment
eval(
```

```
  # inner environment
  eval(quote({
    x <- 10
    y <- 200
    x + y
  }), new.env()),
  envir = parent.frame()
)
```

Note here that the bindings for `x` and `y` are found in the inner environment, while bindings for functions `eval()`, `quote()`, etc. are found in the outer environment.

---

## 20.2 Quosures (Exercises 20.3.6)

---

**Q1.** Predict what each of the following quosures will return if evaluated.

```
q1 <- new_quosure(expr(x), env(x = 1))
q1
#> <quosure>
#> expr: ^x
#> env:  0x55a75df39058
q2 <- new_quosure(expr(x + !!q1), env(x = 10))
q2
#> <quosure>
#> expr: ^x + (^x)
#> env:  0x55a76237e090
q3 <- new_quosure(expr(x + !!q2), env(x = 100))
q3
#> <quosure>
#> expr: ^x + (^x + (^x))
#> env:  0x55a7626665d0
```

**A1.** Correctly predicted

```
q1 <- new_quosure(expr(x), env(x = 1))
eval_tidy(q1)
#> [1] 1
```

```r
q2 <- new_quosure(expr(x + !!q1), env(x = 10))
eval_tidy(q2)
#> [1] 11

q3 <- new_quosure(expr(x + !!q2), env(x = 100))
eval_tidy(q3)
#> [1] 111
```

---

**Q2.** Write an `enenv()` function that captures the environment associated with an argument. (Hint: this should only require two function calls.)

**A2.** We can make use of the `get_env()` helper to get the environment associated with an argument:

```r
enenv <- function(x) {
  x <- enquo(x)
  get_env(x)
}

enenv(x)
#> <environment: R_GlobalEnv>

foo <- function(x) enenv(x)
foo()
#> <environment: 0x55a7633b3048>
```

---

## 20.3 Data masks (Exercises 20.4.6)

---

**Q1.** Why did I use a `for` loop in `transform2()` instead of `map()`? Consider `transform2(df, x = x * 2, x = x * 2)`.

**A1.** To see why `map()` is not appropriate for this function, let's create a version of the function with `map()` and see what happens.

```r
transform2 <- function(.data, ...) {
  dots <- enquos(...)
```

```r
  for (i in seq_along(dots)) {
    name <- names(dots)[[i]]
    dot <- dots[[i]]

    .data[[name]] <- eval_tidy(dot, .data)
  }

  .data
}

transform3 <- function(.data, ...) {
  dots <- enquos(...)

  purrr::map(dots, function(x, .data = .data) {
    name <- names(x)
    dot <- x

    .data[[name]] <- eval_tidy(dot, .data)

    .data
  })
}
```

When we use a `for()` loop, in each iteration, we are updating the `x` column with the current expression under evaluation. That is, repeatedly modifying the same column works.

```r
df <- data.frame(x = 1:3)
transform2(df, x = x * 2, x = x * 2)
#>    x
#> 1  4
#> 2  8
#> 3 12
```

If we use `map()` instead, we are trying to evaluate all expressions at the same time; i.e., the same column is being attempted to modify on using multiple expressions.

```r
df <- data.frame(x = 1:3)
transform3(df, x = x * 2, x = x * 2)
#> Error in `purrr::map()`:
#> i In index: 1.
#> i With name: x.
```

```
#> Caused by error:
#> ! promise already under evaluation: recursive default argument
↪  reference or earlier problems?
```

---

**Q2.** Here's an alternative implementation of `subset2()`:

```
subset3 <- function(data, rows) {
  rows <- enquo(rows)
  eval_tidy(expr(data[!!rows, , drop = FALSE]), data = data)
}
df <- data.frame(x = 1:3)
subset3(df, x == 1)
```

Compare and contrast `subset3()` to `subset2()`. What are its advantages and disadvantages?

**A2.** Let's first juxtapose these functions and their outputs so that we can compare them better.

```
subset2 <- function(data, rows) {
  rows <- enquo(rows)
  rows_val <- eval_tidy(rows, data)
  stopifnot(is.logical(rows_val))

  data[rows_val, , drop = FALSE]
}

df <- data.frame(x = 1:3)
subset2(df, x == 1)
#>   x
#> 1 1
```

```
subset3 <- function(data, rows) {
  rows <- enquo(rows)
  eval_tidy(expr(data[!!rows, , drop = FALSE]), data = data)
}

subset3(df, x == 1)
#>   x
#> 1 1
```

#### Disadvantages of `subset3()` over `subset2()`

When the filtering conditions specified in `rows` don't evaluate to a logical, the function doesn't fail informatively. Indeed, it silently returns incorrect result.

```
rm("x")
exists("x")
#> [1] FALSE

subset2(df, x + 1)
#> Error in subset2(df, x + 1): is.logical(rows_val) is not TRUE

subset3(df, x + 1)
#>      x
#> 2    2
#> 3    3
#> NA  NA
```

#### Advantages of `subset3()` over `subset2()`

Some might argue that the function being shorter is an advantage, but this is very much a subjective preference.

---

**Q3.** The following function implements the basics of `dplyr::arrange()`. Annotate each line with a comment explaining what it does. Can you explain why `!!.na.last` is strictly correct, but omitting the `!!` is unlikely to cause problems?

```
arrange2 <- function(.df, ..., .na.last = TRUE) {
  args <- enquos(...)
  order_call <- expr(order(!!!args, na.last = !!.na.last))
  ord <- eval_tidy(order_call, .df)
  stopifnot(length(ord) == nrow(.df))
  .df[ord, , drop = FALSE]
}
```

**A3.** Annotated version of the function:

```
arrange2 <- function(.df, ..., .na.last = TRUE) {
  # capture user-supplied expressions (and corresponding
  ↪  environments) as quosures
  args <- enquos(...)
```

```r
  # create a call object by splicing a list of quosures
  order_call <- expr(order(!!!args, na.last = !!.na.last))

  # and evaluate the constructed call in the data frame
  ord <- eval_tidy(order_call, .df)

  # sanity check
  stopifnot(length(ord) == nrow(.df))

  .df[ord, , drop = FALSE]
}
```

To see why it doesn't matter whether whether we unquote the `.na.last` argument or not, let's have a look at this smaller example:

```r
x <- TRUE
eval(expr(c(x = !!x)))
#>    x
#> TRUE
eval(expr(c(x = x)))
#>    x
#> TRUE
```

As can be seen:

- without unquoting, `.na.last` is found in the function environment
- with unquoting, `.na.last` is included in the `order` call object itself

---

## 20.4 Using tidy evaluation (Exercises 20.5.4)

---

**Q1.** I've included an alternative implementation of `threshold_var()` below. What makes it different to the approach I used above? What makes it harder?

```r
threshold_var <- function(df, var, val) {
  var <- ensym(var)
  subset2(df, `$`(.data, !!var) >= !!val)
}
```

**A1.** First, let's compare the two definitions for the same function and make sure that they produce the same output:

```
threshold_var_old <- function(df, var, val) {
  var <- as_string(ensym(var))
  subset2(df, .data[[var]] >= !!val)
}

threshold_var_new <- threshold_var

df <- data.frame(x = 1:10)

identical(
  threshold_var(df, x, 8),
  threshold_var(df, x, 8)
)
#> [1] TRUE
```

The key difference is in the subsetting operator used:

- The old version uses non-quoting `[[` operator. Thus, `var` argument first needs to be converted to a string.
- The new version uses quoting `$` operator. Thus, `var` argument is first quoted and then unquoted (using `!!`).

---

## 20.5 Base evaluation (Exercises 20.6.3)

---

**Q1.** Why does this function fail?

