R language: a tour

2025-01-17

M1 MIDS/MFA/LOGOS
Université Paris Cité
Année 2024
Course Homepage
Moodle

> **Objectives**
>
> This workbook intends to walk you through basic aspects of the `R` language and programming environment.

Packages

Base `R` can do a lot. But the full power of `R` comes from a fast growing collection of `packages`.

Packages are first *installed* (that is downloaded from `cran` and copied somewhere on the hard drive), and if needed, *loaded* during a session.

- Installation can usually be performed using command `install.packages()`. In some circumstances, ad hoc installation commands (often from packages `devtools`) are needed
- Package `pak` offers an interesting alternative to base `R` `install.packages()`
- Once a package has been installed/downloaded on your hard drive
  - if you want all objects exported by the package to be available in your session, you should *load* the package, using `library()` or `require()` (what's the difference?). Technically, this loads the `NameSpace` defined by the package.
  - if you just want to pick some objects exported from the package, you can use *qualified names* like `package_name::object_name` to access the object (function, dataset, …).

For example, when we write

```r
gapminder <- gapminder::gapminder
```

we assign dataframe/tibble `gapminder` from package `gapminder` to identifier `"gapminder"` in global environment 😶.

Function `p_load()` from `pacman` (package manager) blends installation and loading: if the package named in the argument of `p_load()` is not installed (not among the `installed.packages()`), `p_load()` attempts to install the package. If installation is successful, the package is loaded.

```r
if (! require(pak)){
  install.packages("pak")
}
```

```r
stopifnot(
  require("tidyverse"),
  require("lobstr"),
  require("ggforce"),
  require("nycflights13"),
  require("patchwork"),
  require("viridis"),
  require("MASS"),
  require("gapminder"),
  require("pryr"),
  require("pak")
)
```

Optional arguments

A very nice feature of `R` is that functions from base `R` as well as from packages have *optional* arguments with sensible *default* values. Look for example at documentation of `require()` using expression `?require`.
Optional settings may concern individual functions or the collection of functions exported by some packages. In the next *chunk*, we reset the default color scales used by graphical functions from `ggplot2`.

```r
opts <- options()  # save old options

options(ggplot2.discrete.colour="viridis")
options(ggplot2.continuous.colour="viridis")
```

! ☛

You shall not confuse *installing* (on your hard-drive) and *loading* (in session) a package.

Question for Pythonistas

- In 🐍 what is the analogue of `install.packages()`?
- In 🐍 what is the analogue of `require()/library()`?

Solution

In , you can install a package `pck` using `pip install pck` or `conda install pck` (for example).
In , the analogue of `require(pck)` could be

```
from pck import *
```

Note that in `R`, once a package in installed on the hard drive, you do not need to write something like

```
import pck
```

to be able to use objects exported by `pck` using qualified names (like `pck.ze_object`), you just need to use `R` qualified names:

```
pck::ze_object
```

Numerical (atomic) vectors

Numerical (atomic) vectors form the most primitive type of `R`.

Vector creation and assignment

The next three lines create three numerical atomic vectors.

In IDE `Rstudio`, have a look at the `environment` pane on the right before running the chunk, and after.

Use `ls()` to investigate the *environment* before and after the execution of the three assignments.

```
1 ls()
2 x <- c(1, 2, 12)
3 y <- 5:7
4 z <- 10:1
5 x ; y ; z
6 ls()
```

Question

- What are the identifiers known in the global environment before execution of lines 2-4?
- What are the identifiers known in the global environment after execution of lines 2-4?
- Which objects are attached to identifiers `x`, `y`, and `z`?

Solution

```
ls()
```

```
[1] "opts"
```

```
x <- c(1, 2, 12)
y <- 5:7
z <- 10:1
x ; y ; z
```

```
[1]  1  2 12
[1] 5 6 7
 [1] 10  9  8  7  6  5  4  3  2  1
```

```
ls()
```

```
[1] "opts" "x"    "y"    "z"
```

The chunks adds three identifiers `x,y,z` to the global environment. Identifiers are bound to `R` objects which turn out to be numerical vectors.

Question

What does the next chunk?

```
ls()
w <- y
ls()
```

Solution

The chunk inserts a new identifier `w` in the global environment. This identifier is associated with the same object as `y`.

Question

- Is the content of object denoted by `y` copied to a new object bound to `w`?
- Interpret the result of `w == y`.
- Interpret the result of `identical(w,y)` (use `help("identical")` if needed).

```
w == y
identical(w,y)
```

Solution

Package `lobstr` lets us explore low-level aspects of `R` (and much more). Function `lobstr::obj_addr()` returns the address of the object denoted by the argument.

```
lobstr::obj_addr(w)
```

```
[1] "0x5ef78c175cb0"
```

```
lobstr::obj_addr(y)
```

```
[1] "0x5ef78c175cb0"
```

Now, if we modify either `y` or `w`

```
y <- y + 1
identical(y, w)
```

```
[1] FALSE
```

```
c(lobstr::obj_addr(w), lobstr::obj_addr(y))
```

```
[1] "0x5ef78c175cb0" "0x5ef786b00278"
```

The address associated with `y` has changed!

⚠ 👉

The meaning of assignment in `R` differs from its counterpart in `Python`. In `Python`, assignment is shallow. In `R`, assignment creates a new identifier bound to the same object as the right-hand side of the assignment. If either side of the assignment is modified, it is copied to a new object before modification. This is called *copy-on-modify*.

Indexation, slicing, modification

Slicing a vector can be done in two ways:

- providing a vector of indices to be selected. Indices need not be consecutive.
- providing a Boolean mask, that is a logical vector to select a set of positions.

```
x <- c(1, 2, 12) ; y <- 5:7 ; z <- 10:1
```

Question

Explain the next lines

```
z[1]    # slice of length 1
z[0]    # What did you expect?
z[x]    # slice of length ??? index error ?
z[y]
z[x %% 2]    # what happens with x[0] ?
z[0 == (x %% 2)] # masking
z[c(2, 1, 1)]
```

Solution

- Indices start at `1` (not like in `C`, `Java`, or `Python`)
- 👉 `z[0]` does not return an Error message. It returns an empty vector with the same basetype as `x`
- `z[x]` returns a vector made of `z[x[1]], z[x[2]]` and `z[x[3]]==z[12]`. Note again that `z[12]` does not raise an exception. It is simply not available (`NA`).
- `x %% 2` returns `1 0 0` as `%%` stands for `mod`. `z[x %% 2]` returns the same thing as `z[1]`

`c( )` stands for combine, or concatenate.

Question

If the length of mask and and the length of the sliced vector do not coincide, what happens?

Solution

No error is signaled, the returned sequence is as long as the number of truthies in the mask.
Out of bound truthies show up as `NA`

```
z[rep(c(TRUE, FALSE), 6)]
```

```
[1] 10  8  6  4  2 NA
```

A scalar is just a vector of length 1!

```
class(z)
```

```
[1] "integer"
```

```
class(z[1])
```

```
[1] "integer"
```

```
class(z[c(2,1)])
```

```
[1] "integer"
```

Question

Explain the next lines

```
y[2:3] <- z[2:3]
y == z[-10]

z[-11]
```

Solution

We can assign a slice of a vector to a slice of identical size of another vector.
What is the result of `z[-11]`, `z[-c(11:7)]`?

Question

Explain the next line

```
z[-(1:5)]
```

Solution

We pick all positions in `z` but the ones in `1:5`, that is 6, 7, 8, 9, 10

Question

How would you select the last element from a vector (say `z`)?

Solution

```
z[length(z)]
```

```
[1] 1
```

R is not Python (reminder)!

Question

Reverse the entries of a vector. Find two ways to do that.

Solution

```
z[seq(length(z), 1, by=-1)]
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
z[length(z):1]
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
rev(z)   # the simplest way, once you know rev()
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

In statistics, machine learning, we are often faced with the task of building grids of regularly spaced elements (these elements can be numeric or not). `R` offers a collection of tools to perform this. The most basic tool is `rep()`.

Question

- Repeat a vector 2 times
- Repeat each element of a vector twice

Solution

```
w <- c(1, 7, 9)
rep(w, 2)
```

```
[1] 1 7 9 1 7 9
```

```
rep(w, rep(2, length(w)))
```

```
[1] 1 1 7 7 9 9
```

Now, we can try something more fancy.

```
rep(w, 1:3)
```

```
[1] 1 7 7 9 9 9
```

What are the requirements on the second (`times`) argument?

Let us remove objects from the global environment.

```
rm(w, x, y ,z)
```

Numbers

So far, we told about numeric vectors. Numeric vectors are vectors of floating point numbers. `R` distinguishes several kinds of numbers.

- Integers
- Floating point numbers (`double`)

To check whether a vector is made of `numeric` or of `integer`, use `is.numeric()` or `is.integer()`. Use `as.integer`, `as.numeric()` to enforce type conversion.

Question

Explain the outcome of the next chunks

```
class(113L) ; class(113) ; class(113L + 113) ; class(2 * 113L) ; class(pi) ; as.intege
```

```
[1] "integer"
[1] "numeric"
[1] "numeric"
[1] "numeric"
[1] "numeric"
[1] 3
```

```
floor(pi) ; class(floor(pi)) # mind the floor
```

```
[1] 3
[1] "numeric"
```

Integer arithmetic

```
29L * 31L ; 899L %/% 32L ; 899L %% 30L
```

```
[1] 899
```

```
[1] 28
```

```
[1] 29
```

R integers are not the natural numbers from Mathematics
R numerics are not the real numbers from Mathematics

```
.Machine$double.eps
```

```
[1] 2.220446e-16
```

```
.Machine$double.xmax
```

```
[1] 1.797693e+308
```

```
.Machine$sizeof.longlong
```

```
[1] 8
```

```
u <- double(19L)
v <- numeric(5L)
w <- integer(7L)
lapply(list(u, v, w), typeof)
```

```
[[1]]
[1] "double"

[[2]]
[1] "double"

[[3]]
[1] "integer"
```

```
length(c(u, v, w))
```

```
[1] 31
```

```
typeof(c(u, v, w))
```

```
[1] "double"
```

R is (sometimes) able to make sensible use of Infinite.

```
log(0)
```

```
[1] -Inf
```

```
log(Inf)
```

```
[1] Inf
```

```
1/0
```

```
[1] Inf
```

```
0/0
```

```
[1] NaN
```

```
max(c( 0/0,1,10))
```

```
[1] NaN
```

```
max(c(NA,1,10))
```

```
[1] NA
```

```
max(c(-Inf,1,10))
```

```
[1] 10
```

```r
is.finite(c(-Inf,1,10))
```

```
[1] FALSE  TRUE  TRUE
```

```r
is.na(c(NA,1,10))
```

```
[1]  TRUE FALSE FALSE
```

```r
is.nan(c(NaN,1,10))
```

```
[1]  TRUE FALSE FALSE
```

Computing with vectors

Summing, scalar multiplication

```r
x <- 1:3
y <- 9:7

sum(x) ; prod(x)
```

```
[1] 6
```

```
[1] 6
```

```r
z <- cumsum(1:3)
w <- cumprod(3:5)

x + y
```

```
[1] 10 10 10
```

```r
x + z
```

```
[1] 2 5 9
```

```r
2 * w
```

```
[1]   6  24 120
```

```r
2 + w
```

```
[1]  5 14 62
```

```r
w / 2
```

```
[1]  1.5  6.0 30.0
```

> **Question**
>
> How would you compute a factorial?

Solution

```r
n <- 10
cumprod(1:n)
```

```
 [1]       1       2       6      24     120     720    5040   40320  362880
[10] 3628800
```

Question

Approximate $\sum_{n=1}^{\infty} 1/n^2$ within 10^{-3}?

Solution

$$\sum_{n>N} \frac{1}{n^2} < \sum_{n>N} \frac{1}{n(n-1)} = \sum_{n>N} \left(\frac{1}{n-1} - \frac{1}{n}\right) = \frac{1}{N}$$

So we may pick $N = 1000$.

```
sum(x*y) # inner product
```

```
[1] 46
```

```
prod(1:5) # factorial(n) as prod(1:n)
```

```
[1] 120
```

```
N <- 1000L
sum(1/((1:N)^2)) ; pi^2/6 # grand truth
```

```
[1] 1.643935
[1] 1.644934
```

```
(pi^2/6 - sum(1/((1:N)^2))) < 1e-3
```

```
[1] TRUE
```

```
# N <- 999L
# (pi^2/6 - sum(1/((1:N)^2))) < 1e-3
```

Question

How would you compute the inner product between two (atomic numeric) vectors?

Solution

Inner product between two vectors can be computed as a matrix product between a row vector and a column vector using `%*%`. Is this a good idea?

```
matrix(w, ncol=3) %*% matrix(y, nrow=3) == sum(w * y)
```

```
     [,1]
[1,] TRUE
```

What we have called `vectors` so far are indeed `atomic vectors`.

- Read Chapter on Vectors in `R advanced Programming`
- Keep an eye on package `vctrs` for getting insights into the `R` vectors.

Numerical matrices

`R` offers a `matrix` class.

```
A <- matrix(1:50, nrow=5)
A
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    6   11   16   21   26   31   36   41    46
```

```
[2,]    2    7   12   17   22   27   32   37   42    47
[3,]    3    8   13   18   23   28   33   38   43    48
[4,]    4    9   14   19   24   29   34   39   44    49
[5,]    5   10   15   20   25   30   35   40   45    50
```

```r
class(A)
```

```
[1] "matrix" "array"
```

Question

From the evaluation of the preceding chunk, can you guess whether it is easier the traverse a matrix in row-first order or in column-first order?

Solution

Default traversal seems to proceed columnwise.

Creation, transposition and reshaping

A vector can be turned into a column matrix.

```r
v <- as.matrix(1:5)
v
```

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

A matrix can be transposed

```r
t(v)   # transpose
```

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

```r
cat(dim(v), ' ', dim(t(v)), '\n')
```

```
5 1   1 5
```

```r
A <- matrix(1, nrow=5, ncol=2) ; A
```

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

Question

`lobstr::mem_used()` allows us to keep track of the amount of memory used by our R session. `lobstr::obj_size()` tells us the amount of memory used by the representation of an object.
Comment the next chunk

```
m1 <-lobstr::mem_used()
A <- matrix(rnorm(100000L), nrow=1000L)
m2 <- lobstr::mem_used()
lobstr::obj_size(A)
```

```
800.22 kB
```

```
B <- t(A)
lobstr::obj_size(B)
```

```
800.22 kB
```

```
m3 <- lobstr::mem_used()
m2-m1 ; m3-m2
```

```
802.44 kB
1.09 MB
```

Question

- Is there a difference between the next two assignments?
- How would you assign value to all entries of a matrix?

```
A <- matrix(rnorm(16), nrow=4)
A[] <- 0 ; A
```

```
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    0    0    0    0
```

```
A   <- 0 ; A
```

```
[1] 0
```

Solution

There is!
The first assignment assigns `0` to every entry in `A`.
The second assignment binds `0` to name `A`

Question

What is the final shape of `A`?