```
lm3a <- function(formula, data) {
  formula <- enexpr(formula)
  lm_call <- expr(lm(!!formula, data = data))
  eval(lm_call, caller_env())
}

lm3a(mpg ~ disp, mtcars)$call
#> Error in as.data.frame.default(data, optional = TRUE):
#> cannot coerce class ‘"function"’ to a data.frame
```

**A1.** This doesn't work because when `lm_call` call is evaluated in `caller_env()`, it finds a binding for `base::data()` function, and not `data` from execution environment.

To make it work, we need to unquote `data` into the expression:

```
lm3a <- function(formula, data) {
  formula <- enexpr(formula)
  lm_call <- expr(lm(!!formula, data = !!data))
  eval(lm_call, caller_env())
}

is_call(lm3a(mpg ~ disp, mtcars)$call)
#> [1] TRUE
```

---

**Q2.** When model building, typically the response and data are relatively constant while you rapidly experiment with different predictors. Write a small wrapper that allows you to reduce duplication in the code below.

```
lm(mpg ~ disp, data = mtcars)
lm(mpg ~ I(1 / disp), data = mtcars)
lm(mpg ~ disp * cyl, data = mtcars)
```

**A2.** Here is a small wrapper that allows you to enter only the predictors:

```
lm_custom <- function(data = mtcars, x, y = mpg) {
  x <- enexpr(x)
  y <- enexpr(y)
  data <- enexpr(data)

  lm_call <- expr(lm(formula = !!y ~ !!x, data = !!data))

  eval(lm_call, caller_env())
}

identical(
  lm_custom(x = disp),
  lm(mpg ~ disp, data = mtcars)
)
#> [1] TRUE

identical(
```

```
  lm_custom(x = I(1 / disp)),
  lm(mpg ~ I(1 / disp), data = mtcars)
)
#> [1] TRUE

identical(
  lm_custom(x = disp * cyl),
  lm(mpg ~ disp * cyl, data = mtcars)
)
#> [1] TRUE
```

But the function is flexible enough to also allow changing both the data and the dependent variable:

```
lm_custom(data = iris, x = Sepal.Length, y = Petal.Width)
#>
#> Call:
#> lm(formula = Petal.Width ~ Sepal.Length, data = iris)
#>
#> Coefficients:
#>  (Intercept)  Sepal.Length
#>      -3.2002        0.7529
```

---

**Q3.** Another way to write `resample_lm()` would be to include the resample expression `(data[sample(nrow(data), replace = TRUE), , drop = FALSE])` in the data argument. Implement that approach. What are the advantages? What are the disadvantages?

**A3.** In this variant of `resample_lm()`, we are providing the resampled data as an argument.

```
resample_lm3 <- function(formula,
                         data,
                         resample_data = data[sample(nrow(data),
                          ↪  replace = TRUE), , drop = FALSE],
                         env = current_env()) {
  formula <- enexpr(formula)
  lm_call <- expr(lm(!!formula, data = resample_data))
  expr_print(lm_call)
  eval(lm_call, env)
}
```

```
df <- data.frame(x = 1:10, y = 5 + 3 * (1:10) + round(rnorm(10),
 ↪  2))
resample_lm3(y ~ x, data = df)
#> lm(y ~ x, data = resample_data)
#>
#> Call:
#> lm(formula = y ~ x, data = resample_data)
#>
#> Coefficients:
#> (Intercept)            x
#>       2.654        3.420
```

This makes use of R's lazy evaluation of function arguments. That is, `resample_data` argument will be evaluated only when it is needed in the function.

---

## 20.6 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
 ↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
```

```
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  purrr         1.0.4   2025-02-05 [1] RSPM
#>  rlang       * 1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  withr         3.0.2   2024-10-28 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ------------------------------------------------------------
```

# Chapter 21

# Translation

Needed libraries:

```r
library(rlang)
library(purrr)
```

## 21.1 HTML (Exercises 21.2.6)

---

**Q1.** The escaping rules for `<script>` tags are different because they contain JavaScript, not HTML. Instead of escaping angle brackets or ampersands, you need to escape `</script>` so that the tag isn't closed too early. For example, `script("'</script>'")`, shouldn't generate this:

```html
  <script>'</script>'</script>
```

But

```html
  <script>'<\/script>'</script>
```

Adapt the `escape()` to follow these rules when a new argument `script` is set to `TRUE`.

**A1.** Let's first start with the boilerplate code included in the book:

```
escape <- function(x, ...) UseMethod("escape")

escape.character <- function(x, script = FALSE) {
  if (script) {
    x <- gsub("</script>", "<\\/script>", x, fixed = TRUE)
  } else {
    x <- gsub("&", "&amp;", x)
    x <- gsub("<", "&lt;", x)
    x <- gsub(">", "&gt;", x)
  }

  html(x)
}

escape.advr_html <- function(x, ...) x
```

We will also need to tweak the boilerplate to pass this additional parameter to `escape()`:

```
html <- function(x) structure(x, class = "advr_html")

print.advr_html <- function(x, ...) {
  out <- paste0("<HTML> ", x)
  cat(paste(strwrap(out), collapse = "\n"), "\n", sep = "")
}

dots_partition <- function(...) {
  dots <- list2(...)

  if (is.null(names(dots))) {
    is_named <- rep(FALSE, length(dots))
  } else {
    is_named <- names(dots) != ""
  }

  list(
    named = dots[is_named],
    unnamed = dots[!is_named]
  )
}

tag <- function(tag, script = FALSE) {
  force(script)
  new_function(
    exprs(... = ),
```

```
    expr({
      dots <- dots_partition(...)
      attribs <- html_attributes(dots$named)
      children <- map_chr(.x = dots$unnamed, .f = ~ escape(.x,
 ↪  !!script))

      html(paste0(
        !!paste0("<", tag),
        attribs,
        ">",
        paste(children, collapse = ""),
        !!paste0("</", tag, ">")
      ))
    }),
    caller_env()
  )
}

void_tag <- function(tag) {
  new_function(
    exprs(... = ),
    expr({
      dots <- dots_partition(...)
      if (length(dots$unnamed) > 0) {
        abort(!!paste0("<", tag, "> must not have unnamed
          ↪  arguments"))
      }
      attribs <- html_attributes(dots$named)

      html(paste0(!!paste0("<", tag), attribs, " />"))
    }),
    caller_env()
  )
}

p <- tag("p")
script <- tag("script", script = TRUE)
```

```
script("'</script>'")
#> <HTML> <script>'<\/script>'</script>
```

---

**Q2.** The use of `...` for all functions has some big downsides. There's no input validation and there will be little information in the documentation or autocomplete about how they are used in the function. Create a new function that, when given a named list of tags and their attribute names (like below), creates tag functions with named arguments.

```
list(
  a = c("href"),
  img = c("src", "width", "height")
)
```

All tags should get `class` and `id` attributes.

---

**Q3.** Reason about the following code that calls `with_html()` referencing objects from the environment. Will it work or fail? Why? Run the code to verify your predictions.

```
greeting <- "Hello!"
with_html(p(greeting))
p <- function() "p"
address <- "123 anywhere street"
with_html(p(address))
```

**A3.** To work with this, we first need to copy-paste relevant code from the book:

```
tags <- c(
  "a",
  "abbr",
  "address",
  "article",
  "aside",
  "audio",
  "b",
  "bdi",
  "bdo",
  "blockquote",
  "body",
  "button",
  "canvas",
  "caption",
  "cite",
  "code",
  "colgroup",
  "data",
```

```
  "datalist",
  "dd",
  "del",
  "details",
  "dfn",
  "div",
  "dl",
  "dt",
  "em",
  "eventsource",
  "fieldset",
  "figcaption",
  "figure",
  "footer",
  "form",
  "h1",
  "h2",
  "h3",
  "h4",
  "h5",
  "h6",
  "head",
  "header",
  "hgroup",
  "html",
  "i",
  "iframe",
  "ins",
  "kbd",
  "label",
  "legend",
  "li",
  "mark",
  "map",
  "menu",
  "meter",
  "nav",
  "noscript",
  "object",
  "ol",
  "optgroup",
  "option",
  "output",
  "p",
  "pre",
  "progress",
```

```
  "q",
  "ruby",
  "rp",
  "rt",
  "s",
  "samp",
  "script",
  "section",
  "select",
  "small",
  "span",
  "strong",
  "style",
  "sub",
  "summary",
  "sup",
  "table",
  "tbody",
  "td",
  "textarea",
  "tfoot",
  "th",
  "thead",
  "time",
  "title",
  "tr",
  "u",
  "ul",
  "var",
  "video"
)

void_tags <- c(
  "area",
  "base",
  "br",
  "col",
  "command",
  "embed",
  "hr",
  "img",
  "input",
  "keygen",
  "link",
  "meta",
  "param",
```

```
  "source",
  "track",
  "wbr"
)

html_tags <- c(
  tags %>% set_names() %>% map(tag),
  void_tags %>% set_names() %>% map(void_tag)
)

with_html <- function(code) {
  code <- enquo(code)
  eval_tidy(code, html_tags)
}
```

Note that `with_html()` uses `eval_tidy()`, and therefore `code` argument is evaluated first in the `html_tags` named list, which acts as a data mask, and if no object is found in the data mask, searches in the caller environment.

For this reason, the first example code will work:

```
greeting <- "Hello!"
with_html(p(greeting))
#> <HTML> <p>Hello!</p>
```

The following code, however, is not going to work because there is already `address` element in the data mask, and so `p()` will take a function `address()` as an input, and `escape()` doesn't know how to deal with objects of `function` type:

```
"address" %in% names(html_tags)
#> [1] TRUE

p <- function() "p"
address <- "123 anywhere street"
with_html(p(address))
#> Error in `map_chr()`:
#> i In index: 1.
#> Caused by error in `UseMethod()`:
#> ! no applicable method for 'escape' applied to an object of
↪  class "function"
```

**Q4.** Currently the HTML doesn't look terribly pretty, and it's hard to see the structure. How could you adapt `tag()` to do indenting and formatting? (You may need to do some research into block and inline tags.)

**A4.** Let's first have a look at what it currently looks like:

```
with_html(
  body(
    h1("A heading", id = "first"),
    p("Some text &", b("some bold text.")),
    img(src = "myimg.png", width = 100, height = 100)
  )
)
#> <HTML> <body><h1 id='first'>A heading</h1><p>Some
#> text &amp;<b>some bold text.</b></p><img
#> src='myimg.png' width='100' height='100' /></body>
```

We can improve this to follow the Google HTML/CSS Style Guide.

For this, we need to create a new function to indent the code conditionally:

```
print.advr_html <- function(x, ...) {
  cat(paste("<HTML>", x, sep = "\n"))
}

indent <- function(x) {
  paste0("  ", gsub("\n", "\n  ", x))
}

format_code <- function(children, indent = FALSE) {
  if (indent) {
    paste0("\n", paste0(indent(children), collapse = "\n"), "\n")
  } else {
    paste(children, collapse = "")
  }
}
```

We can then update the `body()` function to use this new helper:

```
html_tags$body <- function(...) {
  dots <- dots_partition(...)
  attribs <- html_attributes(dots$named)
  children <- map_chr(dots$unnamed, escape)

  html(paste0(
    "<body",
```

```
    attribs,
    ">",
    format_code(children, indent = TRUE),
    "</body>"
  ))
}
```

The new formatting looks much better:

```
with_html(
  body(
    h1("A heading", id = "first"),
    p("Some text &", b("some bold text.")),
    img(src = "myimg.png", width = 100, height = 100)
  )
)
#> <HTML>
#> <body>
#>   <h1 id='first'>A heading</h1>
#>   <p>Some text &amp;<b>some bold text.</b></p>
#>   <img src='myimg.png' width='100' height='100' />
#> </body>
```

---

## 21.2 LaTeX (Exercises 21.3.8)

I didn't manage to solve these exercises, and so I'd recommend checking out the solutions in the official solutions manual.

---

**Q1.** Add escaping. The special symbols that should be escaped by adding a backslash in front of them are `\`, `$`, and `%`. Just as with HTML, you'll need to make sure you don't end up double-escaping. So you'll need to create a small S3 class and then use that in function operators. That will also allow you to embed arbitrary LaTeX if needed.

---

**Q2.** Complete the DSL to support all the functions that `plotmath` supports.

---

## 21.3 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  purrr       * 1.0.4   2025-02-05 [1] RSPM
#>  rlang       * 1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
```

```
#>  tools          4.4.3   2025-02-28 [3] local
#>  utils        * 4.4.3   2025-02-28 [3] local
#>  vctrs          0.6.5   2023-12-01 [1] RSPM
#>  xfun           0.51    2025-02-19 [1] RSPM
#>  yaml           2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 22

# Debugging

No exercises.

# Chapter 23

# Measuring performance

Attaching the needed libraries:

```r
library(profvis, warn.conflicts = FALSE)
library(dplyr, warn.conflicts = FALSE)
```

## 23.1 Profiling (Exercises 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.

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

**A1.** Let's source the functions mentioned in exercises.

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

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

```r
profvis(f())
#> Error in parse_rprof_lines(lines, expr_source): No parsing
↪  data available. Maybe your function was too fast?
```

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)
#> {
#>     if (...length()) {
#>         dots <- match.call(expand.dots = FALSE)$...
#>         if (!all(vapply(dots, function(x) is.symbol(x) ||
↪  is.character(x),
#>             NA, USE.NAMES = FALSE)))
#>             stop("... must contain names or character
↪  strings")
#>         list <- .Primitive("c")(list, vapply(dots,
↪  as.character,
#>             ""))
#>     }
#>     .Internal(remove(list, envir, inherits))
#> }
#> <bytecode: 0x557dd7cf5c10>
#> <environment: namespace:base>
```

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

---

## 23.2 Microbenchmarking (Exercises 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.** Let's benchmark first using these two approaches:

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

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

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

t_bench_df <- bench_df %>%
  select(expression, time) %>%
  rowwise() %>%
  mutate(bench_mean = mean(unlist(time))) %>%
  ungroup() %>%
  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" = c(t1_systime_gc["elapsed"],
   ↪  t2_systime_gc["elapsed"]),
  "systime_with_nogc" = c(t1_systime_nogc["elapsed"],
   ↪  t2_systime_nogc["elapsed"])
) %>%
  mutate(
    systime_with_gc = systime_with_gc * 1e6, # in microseconds
    systime_with_nogc = systime_with_nogc * 1e6 # in microseconds
  )
```

Now we can compare results from these alternatives:

```
# note that system time columns report time in microseconds
full_join(t_bench_df, t_systime_df, by = "expression")
#> # A tibble: 2 x 4
#>   expression bench_mean systime_with_gc systime_with_nogc
#>   <bch:expr>   <bch:tm>           <dbl>             <dbl>
#> 1 sqrt(x)       813.9ns           0.585             0.443
#> 2 x^0.5          2.04us           1.95              1.95
```

The comparison reveals that these two approaches yield quite similar results. Slight differences in exact values is possibly due to differences in the precision of timers used internally by these functions.