```
A <- matrix(1, nrow=5, ncol=2)
A
A[] <- 1:15
A
```

We can easily generate diagonal matrices and constant matrices.

```r
diag(1, 3)  # building identity matrix
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

```r
matrix(0, 3, 3) # building null matrix
```

```
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
```

> **Question**
>
> Is there any difference between the next two assignments?
>
> ```r
> B <- A[]
> B ; A
> ```
>
> ```
> [1] 0
> [1] 0
> ```
>
> ```r
> lobstr::obj_addr(B) ; lobstr::obj_addr(A)
> ```
>
> ```
> [1] "0x5ef78b53c840"
> [1] "0x5ef78bcc3358"
> ```
>
> ```r
> B <- A
> lobstr::obj_addr(B) ; lobstr::obj_addr(A)
> ```
>
> ```
> [1] "0x5ef78bcc3358"
> [1] "0x5ef78bcc3358"
> ```

Indexation, slicing, modification

Indexation consists in getting one item from a vector/list/matrix/array/dataframe.

Slicing and subsetting consists in picking a substructure:

- subsetting a vector returns a vector
- subsetting a list returns a list
- subsetting a matrix/array returns a matrix/array (beware of implicit simplifications and dimension dropping)
- subsetting a dataframe returns a dataframe or a vector (again, beware of implicit simplifications).

Question

Explain the next results

```r
A <- matrix(1, nrow=5, ncol=2)

dim(A[sample(5, 3), -1])
dim(A[sample(5, 3), 1])
length(A[sample(5, 3), 1])
is.vector(A[sample(5, 3), 1])
A[10:15]
A[60]
dim(A[])
```

```
NULL
NULL
[1] 3
[1] TRUE
[1]  1 NA NA NA NA NA
[1] NA
[1] 5 2
```

Question

How would you create a fresh copy of a matrix?

Solution

```r
A <- matrix(rnorm(10), ncol=2L)
B <- matrix(0, nrow=5L, ncol=2L)

B[] <- A
all(B==A) ; identical(A, B) ; lobstr::obj_addrs(list(A, B))
```

```
[1] TRUE
[1] TRUE
[1] "0x5ef78acf6a78" "0x5ef78acf72b8"
```

Computing with matrices

* **versus %*%** %*% stands for matrix multiplication. In order to use it, the two matrices should have conformant dimensions.

```r
t(v) %*% A
```

```
         [,1]     [,2]
[1,] 3.165927 18.59887
```

There are a variety of reasonable products around. Some of them are available in R.

Question

How would you compute the Hilbert-Schmidt inner product between two matrices?

$$\langle A, B\rangle_{\text{HS}} = \text{Trace}(A \times B^\top)$$

Solution

In R, `trace()` does not return the trace of a matrix! Function is used for debugging. Just remember that the trace of a matrix is the sum of its diagonal elements.

```
A <- matrix(runif(6), 2, 3)
B <- matrix(runif(6), 2, 3)
foo <- sum(diag(A %*% t(B)))
bar <- sum(A * B)
foo ; bar
```

```
[1] 1.154759
[1] 1.154759
```

Are you surprised?

Question

How can you invert a square (invertible) matrix?

Use `solve(A)` which is a shorthand for `solve(A, diag(1, nrow(3)))`.

Logicals

- `R` has constants `TRUE` and `FALSE`.
- Numbers can be coerced to `logicals`.

Question

- Which numbers are truthies? falsies?
- What is the value (if any) of `! pi & TRUE` ?
- What is the meaning of `all( )` ?
- What is the meaning of `any( )` ?
- Recall *De Morgan's laws*. Check them with `R`.
- Is `|` denoting an inclusive or an exclusive OR?

Solution

```r
w <- c(TRUE, FALSE, FALSE)

sum(w)
```

```
[1] 1
```

```r
any(w)
```

```
[1] TRUE
```

```r
all(w)
```

```
[1] FALSE
```

```r
!w
```

```
[1] FALSE  TRUE  TRUE
```

```r
TRUE  & FALSE
```

```
[1] FALSE
```

```r
TRUE | FALSE
```

```
[1] TRUE
```

```r
TRUE | TRUE
```

```
[1] TRUE
```

Handling three-valued logic

Question

```r
TRUE &  (1> (0/0))
(1> (0/0)) | TRUE
(1> (0/0)) | FALSE
TRUE || (1> (0/0))
TRUE |  (1> (0/0))
TRUE || stopifnot(4<3)
# TRUE |  stopifnot(4<3)
FALSE && stopifnot(4<3)
# FALSE & stopifnot(4<3)
```

Solution

```
TRUE &  (1> (0/0))
```

```
[1] NA
```

```
(1> (0/0)) | TRUE
```

```
[1] TRUE
```

```
(1> (0/0)) | FALSE
```

```
[1] NA
```

```
TRUE || (1> (0/0))
```

```
[1] TRUE
```

```
TRUE |  (1> (0/0))
```

```
[1] TRUE
```

```
TRUE || stopifnot(4<3)
```

```
[1] TRUE
```

```
# TRUE |  stopifnot(4<3)
FALSE && stopifnot(4<3)
```

```
[1] FALSE
```

```
# FALSE & stopifnot(4<3)
```

Question

What is the difference between logical operators `||` and `|` ?

Solution

`||` is *lazy*. It does not evaluate its second argument if the first one evaluates to `TRUE`. `&&` is also lazy.

Remark: favor `&`, `|` over `&&`, `||`.

`all` and `any`

Look at the definition of `all` and `any`.

Question

- How would you check that a square matrix is symmetric?
- How would you check that a matrix is diagonal?

Solution

A square matrix is symmetric iff it is equal to its transpose. Recall that `t(A)` denotes the transpose of matrix `A`.

```
A <- matrix(rnorm(9), nrow=3, ncol=3) # a.s. non-symmetric
all(A == t(A))
```

```
[1] FALSE
```

```
A <- A %*% t(A)   # build a symmetric matrix, A + t(A) would work also
all(A == t(A))
```

```
[1] TRUE
```

`A == t(A)` returns a matrix a logical matrix, whose entries are all `TRUE` iff `A` is symmetric.

`all()` works for matrices as well as for vectors. This is sensible as matrices can be considered as vectors with some additional structure.

Lists

While an instance of an atomic `vector` contains objects of the same type/class, an instance of `list` may contain objects of widely different types.

Question

Check an explain the output of the next chunk

```
p <- c(2, 7, 8)
q <- c("A", "B", "C")
x <- list(p, q)
x[2]
x
length(x)
rlang::is_vector(x)
rlang::is_atomic(x)
y <- c(p, q)
y
length(y)
rlang::is_atomic(y)
rlang::is_list(y)
```

```
[[1]]
[1] "A" "B" "C"
```

```
[[1]]
[1] 2 7 8

[[2]]
[1] "A" "B" "C"
```

```
[1] 2
```

```
[1] TRUE
```

```
[1] FALSE
```

```
[1] "2" "7" "8" "A" "B" "C"
```

```
[1] 6
```

```
[1] TRUE
```

```
[1] FALSE
```

`lobstr::tree( )` shows the structure of `x` and of `y`.

```
### Question
#| label: enigma-list-2
#| eval: true
lobstr::tree(x)
```

```
<list>
 <dbl [3]>2, 7, 8
 <chr [3]>"A", "B", "C"
```

```
lobstr::tree(y)
```

```
<chr [6]>"2", "7", "8", "A", "B", "C"
```

Question

- How would you build a list made of `p`, `q`, and `x`?
- What is `x[2]` made of?
- How does it compare with `x[[2]]`?

Solution

```
nl <- list(p=p, q=q, x=x)
nl
```

```
$p
[1] 2 7 8

$q
[1] "A" "B" "C"

$x
$x[[1]]
[1] 2 7 8

$x[[2]]
[1] "A" "B" "C"
```

Note that we have defined a *named* list. Each `=` expression, binds the string on the left-hand side to the object on the right-hand side. List elements can be extracted in defferent ways.

```
names(nl)
```

```
[1] "p" "q" "x"
```

```
nl$q
```

```
[1] "A" "B" "C"
```

```
nl[["q"]]
```

```
[1] "A" "B" "C"
```

```
nl[[2]]
```

```
[1] "A" "B" "C"
```

Question

Read and understand the next expressions.

```
is_atomic(p);  is_atomic(p[2]) ; is_atomic(p[[2]])

is_list(q); is_atomic(q)

is_list(x); is_atomic(x) ; class(x)

class(x[2]) ; class(x[[2]])
length(x[2]) ; length(x[[2]])

identical(q, x[[2]]) ; identical(q, x[2])

obj_addr(q) ; obj_addr(x[[2]]) ; obj_addr(x[2])
ref(x)
obj_addrs(x)
identical(x[2],x[[2]])
```

Functions `is_atomic()`, `is_list()`, `...`, `obj_addr()` are from packages `rlang` and `lobstr`. See https://rlang.r-lib.org and https://lobstr.r-lib.org

Solution

`p` and `a` are atomic vectors with different base types. They are not lists. A list like `x` is not an atomic vector.
Inspection of object addresses shows that when building `x` from objects `p` and `q`, objects bound to `"p"` and `"q"` are not copied.
Note that `x[[2]]` and `x[2]` are different objects, the former is one element list, the second is an atomic vector.

```
lobstr::ref(x[2])
```

```
 [1:0x5ef78b4d3788] <list>
[2:0x5ef783bd9cc8] <chr>
```

```
lobstr::obj_addr(x[[2]])
```

```
[1] "0x5ef783bd9cc8"
```

Question

How would you replace `"A"` in `x` with `"K"`?

Solution

```
x[[2]][1] <- "K"
x
```

```
[[1]]
[1] 2 7 8

[[2]]
[1] "K" "B" "C"
```

```r
lobstr::ref(x)
```

```
 [1:0x5ef78b7a3158] <list>
[2:0x5ef783bd85f8] <dbl>
[3:0x5ef78b90d038] <chr>
```

```r
w <- c(2, 7, 8)
v <- c("A", "B", "C")
x <- list(w, v)
```

! Read Chapter on Lists in `R advanced Programming`

Lookup tables (aka dictionaries) using named vectors

A lookup table maps strings to values. It can be implemented using named vectors. If we want to map: `"seine"` to `"75"`, `"loire"` to `"42"`, `"rhone"` to `"69"`, `"savoie"` to `"73"` we can proceed in the following way:

```r
codes <- c(75L, 42L, 69L, 73L)
names(codes) <- c("seine", "loire", "rhône", "savoie")

codes["rhône"];  codes["aube"]
```

```
rhône
   69
<NA>
  NA
```

Question

What is the class of `codes` ?

Solution

```r
names(codes)
```

```
[1] "seine"  "loire"  "rhône"  "savoie"
```

```r
class(codes); class(names(codes))
```

```
[1] "integer"
[1] "character"
```

```r
is_atomic(codes); is_character(codes) ; is_integer(codes)
```

```
[1] TRUE
[1] FALSE
[1] TRUE
```

Question

Capitalize the `names` used by `codes`

Package `stringr` offers a function `str_to_title()` that could be of interest.

Solution

```
names(codes) <- stringr::str_to_title(names(codes))
codes
```

```
 Seine  Loire  Rhône Savoie 
    75     42     69     73 
```

Factors

Factors exist in Base `R`. They play a very important role. Qualitative/Categorical variables are implemented as Factors.

Meta-package `tidyverse` offers a package dedicated to `factor` engineering: `forcats`.

```
yraw <- c("g1","g1","g2","g2","g2","g3")
print(yraw)
```

```
[1] "g1" "g1" "g2" "g2" "g2" "g3"
```

```
summary(yraw)
```

```
   Length     Class      Mode 
        6 character character 
```

```
is.vector(yraw) ; is.atomic(yraw)
```

```
[1] TRUE
```

```
[1] TRUE
```

Question

`yraw` takes few values. It makes sense to make it a `factor`. How does it change the behavior of *generic* function `summary` ?

Solution

```
fyraw <- as.factor(yraw)
levels(fyraw)
```

```
[1] "g1" "g2" "g3"
```

```
summary(fyraw)
```

```
g1 g2 g3 
 2  3  1 
```

Load the (celebrated) `iris` dataset, and inspect variable `Species`

```
data(iris)

species <- iris$Species

levels(species)
```

```
[1] "setosa"     "versicolor" "virginica" 
```

```r
summary(species)
```

```
    setosa versicolor  virginica 
        50         50         50 
```

Question

We may want to collapse `virginica` and `versicolor` into a single level called `versinica`

`forcats` offer a function `fct_collapse()`.

Solution

```r
col_species <- forcats::fct_collapse(
  species,
  versinica = c("versicolor", "virginica")
)

summary(col_species)
```

```
   setosa versinica 
       50       100 
```

Factors are used to represent *categorical* variables.

Question

- Load the `whiteside` data from package `MASS`.
- Have a glimpse.
- Assign column `Insul` to `y`

Solution

```r
whiteside <- MASS::whiteside  # importing the whiteside data
# ?whiteside                  # what are the whiteside data about?

tibble::glimpse(whiteside)
```

```
Rows: 56
Columns: 3
$ Insul <fct> Before, Before, Before, Before, Before, Before, Before, Before, ~
$ Temp  <dbl> -0.8, -0.7, 0.4, 2.5, 2.9, 3.2, 3.6, 3.9, 4.2, 4.3, 5.4, 6.0, 6.~
$ Gas   <dbl> 7.2, 6.9, 6.4, 6.0, 5.8, 5.8, 5.6, 4.7, 5.8, 5.2, 4.9, 4.9, 4.3,~
```

```r
y <- whiteside$Insul          # picking a factor column
```

Question

- What is the `class` of `y`?
- Is `y` a `vector`
- Is `y` *ordered*? What does *ordered* mean here?
- What are the `levels` of `y` ? How many levels has `y`?
- Can you slice `y` ?
- What are the binary representations of the different levels of `y`?

Solution

```
is.factor(y) ; is.vector(y) ; is.ordered(y)
```

```
[1] TRUE
[1] FALSE
[1] FALSE
```

```
class(y)
```

```
[1] "factor"
```

```
levels(y)
```

```
[1] "Before" "After"
```

```
nlevels(y)
```

```
[1] 2
```

```
y[1:10] # yes we can
```

```
 [1] Before Before Before Before Before Before Before Before Before Before
Levels: Before After
```

```
pryr::bits(y[31]) # looks like the two levels are represented by integers
```

```
[1] "00000000 00000000 00000000 00000010"
```

Question

Summarize factor `y`

Solution

```
summary(y)   # counts
```

```
Before  After 
    26     30 
```

```
table(y)     # one-way contingency table
```

```
y
Before  After 
    26     30 
```

```
table(y)/sum(table(y))*100   # one-way contingency table as percentages
```

```
y
  Before    After 
46.42857 53.57143 
```

```r
table(y) |>
  knitr::kable(col.names = c("Insulation", "Frequency"),
               caption = "Whiteside data")  # Pb encoding sur machine windows
```

Table 2: Whiteside data

Insulation	Frequency
Before	26
After	30

```r
forcats::fct_count(y) |>
  knitr::kable(col.names = c("Insulation", "Frequency"),
               caption = "Whiteside data")
```

Table 3: Whiteside data

Insulation	Frequency
Before	26
After	30

Factors nuts and bolts

When coercing a vector (integer, character, …) to a factor, use `forcats::as_factor()` rather than base R `as.factor()`.