---

**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
) %>%
  select(expression, median) %>%
  arrange(median)
#> # A tibble: 4 x 2
#>   expression      median
#>   <bch:expr>    <bch:tm>
#> 1 sqrt(x)         3.28us
#> 2 exp(log(x)/2)  12.54us
#> 3 x^0.5           18.3us
#> 4 x^(1/2)        18.43us
```

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

---

## 23.3 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
```

```
#> - Packages ---------------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bench         1.1.4   2025-01-16 [1] RSPM
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  dplyr       * 1.1.4   2023-11-17 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  generics      0.1.3   2022-07-05 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  htmlwidgets   1.6.4   2023-12-06 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  profmem       0.6.0   2020-12-13 [1] RSPM
#>  profvis     * 0.4.0   2024-09-20 [1] RSPM
#>  R6            2.6.1   2025-02-15 [1] RSPM
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tibble        3.2.1   2023-03-20 [1] RSPM
#>  tidyselect    1.2.1   2024-03-11 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utf8          1.2.4   2023-10-22 [1] RSPM
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  withr         3.0.2   2024-10-28 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
```

```
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
#>
#> ----------------------------------------------------------
```

# Chapter 24

# Improving performance

Attaching the needed libraries:

```
library(ggplot2)
library(dplyr)
library(purrr)
```

## 24.1 Exercises 24.3.1

**Q1.** What are faster alternatives to `lm()`? Which are specifically designed to work with larger datasets?

**A1.** Faster alternatives to `lm()` can be found by visiting CRAN Task View: High-Performance and Parallel Computing with R page.

Here are some of the available options:

- `speedglm::speedlm()` (for large datasets)
- `biglm::biglm()` (specifically designed for data too large to fit in memory)
- `RcppEigen::fastLm()` (using the `Eigen` linear algebra library)

High performances can be obtained with these packages especially if R is linked against an optimized BLAS, such as ATLAS. You can check this information using `sessionInfo()`:

```r
sessInfo <- sessionInfo()
sessInfo$matprod
#> [1] "default"
sessInfo$LAPACK
#> [1]
↪  "/usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.26.so"
```

Comparing performance of different alternatives:

```r
library(gapminder)

# having a look at the data
glimpse(gapminder)
#> Rows: 1,704
#> Columns: 6
#> $ country   <fct> "Afghanistan", "Afghanistan", "Afghanist~
#> $ continent <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia~
#> $ year      <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982~
#> $ lifeExp   <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, ~
#> $ pop       <int> 8425333, 9240934, 10267083, 11537966, 13~
#> $ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, ~

bench::mark(
  "lm"       = stats::lm(lifeExp ~ continent * gdpPercap,
   ↪  gapminder),
  "speedglm" = speedglm::speedlm(lifeExp ~ continent * gdpPercap,
   ↪  gapminder),
  "biglm"    = biglm::biglm(lifeExp ~ continent * gdpPercap,
   ↪  gapminder),
  "fastLm"   = RcppEigen::fastLm(lifeExp ~ continent * gdpPercap,
   ↪  gapminder),
  check      = FALSE,
  iterations = 1000
)[1:5]
#> # A tibble: 4 x 5
#>   expression      min   median `itr/sec` mem_alloc
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 lm         836.52us 872.15us     1112.    1.26MB
#> 2 speedglm     1.05ms   1.07ms      913.   70.75MB
#> 3 biglm      766.57us 794.29us     1246.  589.44KB
#> 4 fastLm     939.28us 968.69us     1027.    4.54MB
```

The results might change depending on the size of the dataset, with the performance benefits accruing bigger the dataset.

You will have to experiment with different algorithms and find the one that fits the needs of your dataset the best.

**Q2.** What package implements a version of `match()` that's faster for repeated look ups? How much faster is it?

**A2.** The package (and the respective function) is `fastmatch::fmatch()`[1].

The documentation for this function notes:

> It is slightly faster than the built-in version because it uses more specialized code, but in addition it retains the hash table within the table object such that it can be re-used, dramatically reducing the look-up time especially for large table.

With a small vector, `fmatch()` is only slightly faster, but of the same order of magnitude.

```
library(fastmatch, warn.conflicts = FALSE)

small_vec <- c("a", "b", "x", "m", "n", "y")

length(small_vec)
#> [1] 6

bench::mark(
  "base" = match(c("x", "y"), small_vec),
  "fastmatch" = fmatch(c("x", "y"), small_vec)
)[1:5]
#> # A tibble: 2 x 5
#>   expression      min   median `itr/sec` mem_alloc
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 base         1.14us   1.19us   778434.     2.8KB
#> 2 fastmatch    1.02us   1.08us   886561.    2.66KB
```

But, with a larger vector, `fmatch()` is orders of magnitude faster!

```
large_vec <- c(rep(c("a", "b"), 1e4), "x", rep(c("m", "n"), 1e6),
 ↪   "y")

length(large_vec)
#> [1] 2020002

bench::mark(
```

[1]In addition to Google search, you can also try packagefinder to search for CRAN packages.

```r
  "base" = match(c("x", "y"), large_vec),
  "fastmatch" = fmatch(c("x", "y"), large_vec)
)[1:5]
#> Warning: Some expressions had a GC in every iteration; so
#> filtering is disabled.
#> # A tibble: 2 x 5
#>   expression      min   median `itr/sec` mem_alloc
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 base        24.31ms  26.25ms      28.5    31.4MB
#> 2 fastmatch    1.02us   1.06us  894241.         0B
```

We can also look at the hash table:

```r
fmatch.hash(c("x", "y"), small_vec)
#> [1] "a" "b" "x" "m" "n" "y"
#> attr(,".match.hash")
#> <hash table>
```

Additionally, `{fastmatch}` provides equivalent of the familiar infix operator:

```r
library(fastmatch)

small_vec <- c("a", "b", "x", "m", "n", "y")

c("x", "y") %in% small_vec
#> [1] TRUE TRUE

c("x", "y") %fin% small_vec
#> [1] TRUE TRUE
```

**Q3.** List four functions (not just those in base R) that convert a string into a date time object. What are their strengths and weaknesses?

**A3.** Here are four functions that convert a string into a date time object:

- `base::as.POSIXct()`

```r
base::as.POSIXct("2022-05-05 09:23:22")
#> [1] "2022-05-05 09:23:22 UTC"
```

- `base::as.POSIXlt()`

```r
base::as.POSIXlt("2022-05-05 09:23:22")
#> [1] "2022-05-05 09:23:22 UTC"
```

- `lubridate::ymd_hms()`

```r
lubridate::ymd_hms("2022-05-05-09-23-22")
#> [1] "2022-05-05 09:23:22 UTC"
```

- `fasttime::fastPOSIXct()`

```r
fasttime::fastPOSIXct("2022-05-05 09:23:22")
#> [1] "2022-05-05 09:23:22 UTC"
```

We can also compare their performance:

```r
bench::mark(
  "as.POSIXct" = base::as.POSIXct("2022-05-05 09:23:22"),
  "as.POSIXlt" = base::as.POSIXlt("2022-05-05 09:23:22"),
  "ymd_hms" = lubridate::ymd_hms("2022-05-05-09-23-22"),
  "fastPOSIXct" = fasttime::fastPOSIXct("2022-05-05 09:23:22"),
  check = FALSE,
  iterations = 1000
)
#> # A tibble: 4 x 6
#>   expression       min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>  <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1 as.POSIXct   28.56us  30.15us    32561.        0B     32.6
#> 2 as.POSIXlt   19.85us  20.66us    47148.        0B      0
#> 3 ymd_hms       2.19ms   2.25ms      444.    21.5KB      4.03
#> 4 fastPOSIXct   1.18us   1.25us   746534.        0B      0
```

There are many more packages that implement a way to convert from string to a date time object. For more, see CRAN Task View: Time Series Analysis

**Q4.** Which packages provide the ability to compute a rolling mean?

**A4.** Here are a few packages and respective functions that provide a way to compute a rolling mean:

- `RcppRoll::roll_mean()`
- `data.table::frollmean()`
- `roll::roll_mean()`
- `zoo::rollmean()`

- `slider::slide_dbl()`

**Q5.** What are the alternatives to `optim()`?

**A5.** The `optim()` function provides general-purpose optimization. As noted in its docs:

> General-purpose optimization based on Nelder–Mead, quasi-Newton and conjugate-gradient algorithms. It includes an option for box-constrained optimization and simulated annealing.