👉 Useful function to make nice **barplots** when constructing **barplots**.
Recall that when you want to display counts for a univariate *categorical* sample, you use a `barplot`. It is often desirable to rank the levels according to the displayed statistics (usually a count).
This can be done in a seamless way using functions like `forcats::fct_infreq()`.

```r
forcats::fct_count(y, prop = TRUE)
```

```
# A tibble: 2 x 3
  f          n     p
  <fct>  <int> <dbl>
1 Before    26 0.464
2 After     30 0.536
```

```r
z <- sample(y, length(y), replace = TRUE)  # permutation of whiteside$Insul

sort(forcats::fct_infreq(z))        # first level is most frequent one
```

```
 [1] Before Before Before Before Before Before Before Before Before Before
[11] Before Before Before Before Before Before Before Before Before Before
[21] Before Before Before Before Before Before Before Before Before Before
[31] Before After  After  After  After  After  After  After  After  After 
[41] After  After  After  After  After  After  After  After  After  After 
[51] After  After  After  After  After  After 
Levels: Before After
```

```
forcats::fct_count(z)
```

```
# A tibble: 2 x 2
  f          n
  <fct>  <int>
1 Before    31
2 After     25
```

Question

Make `z` ordered with level `After` preceding `Before`. Does ordering impact the behavior of `forcats::fct_count()`?

Solution

```
forcats::fct_count(z)
```

```
# A tibble: 2 x 2
  f          n
  <fct>  <int>
1 Before    31
2 After     25
```

```
forcats::fct_count(factor(z, ordered=TRUE, levels=c("After", "Before")))
```

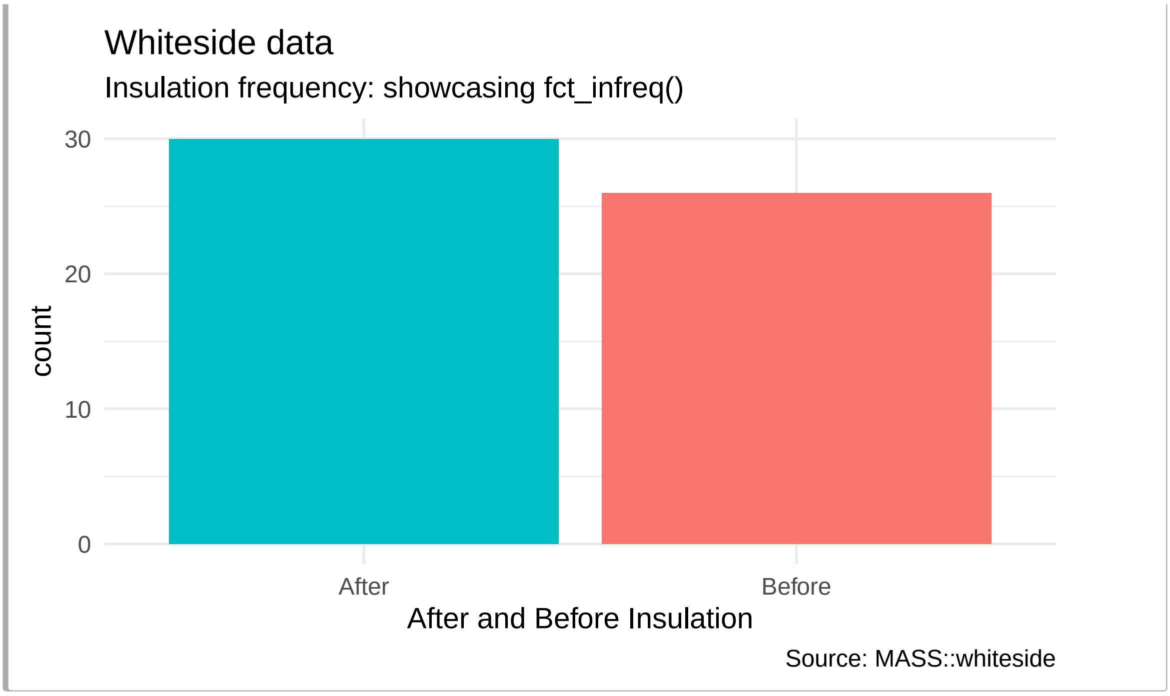
```
# A tibble: 2 x 2
  f          n
  <ord>  <int>
1 After     25
2 Before    31
```

```
forcats::fct_count(forcats::fct_infreq(z))
```

```
# A tibble: 2 x 2
  f          n
  <fct>  <int>
1 Before    31
2 After     25
```

Solution

```
whiteside |>
  ggplot2::ggplot() +
  ggplot2::aes(x=forcats::fct_infreq(Insul), fill=Insul) +
  ggplot2::geom_bar() +
  ggplot2::xlab("After and Before Insulation") +
  ggplot2::theme_minimal() +
  ggplot2::theme(legend.position="None") +
  labs(title="Whiteside data",
    subtitle="Insulation frequency: showcasing fct_infreq()",
    caption="Source: MASS::whiteside")
```



❗ Read Chapter on Factors in `R for Data Science`

Dataframes, `tibbles` and `data.tables`

A dataframe is a list of vectors with equal lengths. This is the way `R` represents and manipulates multivariate samples.

Any software geared at data science supports some kind of dataframe

- `Python Pandas`
- `Python Dask`
- `Spark`
- …

The `iris` dataset is the "Hello world!" of dataframes.

```
data(iris)

iris |>
  glimpse()
```

```
Rows: 150
Columns: 5
$ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4.~
$ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.~
$ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.~
$ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.~
$ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, setosa, s~
```

A `matrix` can be transformed into a `data.frame`

```
A <- matrix(rnorm(10), ncol=2)
data.frame(A)
```

```
          X1          X2
1 -0.3506689 -1.49085270
```

```
2  0.1166999  0.09131630
3  0.4974140 -0.05513997
4  1.2911561 -0.66048268
5 -0.6378476 -0.09422652
```

There are several flavors of dataframes in `R`: `tibble` and `data.table` are modern variants of `data.frame`.

```r
t <- tibble::tibble(
  x=1:3,
  a=letters[11:13],
  d=Sys.Date() + 1:3)

head(t)
```

```
# A tibble: 3 x 3
      x a     d
  <int> <chr> <date>
1     1 k     2025-01-18
2     2 l     2025-01-19
3     3 m     2025-01-20
```

```r
glimpse(t)
```

```
Rows: 3
Columns: 3
$ x <int> 1, 2, 3
$ a <chr> "k", "l", "m"
$ d <date> 2025-01-18, 2025-01-19, 2025-01-20
```

```r
ref(t)
```

```
 [1:0x5ef78b62cc28] <tibble[,3]>
x = [2:0x5ef789e5a678] <int>
a = [3:0x5ef78b6ef308] <chr>
d = [4:0x5ef78b693578] <date>
```

! Read Chapter on data frames and tibbles in Advanced R

Question

Perform a random permutation of the columns of a data.frame/tibble.

Function `sample()` from base `R` is very convenient

Solution

```r
t[sample(names(t))]
```

```
# A tibble: 3 x 3
  d          a         x
  <date>     <chr> <int>
1 2025-01-18 k         1
2 2025-01-19 l         2
3 2025-01-20 m         3
```

```r
# or
t[sample(ncol(t))]
```

```
# A tibble: 3 x 3
  a     d              x
  <chr> <date>     <int>
1 k     2025-01-18     1
2 l     2025-01-19     2
3 m     2025-01-20     3
```

nycflights data

Wrestling with tables is part of the data scientist job. Out of the box data are often messy. In order to perform useful data analysis, we need *tidy* data. The notion of tidy data was elaborated during the last decade by experienced data scientists.

You may benefit from looking at the following online documents.

Tidy data in R for Data Science

Introduction to Table manipulation in R for Data Science in `R`.

More data of that kind is available following guidelines from https://github.com/hadley/nycflights13

In this exercise, you are advised to use functions from dplyr.

> `dplyr` is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges.

```r
data <- nycflights13::flights
```

Question

- Have a glimpse at the data.
- What is the `class` of object `data`?
- What kind of object is `data`?

Hint: use `class()`, `is.data.frame()` `tibble::is_tibble()`

Solution

```r
#| label: flight_glimpse
#| eval: true
data |> glimpse()
```

```
Rows: 336,776
Columns: 19
$ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2~
$ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ~
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ~
$ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -1~
$ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,~
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,~
```

```
$ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -1~
$ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
$ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
$ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
$ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",~
$ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",~
$ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
$ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
$ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6~
$ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
$ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 0~
```

```r
class(data)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```r
is.data.frame(data)
```

```
[1] TRUE
```

```r
is_tibble(data)
```

```
[1] TRUE
```

Question

Extract the name and the type of each column.

Solution

```r
colnames(data)                 # name of columns
```

```
 [1] "year"           "month"          "day"            "dep_time"
 [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
 [9] "arr_delay"      "carrier"        "flight"         "tailnum"
[13] "origin"         "dest"           "air_time"       "distance"
[17] "hour"           "minute"         "time_hour"
```

```r
sapply(data, class)            # old school R, a dataframe is a list
```

```
$year
[1] "integer"

$month
[1] "integer"

$day
[1] "integer"

$dep_time
[1] "integer"

$sched_dep_time
[1] "integer"

$dep_delay
[1] "numeric"
```

```
$arr_time
[1] "integer"

$sched_arr_time
[1] "integer"

$arr_delay
[1] "numeric"

$carrier
[1] "character"

$flight
[1] "integer"

$tailnum
[1] "character"

$origin
[1] "character"

$dest
[1] "character"

$air_time
[1] "numeric"

$distance
[1] "numeric"

$hour
[1] "numeric"

$minute
[1] "numeric"

$time_hour
[1] "POSIXct" "POSIXt"
```

```
lapply(data, class)          # old school R, a dataframe is a list
```

```
$year
[1] "integer"

$month
[1] "integer"

$day
[1] "integer"

$dep_time
[1] "integer"
```

```
$sched_dep_time
[1] "integer"

$dep_delay
[1] "numeric"

$arr_time
[1] "integer"

$sched_arr_time
[1] "integer"

$arr_delay
[1] "numeric"

$carrier
[1] "character"

$flight
[1] "integer"

$tailnum
[1] "character"

$origin
[1] "character"

$dest
[1] "character"

$air_time
[1] "numeric"

$distance
[1] "numeric"

$hour
[1] "numeric"

$minute
[1] "numeric"

$time_hour
[1] "POSIXct" "POSIXt"
```

```r
purrr::map(data, class)            # tidyverse way
```

```
$year
[1] "integer"

$month
[1] "integer"
```

```
$day
[1] "integer"

$dep_time
[1] "integer"

$sched_dep_time
[1] "integer"

$dep_delay
[1] "numeric"

$arr_time
[1] "integer"

$sched_arr_time
[1] "integer"

$arr_delay
[1] "numeric"

$carrier
[1] "character"

$flight
[1] "integer"

$tailnum
[1] "character"

$origin
[1] "character"

$dest
[1] "character"

$air_time
[1] "numeric"

$distance
[1] "numeric"

$hour
[1] "numeric"

$minute
[1] "numeric"

$time_hour
[1] "POSIXct" "POSIXt"
```

Compute the mean of the numerical columns

Base R has plenty of functions that perform statistical computations on univariate samples. Look at the documentation of mean (just type ?mean). For a while, leave aside the optional arguments.

In database parlance, we are performing *aggregation*

```
mean(data$dep_delay)
```

```
[1] NA
```

```
# mean(data[["dep_delay"]])
```

If we want the mean of all numerical columns, we need to project the data frame on numerical columns.

A verb of the summarize family can be useful.

> Have a look at across in latest versions of dplyr()
> Use across() from dplyr 1.x. See Documentation

Solution

```
data |>
  dplyr::select(where(is.numeric)) |>  # projecting on numerical columns
  purrr::map(mean)                      # applying the treatment to each column
```

```
$year
[1] 2013

$month
[1] 6.54851

$day
[1] 15.71079

$dep_time
[1] NA

$sched_dep_time
[1] 1344.255

$dep_delay
[1] NA

$arr_time
[1] NA

$sched_arr_time
[1] 1536.38

$arr_delay
[1] NA
```

```
$flight
[1] 1971.924

$air_time
[1] NA

$distance
[1] 1039.913

$hour
[1] 13.18025

$minute
[1] 26.2301
```

```
data |>
  dplyr::select(where(is.numeric)) |>  # projecting on numerical columns
  dplyr::summarise(across(everything(), mean, na.rm=T))
```

```
# A tibble: 1 x 14
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <dbl> <dbl> <dbl>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
1  2013  6.55  15.7    1349.          1344.      12.6    1502.          1536.
# i 6 more variables: arr_delay <dbl>, flight <dbl>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>
```

```
data |>
  dplyr::summarise(across(where(is.numeric), mean))
```

```
# A tibble: 1 x 14
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <dbl> <dbl> <dbl>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
1  2013  6.55  15.7       NA          1344.        NA       NA          1536.
# i 6 more variables: arr_delay <dbl>, flight <dbl>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>
```

```
data |>
  dplyr::summarise(across(.cols=where(is.numeric), .fns=mean, na.rm=T))
```

```
# A tibble: 1 x 14
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <dbl> <dbl> <dbl>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
1  2013  6.55  15.7    1349.          1344.      12.6    1502.          1536.
# i 6 more variables: arr_delay <dbl>, flight <dbl>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>
```