There are many alternatives and the exact one you would want to choose would depend on the type of optimization you would like to do.

Most available options can be seen at CRAN Task View: Optimization and Mathematical Programming.

## 24.2 Exercises 24.4.3

**Q1.** What's the difference between `rowSums()` and `.rowSums()`?

**A1.** The documentation for these functions state:

> The versions with an initial dot in the name (.colSums() etc) are 'bare-bones' versions for use in programming: they apply only to numeric (like) matrices and do not name the result.

Looking at the source code,

- `rowSums()` function does a number of checks to validate if the arguments are acceptable

```
rowSums
#> function (x, na.rm = FALSE, dims = 1L)
#> {
#>     if (is.data.frame(x))
#>         x <- as.matrix(x)
#>     if (!is.array(x) || length(dn <- dim(x)) < 2L)
#>         stop("'x' must be an array of at least two
↪  dimensions")
#>     if (dims < 1L || dims > length(dn) - 1L)
#>         stop("invalid 'dims'")
#>     p <- prod(dn[-(id <- seq_len(dims))])
#>     dn <- dn[id]
```

```
#>     z <- if (is.complex(x))
#>         .Internal(rowSums(Re(x), prod(dn), p, na.rm)) + (0+1i)
 ↪  *
#>             .Internal(rowSums(Im(x), prod(dn), p, na.rm))
#>     else .Internal(rowSums(x, prod(dn), p, na.rm))
#>     if (length(dn) > 1L) {
#>         dim(z) <- dn
#>         dimnames(z) <- dimnames(x)[id]
#>     }
#>     else names(z) <- dimnames(x)[[1L]]
#>     z
#> }
#> <bytecode: 0x56402e467fe0>
#> <environment: namespace:base>
```

- `.rowSums()` directly proceeds to computation using an internal code which is built in to the R interpreter

```
.rowSums
#> function (x, m, n, na.rm = FALSE)
#> .Internal(rowSums(x, m, n, na.rm))
#> <bytecode: 0x564031f96708>
#> <environment: namespace:base>
```

But they have comparable performance:

```
x <- cbind(x1 = 3, x2 = c(4:1e4, 2:1e5))

bench::mark(
  "rowSums" = rowSums(x),
  ".rowSums" = .rowSums(x, dim(x)[[1]], dim(x)[[2]])
)[1:5]
#> # A tibble: 2 x 5
#>   expression      min   median `itr/sec` mem_alloc
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 rowSums       820us   1.29ms      838.     859KB
#> 2 .rowSums      818us   1.29ms      814.     859KB
```

**Q2.** Make a faster version of `chisq.test()` that only computes the chi-square test statistic when the input is two numeric vectors with no missing values. You can try simplifying `chisq.test()` or by coding from the mathematical definition.

**A2.** If the function is supposed to accept only two numeric vectors without missing values, then we can make `chisq.test()` do less work by removing code corresponding to the following :

- checks for data frame and matrix inputs
- goodness-of-fit test
- simulating $p$-values
- checking for missing values

This leaves us with a much simpler, bare bones implementation:

```
my_chisq_test <- function(x, y) {
  x <- table(x, y)
  n <- sum(x)

  nr <- as.integer(nrow(x))
  nc <- as.integer(ncol(x))

  sr <- rowSums(x)
  sc <- colSums(x)
  E <- outer(sr, sc, "*") / n
  v <- function(r, c, n) c * r * (n - r) * (n - c) / n^3
  V <- outer(sr, sc, v, n)
  dimnames(E) <- dimnames(x)

  STATISTIC <- sum((abs(x - E))^2 / E)
  PARAMETER <- (nr - 1L) * (nc - 1L)
  PVAL <- pchisq(STATISTIC, PARAMETER, lower.tail = FALSE)

  names(STATISTIC) <- "X-squared"
  names(PARAMETER) <- "df"

  structure(
    list(
      statistic = STATISTIC,
      parameter = PARAMETER,
      p.value = PVAL,
      method = "Pearson's Chi-squared test",
      observed = x,
      expected = E,
      residuals = (x - E) / sqrt(E),
      stdres = (x - E) / sqrt(V)
    ),
    class = "htest"
  )
}
```

And, indeed, this custom function performs slightly better[2] than its base equivalent:

[2]Deliberately choosing a larger dataset to stress test the new function.

```r
m <- c(rep("a", 1000), rep("b", 9000))
n <- c(rep(c("x", "y"), 5000))

bench::mark(
  "base" = chisq.test(m, n)$statistic[[1]],
  "custom" = my_chisq_test(m, n)$statistic[[1]]
)[1:5]
#> # A tibble: 2 x 5
#>   expression      min   median `itr/sec` mem_alloc
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 base          850us    879us     1129.    1.57MB
#> 2 custom        648us    675us     1480.    1.12MB
```

**Q3.** Can you make a faster version of `table()` for the case of an input of two integer vectors with no missing values? Can you use it to speed up your chi-square test?

**A3.** In order to make a leaner version of `table()`, we can take a similar approach and trim the unnecessary input checks in light of our new API of accepting just two vectors without missing values. We can remove the following components from the code:

- extracting data from objects entered in `...` argument
- dealing with missing values
- other input validation checks

In addition to this removal, we can also use `fastmatch::fmatch()` instead of `match()`:

```r
my_table <- function(x, y) {
  x_sorted <- sort(unique(x))
  y_sorted <- sort(unique(y))

  x_length <- length(x_sorted)
  y_length <- length(y_sorted)

  bin <-
    fastmatch::fmatch(x, x_sorted) +
    x_length * fastmatch::fmatch(y, y_sorted) -
    x_length

  y <- tabulate(bin, x_length * y_length)

  y <- array(
    y,
```

```
    dim = c(x_length, y_length),
    dimnames = list(x = x_sorted, y = y_sorted)
  )

  class(y) <- "table"
  y
}
```

The custom function indeed performs slightly better:

```
x <- c(rep("a", 1000), rep("b", 9000))
y <- c(rep(c("x", "y"), 5000))

# `check = FALSE` because the custom function has an additional
 ↪  attribute:
# ".match.hash"
bench::mark(
  "base" = table(x, y),
  "custom" = my_table(x, y),
  check = FALSE
)[1:5]
#> # A tibble: 2 x 5
#>   expression      min   median `itr/sec` mem_alloc
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 base          582us    604us     1652.     960KB
#> 2 custom        326us    334us     2943.     484KB
```

We can also use this function in our custom chi-squared test function and see if the performance improves any further:

```
my_chisq_test2 <- function(x, y) {
  x <- my_table(x, y)
  n <- sum(x)

  nr <- as.integer(nrow(x))
  nc <- as.integer(ncol(x))

  sr <- rowSums(x)
  sc <- colSums(x)
  E <- outer(sr, sc, "*") / n
  v <- function(r, c, n) c * r * (n - r) * (n - c) / n^3
  V <- outer(sr, sc, v, n)
  dimnames(E) <- dimnames(x)
```

```r
  STATISTIC <- sum((abs(x - E))^2 / E)
  PARAMETER <- (nr - 1L) * (nc - 1L)
  PVAL <- pchisq(STATISTIC, PARAMETER, lower.tail = FALSE)

  names(STATISTIC) <- "X-squared"
  names(PARAMETER) <- "df"

  structure(
    list(
      statistic = STATISTIC,
      parameter = PARAMETER,
      p.value = PVAL,
      method = "Pearson's Chi-squared test",
      observed = x,
      expected = E,
      residuals = (x - E) / sqrt(E),
      stdres = (x - E) / sqrt(V)
    ),
    class = "htest"
  )
}
```

And, indeed, this new version of the custom function performs even better than it previously did:

```r
m <- c(rep("a", 1000), rep("b", 9000))
n <- c(rep(c("x", "y"), 5000))

bench::mark(
  "base" = chisq.test(m, n)$statistic[[1]],
  "custom" = my_chisq_test2(m, n)$statistic[[1]]
)[1:5]
#> # A tibble: 2 x 5
#>   expression      min   median `itr/sec` mem_alloc
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 base          849us    882us     1125.    1.28MB
#> 2 custom        387us    398us     2453.  586.98KB
```

## 24.3 Exercises 24.5.1

**Q1.** The density functions, e.g., `dnorm()`, have a common interface. Which arguments are vectorised over? What does `rnorm(10, mean = 10:1)` do?