If applied to a data.frame, `summary()`, produces a summary of each column. The summary depends on the column type. The output of `summary` is a shortened version the list of outputs obtained from applying `summary` to each column (`lapply(data, summary)`).

```
data |>
  summary()
```

```
      year          month             day           dep_time     sched_dep_time
 Min.   :2013   Min.   : 1.000   Min.   : 1.00   Min.   :   1   Min.   : 106
 1st Qu.:2013   1st Qu.: 4.000   1st Qu.: 8.00   1st Qu.: 907   1st Qu.: 906
 Median :2013   Median : 7.000   Median :16.00   Median :1401   Median :1359
```

```
 Mean   :2013   Mean   : 6.549   Mean   :15.71   Mean   :1349   Mean   :1344
 3rd Qu.:2013   3rd Qu.:10.000   3rd Qu.:23.00   3rd Qu.:1744   3rd Qu.:1729
 Max.   :2013   Max.   :12.000   Max.   :31.00   Max.   :2400   Max.   :2359
                                                 NA's   :8255
   dep_delay          arr_time    sched_arr_time   arr_delay
 Min.   : -43.00   Min.   :   1   Min.   :   1   Min.   : -86.000
 1st Qu.:  -5.00   1st Qu.:1104   1st Qu.:1124   1st Qu.: -17.000
 Median :  -2.00   Median :1535   Median :1556   Median :  -5.000
 Mean   :  12.64   Mean   :1502   Mean   :1536   Mean   :   6.895
 3rd Qu.:  11.00   3rd Qu.:1940   3rd Qu.:1945   3rd Qu.:  14.000
 Max.   :1301.00   Max.   :2400   Max.   :2359   Max.   :1272.000
 NA's   :8255      NA's   :8713                  NA's   :9430
   carrier              flight       tailnum             origin
 Length:336776      Min.   :   1   Length:336776      Length:336776
 Class :character   1st Qu.: 553   Class :character   Class :character
 Mode  :character   Median :1496   Mode  :character   Mode  :character
                    Mean   :1972
                    3rd Qu.:3465
                    Max.   :8500

     dest              air_time        distance         hour
 Length:336776      Min.   : 20.0   Min.   :  17   Min.   : 1.00
 Class :character   1st Qu.: 82.0   1st Qu.: 502   1st Qu.: 9.00
 Mode  :character   Median :129.0   Median : 872   Median :13.00
                    Mean   :150.7   Mean   :1040   Mean   :13.18
                    3rd Qu.:192.0   3rd Qu.:1389   3rd Qu.:17.00
                    Max.   :695.0   Max.   :4983   Max.   :23.00
                    NA's   :9430
     minute        time_hour
 Min.   : 0.00   Min.   :2013-01-01 05:00:00.00
 1st Qu.: 8.00   1st Qu.:2013-04-04 13:00:00.00
 Median :29.00   Median :2013-07-03 10:00:00.00
 Mean   :26.23   Mean   :2013-07-03 05:22:54.64
 3rd Qu.:44.00   3rd Qu.:2013-10-01 07:00:00.00
 Max.   :59.00   Max.   :2013-12-31 23:00:00.00
```

Handling NAs

We add now a few NAs to the data….

```r
data2 <- data
data2$arr_time[1:10] <- NA
```

Houston, we have a problem!

Question

How should we compute the column means now?

Solution

```r
data2 |>
  dplyr::summarise(across(is.numeric, mean))
```

```
# A tibble: 1 x 14
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <dbl> <dbl> <dbl>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
1  2013  6.55  15.7       NA          1344.        NA       NA          1536.
# i 6 more variables: arr_delay <dbl>, flight <dbl>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>
```

It is time to look at *optional* arguments of function `mean`.

Question

Decide to ignore `NA` and to compute the mean with the available data

Solution

```r
data2 |>
  dplyr::summarise(across(is.numeric, mean, na.rm=TRUE))
```

```
# A tibble: 1 x 14
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <dbl> <dbl> <dbl>    <dbl>          <dbl>     <dbl>    <dbl>          <dbl>
1  2013  6.55  15.7    1349.          1344.      12.6    1502.          1536.
# i 6 more variables: arr_delay <dbl>, flight <dbl>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>
```

Question

It is possible to remove all rows that contain at least one `NA`.
Show this leads to a different result.

Solution

```r
data2 |>
  drop_na() |>
  dplyr::summarise(across(where(is.numeric), mean, na.rm=FALSE)) |>
  knitr::kable()
```

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	flight	air_time	distance	hour	minute
2013	6.564973	15.74123	1348.813	1340.359	12.55551	1501.929	1532.81	6.895471	1943.125	150.6863	1048.373	13.14126	26.23435

Question

Compute the minimum, the median, the mean and the maximum of numerical columns

Solution

```r
data2 |>
  dplyr::select_if(is.numeric) |>
  lapply(function(x) c(med=median(x, na.rm=TRUE),
                       avg=mean(x, na.rm=TRUE),
                       max=max(x, na.rm=TRUE))
         )
```

```
$year
 med  avg  max
2013 2013 2013

$month
     med      avg      max
 7.00000  6.54851 12.00000

$day
     med      avg      max
16.00000 15.71079 31.00000

$dep_time
    med     avg     max
1401.00 1349.11 2400.00

$sched_dep_time
     med      avg      max
1359.000 1344.255 2359.000

$dep_delay
       med         avg         max
  -2.00000    12.63907 1301.00000

$arr_time
     med      avg      max
1536.000 1502.075 2400.000

$sched_arr_time
    med     avg     max
1556.00 1536.38 2359.00

$arr_delay
        med          avg          max
  -5.000000     6.895377 1272.000000

$flight
     med      avg      max
1496.000 1971.924 8500.000

$air_time
     med      avg      max
129.0000 150.6865 695.0000

$distance
```

```
     med      avg      max
 872.000 1039.913 4983.000

$hour
     med      avg      max
13.00000 13.18025 23.00000

$minute
    med     avg     max
29.0000 26.2301 59.0000
```

```
  data2 |>
  dplyr::summarise(across(where(is.numeric),
                          c(median=median,mean=mean, max=max),
                          na.rm=T))
```

```
# A tibble: 1 x 42
  year_median year_mean year_max month_median month_mean month_max day_median
        <dbl>     <dbl>    <int>        <dbl>      <dbl>     <int>      <dbl>
1        2013      2013     2013            7       6.55        12         16
# i 35 more variables: day_mean <dbl>, day_max <int>, dep_time_median <int>,
#   dep_time_mean <dbl>, dep_time_max <int>, sched_dep_time_median <dbl>,
#   sched_dep_time_mean <dbl>, sched_dep_time_max <int>,
#   dep_delay_median <dbl>, dep_delay_mean <dbl>, dep_delay_max <dbl>,
#   arr_time_median <int>, arr_time_mean <dbl>, arr_time_max <int>,
#   sched_arr_time_median <dbl>, sched_arr_time_mean <dbl>,
#   sched_arr_time_max <int>, arr_delay_median <dbl>, arr_delay_mean <dbl>, ...
```

Question

Obtain a *nicer* output!
Check with https://dplyr.tidyverse.org/reference/scoped.html?q=funs#arguments

Solution

```
data2 |>
  dplyr::summarise(across(where(is.numeric),
                          list(median=median,
                               mean=mean,
                               max=max),
                          na.rm=TRUE)
  ) |>
  tidyr::pivot_longer(cols=everything(), names_to="stat", values_to="value") |>
  head() |>
  gt::gt() |>
  gt::fmt_number(columns="value", decimals=2)
```

stat	value
year_median	2,013.00
year_mean	2,013.00
year_max	2,013.00
month_median	7.00

month_mean	6.55
month_max	12.00

Question

Mimic `summary` on numeric columns

Solution

```
mysum <- data2 |>
  dplyr::summarise(across(is.numeric,
                          list(median=median,
                               mean=mean,
                               max=max,
                               min=min,
                               sd=sd,
                               IQR=IQR) ,
                      na.rm=TRUE))
```

```
mysum |>
  tidyr::pivot_longer(
    cols=everything(),
    names_pattern = "(\\w+)_([a-z]+)",
    names_to= c("variable", "stat"),
    values_to="value") |>
  head(20) |>
  gt::gt() |>
  gt::fmt_number(columns="value", decimals=2)
```

variable	stat	value
year	median	2,013.00
year	mean	2,013.00
year	max	2,013.00
year	min	2,013.00
year	sd	0.00
NA	NA	0.00
month	median	7.00
month	mean	6.55
month	max	12.00
month	min	1.00
month	sd	3.41
NA	NA	6.00
day	median	16.00
day	mean	15.71
day	max	31.00
day	min	1.00
day	sd	8.77
NA	NA	15.00
dep_time	median	1,401.00

dep_time	mean	1,349.11

Question

Compute a new `itinerary` column concatenating the `origin` and `dest` one.
Have a look at Section Operate on a selection of variables