**A1.** The density function family has the following interface:

```r
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Reading the documentation reveals that the following parameters are vectorized: `x`, `q`, `p`, `mean`, `sd`.

This means that something like the following will work:

```r
rnorm(c(1, 2, 3), mean = c(0, -1, 5))
#> [1] 1.124335 0.930398 3.844935
```

But, for functions that don't have multiple vectorized parameters, it won't. For example,

```r
pnorm(c(1, 2, 3), mean = c(0, -1, 5), log.p = c(FALSE, TRUE,
↪  TRUE))
#> [1] 0.84134475 0.99865010 0.02275013
```

The following function call generates 10 random numbers (since `n = 10`) with 10 different distributions with means supplied by the vector `10:1`.

```r
rnorm(n = 10, mean = 10:1)
#>  [1]  8.2421770  9.3920474  7.1362118  7.5789906  5.2551688
#>  [6]  6.0143714  4.6147891  1.1096247  2.8759129 -0.6756857
```

**Q2.** Compare the speed of `apply(x, 1, sum)` with `rowSums(x)` for varying sizes of `x`.

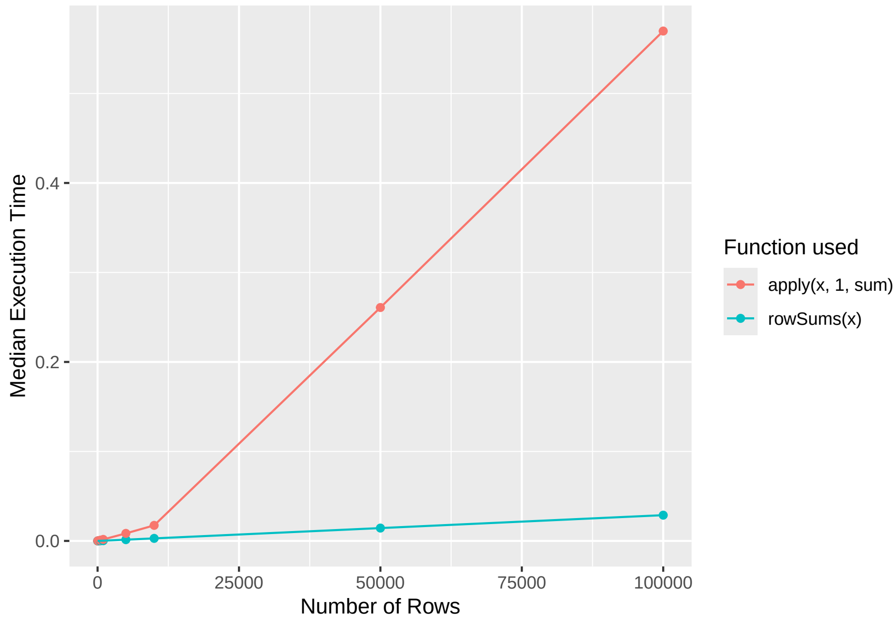
**A2.** We can write a custom function to vary number of rows in a matrix and extract a data frame comparing performance of these two functions.

```r
benc_perform <- function(nRow, nCol = 100) {
  x <- matrix(data = rnorm(nRow * nCol), nrow = nRow, ncol =
↪  nCol)

  bench::mark(
    rowSums(x),
    apply(x, 1, sum)
  )[1:5]
}
```

```
nRowList <- list(10, 100, 500, 1000, 5000, 10000, 50000, 100000)

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

benchDF <- map_dfr(
  .x = nRowList,
  .f = ~ benc_perform(.x),
  .id = "nRows"
) %>%
  mutate(nRows = as.numeric(nRows))
```

Plotting this data reveals that `rowSums(x)` has $O(1)$ behavior, while $O(\mathrm{n})$ behavior.

```
ggplot(
  benchDF,
  aes(
    x = as.numeric(nRows),
    y = median,
    group = as.character(expression),
    color = as.character(expression)
  )
) +
  geom_point() +
  geom_line() +
  labs(
    x = "Number of Rows",
    y = "Median Execution Time",
    colour = "Function used"
  )
```




**Q3.** How can you use `crossprod()` to compute a weighted sum? How much faster is it than the naive `sum(x * w)`?

**A3.** Both of these functions provide a way to compute a weighted sum:

```
x <- c(1:6, 2, 3)
w <- rnorm(length(x))

crossprod(x, w)[[1]]
#> [1] 15.94691
sum(x * w)[[1]]
#> [1] 15.94691
```

But benchmarking their performance reveals that the latter is significantly faster than the former!

```
bench::mark(
  crossprod(x, w)[[1]],
  sum(x * w)[[1]],
  iterations = 1e6
)[1:5]
#> # A tibble: 2 x 5
#>   expression                min   median `itr/sec` mem_alloc
#>   <bch:expr>           <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 crossprod(x, w)[[1]]    400ns    441ns  2115657.        0B
```

```
#> 2 sum(x * w)[[1]]          430ns     481ns  1900443.          0B
```

# Chapter 25

# Rewriting R code in C++

```r
library(Rcpp, warn.conflicts = FALSE)
```

## 25.1 Getting started with C++ (Exercises 25.2.6)

**Q1.** With the basics of C++ in hand, it's now a great time to practice by reading and writing some simple C++ functions. For each of the following functions, read the code and figure out what the corresponding base R function is. You might not understand every part of the code yet, but you should be able to figure out the basics of what the function does.

```cpp
#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;
}
```

**A1.**

`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.** To practice your function writing skills, convert the following functions into C++. For now, assume the inputs have no missing values.

1. `all()`.
2. `cumprod()`, `cummin()`, `cummax()`.
3. `diff()`. Start by assuming lag 1, and then generalise for lag `n`.
4. `range()`.
5. `var()`. Read about the approaches you can take on Wikipedia. Whenever implementing a numerical algorithm, it's always good to check what is already known about the problem.

**A2.** 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)]]
```

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

    return true;
}
```

```r
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)))    6.08us
#> 2 allC(c(rep(TRUE, 1000), rep(FALSE, 1000)))   7.96us
#>     median `itr/sec` mem_alloc `gc/sec`
#>   <bch:tm>     <dbl> <bch:byt>    <dbl>
#> 1   6.48us   118519.    15.8KB        0
#> 2   8.31us   113109.    15.8KB        0
```

- cumprod()

```cpp
#include <vector>
```

```
// [[Rcpp::export]]
std::vector<double> cumprodC(const std::vector<double> &x)
{
    std::vector<double> out{x};

    for (std::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
cumprodC(v1)
#> [1]   10   40  240 1920

# performance benefits?
bench::mark(
  cumprod(v1),
  cumprodC(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 cumprod(v1)    90.1ns    110ns  7421751.        0B
#> 2 cumprodC(v1)  662.1ns    747ns   932536.    4.12KB
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0
```

- cumminC()

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

// [[Rcpp::export]]
std::vector<double> cumminC(const std::vector<double> &x)
{
```

```
    std::vector<double> out{x};

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

    return out;
}
```

```r
v1 <- c(3:1, 2:0, 4:2)

cummin(v1)
#> [1] 3 2 1 1 1 0 0 0 0
cumminC(v1)
#> [1] 3 2 1 1 1 0 0 0 0

# performance benefits?
bench::mark(
  cummin(v1),
  cumminC(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 cummin(v1)     100ns    110ns  6887486.        0B        0
#> 2 cumminC(v1)    741ns    782ns  1075983.    4.12KB        0
```

- cummaxC()

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

// [[Rcpp::export]]
std::vector<double> cummaxC(const std::vector<double> &x)
{
    std::vector<double> out{x};

    for (std::size_t i = 1; i < x.size(); i++)
    {
        out[i] = (out[i] > out[i - 1]) ? out[i] : out[i - 1];
    }

```

```
    return out;
}
```

```r
v1 <- c(3:1, 2:0, 4:2)

cummax(v1)
#> [1] 3 3 3 3 3 3 4 4 4
cummaxC(v1)
#> [1] 3 3 3 3 3 3 4 4 4

# performance benefits?
bench::mark(
  cummax(v1),
  cummaxC(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 cummax(v1)     100ns    110ns  8022878.        0B        0
#> 2 cummaxC(v1)    731ns    792ns  1120300.    4.12KB        0
```

- diff()

```cpp
#include <vector>
#include <functional>
#include <algorithm>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
std::vector<double> diffC(const std::vector<double> &x, int lag)
{
    std::vector<double> vec_start;
    std::vector<double> vec_lagged;
    std::vector<double> vec_diff;

    for (std::size_t i = lag; i < x.size(); i++)
    {
        vec_lagged.push_back(x[i]);
    }

    for (std::size_t i = 0; i < (x.size() - lag); i++)
    {
```

```
        vec_start.push_back(x[i]);
    }

    std::transform(
        vec_lagged.begin(), vec_lagged.end(),
        vec_start.begin(), std::back_inserter(vec_diff),
        std::minus<double>());

    return vec_diff;
}
```

```r
v1 <- c(1, 2, 4, 8, 13)
v2 <- c(1, 2, NA, 8, 13)

diff(v1, 2)
#> [1] 3 6 9
diffC(v1, 2)
#> [1] 3 6 9

diff(v2, 2)
#> [1] NA  6 NA
diffC(v2, 2)
#> [1] NA  6 NA

# performance benefits?
bench::mark(
  diff(v1, 2),
  diffC(v1, 2),
  iterations = 100
)
#> # A tibble: 2 x 6
#>   expression        min   median `itr/sec` mem_alloc
#>   <bch:expr>   <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 diff(v1, 2)    3.81us   4.51us   211525.        0B
#> 2 diffC(v1, 2)   1.06us   1.18us   796142.        0B
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0
```

- 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)     2.36us   2.62us   329893.        0B        0
#> 2 rangeC(v1) 671.02ns 742.03ns  1244934.    4.12KB        0
```

- `var()`

```
#include <vector>
#include <cmath>
#include <numeric>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[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)        5.45us   6.14us   159536.        0B
#> 2 variance(v1) 640.98ns 691.16ns  1336990.    4.12KB
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0
```

## 25.2 Missing values (Exercises 25.4.5)

**Q1.** Rewrite any of the functions from Exercise 25.2.6 to deal with missing values. If `na.rm` is true, ignore the missing values. If `na.rm` is false, return a missing value if the input contains any missing values. Some good functions to practice with are `min()`, `max()`, `range()`, `mean()`, and `var()`.

**A1.** We will only create a version of `range()` that deals with missing values.

The same principle applies to others:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <math.h>
#include <Rcpp.h>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[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 `cumsum()` and `diff()` so they can handle missing values. Note that these functions have slightly more complicated behaviour.

**A2.** The `cumsum()` docs say:

> An `NA` value in `x` causes the corresponding and following elements of the return value to be `NA`, as does integer overflow in cumsum (with a warning).

Similarly, `diff()` docs say:

> `NA`'s propagate.

Therefore, both of these functions don't allow removing missing values and the `NA`s propagate.

As seen from the examples above, `diffC()` already behaves this way.

Similarly, `cumsumC()` propagates `NA`s as well.

```cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
NumericVector cumsumC(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;
}
```

```r
v1 <- c(1, 2, 3, 4)
v2 <- c(1, 2, NA, 4)

cumsum(v1)
#> [1]  1  3  6 10
cumsumC(v1)
#> [1]  1  3  6 10

cumsum(v2)
#> [1]  1  3 NA NA
cumsumC(v2)
#> [1]  1  3 NA NA
```

## 25.3 Standard Template Library (Exercises 25.5.7)

**Q1.** To practice using the STL algorithms and data structures, implement the following using R functions in C++, using the hints provided:

**A1.**

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

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[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)   19.9us   21.6us    44667.        0B
#> 2 medianC(v2)         672.1ns  711.6ns  1286658.        0B
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0
```

1. `%in%` using `unordered_set` and the `find()` or `count()` methods.

```
#include <vector>
#include <unordered_set>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
std::vector<bool> matchC(const std::vector<double> &x, const
 ↪  std::vector<double> &table)
{
    std::unordered_set<double> tableUnique(table.begin(),
     ↪  table.end());
```

```
    std::vector<bool> out;

    for (const auto &xElem : x)
    {
        out.push_back(tableUnique.find(xElem) !=
↪   tableUnique.end() ? true : false);
    }

    return out;
}
```

```
x1 <- c(3, 4, 8)
x2 <- c(1, 2, 3, 3, 4, 4, 5, 6)

x1 %in% x2
#> [1]  TRUE  TRUE FALSE
matchC(x1, x2)
#> [1]  TRUE  TRUE FALSE

# performance benefits?
bench::mark(
  x1 %in% x2,
  matchC(x1, x2),
  iterations = 100
)
#> # A tibble: 2 x 6
#>   expression          min   median `itr/sec` mem_alloc
#>   <bch:expr>     <bch:tm> <bch:tm>     <dbl> <bch:byt>
#> 1 x1 %in% x2     871.14ns    1.1us   843319.        0B
#> 2 matchC(x1, x2)   1.21us   1.31us   696314.    4.12KB
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0
```

1. `unique()` using an `unordered_set` (challenge: do it in one line!).

```
#include <unordered_set>
#include <vector>
#include <iostream>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
```

```
std::unordered_set<double> uniqueC(const std::vector<double> &x)
{
    std::unordered_set<double> xSet(x.begin(), x.end());

    return xSet;
}
```

Note that these functions are **not** comparable. As far as I can see, there is no way to get the same output as the R version of the function using the `unordered_set` data structure.

```
v1 <- c(1, 3, 3, 6, 7, 8, 9)

unique(v1)
#> [1] 1 3 6 7 8 9
uniqueC(v1)
#> [1] 9 8 7 6 3 1
```

We can make comparable version using `set` data structure:

```
#include <set>
#include <vector>
#include <iostream>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
std::set<double> uniqueC2(const std::vector<double> &x)
{
    std::set<double> xSet(x.begin(), x.end());

    return xSet;
}
```

```
v1 <- c(1, 3, 3, 6, 7, 8, 9)

unique(v1)
#> [1] 1 3 6 7 8 9
uniqueC2(v1)
#> [1] 1 3 6 7 8 9

# performance benefits?
bench::mark(
  unique(v1),
```

```
  uniqueC2(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 unique(v1)     2.06us   2.37us   404374.        0B
#> 2 uniqueC2(v1) 852.04ns 962.06ns   952918.    4.12KB
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0
```

1. `min()` using `std::min()`, or `max()` using `std::max()`.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
const double minC(const std::vector<double> &x)
{
     return *std::min_element(x.begin(), x.end());
}

// [[Rcpp::export]]
const double maxC(std::vector<double> x)
{
     return *std::max_element(x.begin(), x.end());
}
```

```
v1 <- c(3, 3, 6, 1, 9, 7, 8)

min(v1)
#> [1] 1
minC(v1)
#> [1] 1

# performance benefits?
bench::mark(
  min(v1),
  minC(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 min(v1)       210ns    231ns  3677702.        0B        0
#> 2 minC(v1)      631ns    691ns  1028268.    4.12KB        0

max(v1)
#> [1] 9
maxC(v1)
#> [1] 9

# performance benefits?
bench::mark(
  max(v1),
  maxC(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 max(v1)       200ns    211ns  4060993.        0B        0
#> 2 maxC(v1)      631ns    671ns  1347267.    4.12KB        0
```

1. `which.min()` using `min_element`, or `which.max()` using `max_element`.

```
#include <vector>
#include <algorithm>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
int which_maxC(std::vector<double> &x)
{
    int maxIndex = std::distance(x.begin(),
     ↪  std::max_element(x.begin(), x.end()));

    // R is 1-index based, while C++ is 0-index based
    return maxIndex + 1;
}

// [[Rcpp::export]]
int which_minC(std::vector<double> &x)
{
```

```
    int minIndex = std::distance(x.begin(),
     ↪  std::min_element(x.begin(), x.end()));

    // R is 1-index based, while C++ is 0-index based
    return minIndex + 1;
}
```

```
v1 <- c(3, 3, 6, 1, 9, 7, 8)

which.min(v1)
#> [1] 4
which_minC(v1)
#> [1] 4

# performance benefits?
bench::mark(
  which.min(v1),
  which_minC(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 which.min(v1)     350ns    381ns  2396297.        0B
#> 2 which_minC(v1)    641ns    732ns  1225750.    4.12KB
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0

which.max(v1)
#> [1] 5
which_maxC(v1)
#> [1] 5

# performance benefits?
bench::mark(
  which.max(v1),
  which_maxC(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 which.max(v1)     360ns    390ns  2362715.        0B
```

```
#> 2 which_maxC(v1)     631ns     661ns  1384593.     4.12KB
#>   `gc/sec`
#>      <dbl>
#> 1        0
#> 2        0
```

1. `setdiff()`, `union()`, and `intersect()` for integers using sorted ranges and `set_union`, `set_intersection` and `set_difference`.

Note that the following C++ implementations of given functions are not strictly equivalent to their R versions. As far as I can see, there is no way for them to be identical while satisfying the specifications mentioned in the question.

- `union()`

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
#include <set>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
std::set<int> unionC(std::vector<int> &v1, std::vector<int> &v2)
{
    std::sort(v1.begin(), v1.end());
    std::sort(v2.begin(), v2.end());

    std::vector<int> union_vec(v1.size() + v2.size());
    auto it = std::set_union(v1.begin(), v1.end(), v2.begin(),
     ↪  v2.end(), union_vec.begin());

    union_vec.resize(it - union_vec.begin());
    std::set<int> union_set(union_vec.begin(), union_vec.end());

    return union_set;
}
```

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

union(v1, v2)
#> [1] 1 4 5 6 2 8
```

```r
unionC(v1, v2)
#> [1] 1 2 4 5 6 8
```

- `intersect()`

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
#include <set>
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
std::set<int> intersectC(std::vector<int> &v1, std::vector<int>
  &v2)
{
    std::sort(v1.begin(), v1.end());
    std::sort(v2.begin(), v2.end());

    std::vector<int> union_vec(v1.size() + v2.size());
    auto it = std::set_intersection(v1.begin(), v1.end(),
      v2.begin(), v2.end(), union_vec.begin());

    union_vec.resize(it - union_vec.begin());
    std::set<int> union_set(union_vec.begin(), union_vec.end());

    return union_set;
}
```

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

intersect(v1, v2)
#> [1] 1 4 6
intersectC(v1, v2)
#> [1] 1 4 6
```

- `setdiff()`

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
#include <set>
```

```
using namespace std;
// [[Rcpp::plugins(cpp11)]]

// [[Rcpp::export]]
std::set<int> setdiffC(std::vector<int> &v1, std::vector<int>
 ↪  &v2)
{
    std::sort(v1.begin(), v1.end());
    std::sort(v2.begin(), v2.end());

    std::vector<int> union_vec(v1.size() + v2.size());
    auto it = std::set_difference(v1.begin(), v1.end(),
     ↪  v2.begin(), v2.end(), union_vec.begin());

    union_vec.resize(it - union_vec.begin());
    std::set<int> union_set(union_vec.begin(), union_vec.end());

    return union_set;
}
```

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

setdiff(v1, v2)
#> [1] 5 2
setdiffC(v1, v2)
#> [1] 2 5
```

## 25.4 Session information

```
sessioninfo::session_info(include_base = TRUE)
#> - Session info -------------------------------------------
#>  setting  value
#>  version  R version 4.4.3 (2025-02-28)
#>  os       Ubuntu 24.04.2 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2025-03-16
```

```
#>  pandoc   3.6.3 @ /opt/hostedtoolcache/pandoc/3.6.3/x64/ (via
↪  rmarkdown)
#>  quarto   NA
#>
#> - Packages -----------------------------------------------
#>  package     * version date (UTC) lib source
#>  base        * 4.4.3   2025-02-28 [3] local
#>  bench         1.1.4   2025-01-16 [1] RSPM
#>  bookdown      0.42    2025-01-07 [1] RSPM
#>  cli           3.6.4   2025-02-13 [1] RSPM
#>  compiler      4.4.3   2025-02-28 [3] local
#>  datasets    * 4.4.3   2025-02-28 [3] local
#>  digest        0.6.37  2024-08-19 [1] RSPM
#>  emoji         16.0.0  2024-10-28 [1] RSPM
#>  evaluate      1.0.3   2025-01-10 [1] RSPM
#>  fastmap       1.2.0   2024-05-15 [1] RSPM
#>  glue          1.8.0   2024-09-30 [1] RSPM
#>  graphics    * 4.4.3   2025-02-28 [3] local
#>  grDevices   * 4.4.3   2025-02-28 [3] local
#>  htmltools     0.5.8.1 2024-04-04 [1] RSPM
#>  knitr         1.49    2024-11-08 [1] RSPM
#>  lifecycle     1.0.4   2023-11-07 [1] RSPM
#>  magrittr    * 2.0.3   2022-03-30 [1] RSPM
#>  methods     * 4.4.3   2025-02-28 [3] local
#>  pillar        1.10.1  2025-01-07 [1] RSPM
#>  pkgconfig     2.0.3   2019-09-22 [1] RSPM
#>  profmem       0.6.0   2020-12-13 [1] RSPM
#>  Rcpp        * 1.0.14  2025-01-12 [1] RSPM
#>  rlang         1.1.5   2025-01-17 [1] RSPM
#>  rmarkdown     2.29    2024-11-04 [1] RSPM
#>  sessioninfo   1.2.3   2025-02-05 [1] RSPM
#>  stats       * 4.4.3   2025-02-28 [3] local
#>  stringi       1.8.4   2024-05-06 [1] RSPM
#>  stringr       1.5.1   2023-11-14 [1] RSPM
#>  tibble        3.2.1   2023-03-20 [1] RSPM
#>  tools         4.4.3   2025-02-28 [3] local
#>  utf8          1.2.4   2023-10-22 [1] RSPM
#>  utils       * 4.4.3   2025-02-28 [3] local
#>  vctrs         0.6.5   2023-12-01 [1] RSPM
#>  xfun          0.51    2025-02-19 [1] RSPM
#>  yaml          2.3.10  2024-07-26 [1] RSPM
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.4.3/lib/R/site-library
#>  [3] /opt/R/4.4.3/lib/R/library
#>  * -- Packages attached to the search path.
```

```
#>
#> -------------------------------------------------------
```