Solution

```
data |>
  dplyr::mutate(itinerary=paste(dest, origin, sep="-")) |>
  dplyr::select(itinerary, dest, origin, everything())
```

```
# A tibble: 336,776 x 20
   itinerary dest  origin  year month   day dep_time sched_dep_time dep_delay
   <chr>     <chr> <chr>  <int> <int> <int>    <int>          <int>     <dbl>
 1 IAH-EWR   IAH   EWR     2013     1     1      517            515         2
 2 IAH-LGA   IAH   LGA     2013     1     1      533            529         4
 3 MIA-JFK   MIA   JFK     2013     1     1      542            540         2
 4 BQN-JFK   BQN   JFK     2013     1     1      544            545        -1
 5 ATL-LGA   ATL   LGA     2013     1     1      554            600        -6
 6 ORD-EWR   ORD   EWR     2013     1     1      554            558        -4
 7 FLL-EWR   FLL   EWR     2013     1     1      555            600        -5
 8 IAD-LGA   IAD   LGA     2013     1     1      557            600        -3
 9 MCO-JFK   MCO   JFK     2013     1     1      557            600        -3
10 ORD-LGA   ORD   LGA     2013     1     1      558            600        -2
# i 336,766 more rows
# i 11 more variables: arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Question

Compute the coefficient of variation (ratio between the standard deviation and the mean) for each itinerary. Can you find several ways?

Solution

```
data |>
  dplyr::mutate(itinerary=paste(dest, origin, sep="-")) |>
  dplyr::select(itinerary, dest, origin, everything()) |>
  dplyr::group_by(itinerary) |>
  dplyr::summarise(coef_var=sd(air_time, na.rm=T)/mean(air_time, na.rm=T), .groups = "dr
  slice_sample(n=10)
```

```
# A tibble: 10 x 2
   itinerary coef_var
   <chr>        <dbl>
 1 RIC-JFK     0.0958
 2 PSP-JFK     0.0479
 3 DAY-LGA     0.0738
 4 MSP-JFK     0.0775
```

```
 5 LAX-JFK      0.0552
 6 TPA-JFK      0.0784
 7 GRR-EWR      0.0889
 8 MIA-LGA      0.0718
 9 DEN-LGA      0.0678
10 GSO-LGA      0.0821
```

Question

Compute for each flight the ratio between the `distance` and the `air_time` in different ways and compare the execution time (use `Sys.time()`).

Solution

```
before <- Sys.time()

data |>
  dplyr::mutate(itinerary=paste(dest, origin, sep="-")) |>
  dplyr::group_by(itinerary) |>
  dplyr::summarize(ratio=mean(air_time)/max(distance)) |>
  dplyr::arrange(desc(ratio))
```

```
# A tibble: 224 x 2
   itinerary ratio
   <chr>     <dbl>
 1 BWI-LGA   0.219
 2 MEM-JFK   0.172
 3 MYR-LGA   0.166
 4 CAE-LGA   0.164
 5 AVL-LGA   0.154
 6 LEX-LGA   0.149
 7 SBN-LGA   0.149
 8 SBN-EWR   0.147
 9 JAC-JFK   0.145
10 STL-JFK   0.145
# i 214 more rows
```

```
required_time <- Sys.time() - before
required_time
```

```
Time difference of 0.1298876 secs
```

Question

Which carrier suffers the most delay?

Solution

```r
data |>
  dplyr::select(carrier, arr_delay) |>
  dplyr::filter(arr_delay > 0) |>
  dplyr::group_by(carrier) |>
  dplyr::summarise(ndelays= n()) |>
#  dplyr::arrange(desc(ndelays)) |>
#  head(3)
  dplyr::top_n(3, ndelays)
```

```
# A tibble: 3 x 2
  carrier ndelays
  <chr>     <int>
1 B6        23609
2 EV        24484
3 UA        22222
```

Puzzle

```r
year <- 2012L

data |>
  dplyr::select(year, dest, origin) |>
  head()
```

```
# A tibble: 6 x 3
   year dest  origin
  <int> <chr> <chr>
1  2013 IAH   EWR
2  2013 IAH   LGA
3  2013 MIA   JFK
4  2013 BQN   JFK
5  2013 ATL   LGA
6  2013 ORD   EWR
```

```r
data |>
  dplyr::filter(year==year) |>
  dplyr::summarize(n())
```

```
# A tibble: 1 x 1
   `n()`
   <int>
1 336776
```

```r
data |>
  dplyr::filter(year==2012L) |>
  dplyr::summarize(n())
```

```
# A tibble: 1 x 1
  `n()`
  <int>
1     0
```

```r
data |>
  dplyr::filter(year==.env$year) |>
  dplyr::summarize(n())
```

```
# A tibble: 1 x 1
  `n()`
  <int>
1     0
```

```
data |>
  dplyr::filter(year==.data$year) |>
  dplyr::summarize(n())
```

```
# A tibble: 1 x 1
   `n()`
   <int>
1 336776
```

Question

Can you explain what happens?

Solution

When `dplyr::filter(year==year)` does `year` refer to the column of `data` or to the variable in the global environment?

Flow control

`R` offers the usual flow control constructs:

- branching/alternative `if (...) {...} else {...}`
- iterations (while/for) `while (...) {...}` `for (it in iterable) {...}`
- function calling `callable(...)` (how do we pass arguments? how do we rely on defaults?)

`If () then {} else {}`

If expressions `yes_expr` and `no_expr` are complicated it makes sense to use the `if (...) {...} else {...}` construct

There is also a conditional statement with an optional `else {}`

```
#| label: if-else
#| eval: false
#| collapse: false
if (condition) {
  ...
} else {
  ...
}
```

Question

Is there an `elif` construct in `R`?

Nope!

👉 `R` also offers a `switch`

```r
#| label: switch
switch (object,
  case1 = {action1},
  case2 = {action2},
  ...
)
```

There exists a selection function `ifelse(test, yes_expr, no_expr)`.

```r
ifelse(test, yes, no)
```

Note that `ifelse(...)` is vectorized.

```r
x <-  1L:6L
y <-  rep("odd", 6)
z <- rep("even", 6)

ifelse(x %% 2L, y, z)
```

```
[1] "odd"  "even" "odd"  "even" "odd"  "even"
```

This is a vectorized function

Iterations `for (it in iterable) {...}`

Have a look at Iteration section in R for Data Science

Question

Create a lower triangular matrix which represents the 5 first lines of the Pascal triangle.

Recall

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Solution

```r
T <- matrix(0L, nrow=6, ncol=6)
T[1,1] <- 1L

for (i in 2:ncol(T))
  T[i, 1:i] <- c(0L, T[i-1, 2:i-1]) + T[i-1, 1:i]

colnames(T) <- 0L:5L
rownames(T) <- 0L:5L

T
```

```
  0 1  2  3 4 5
0 1 0  0  0 0 0
1 1 1  0  0 0 0
2 1 2  1  0 0 0
3 1 3  3  1 0 0
4 1 4  6  4 1 0
```

```
5 1 5 10 10 5 1
```

Question

Locate the smallest element in a numerical vector

Solution

```r
v <- sample(1:100, 100)
v[1:10]
```

```
 [1] 13 90 38 67 54 63 66 88 28 82
```

```r
pmin <- 1

# q: what is the purpose of the following loop?
# for (i in 2:length(v)) {
#   if (v[i]<v[pmin]) {
#     pmin <- i
#   }
# }
for (i in seq_along(v)) {
  if (v[i]<v[pmin]) {
    pmin <- i
  }
}

print(stringr::str_c('minimum is at ', pmin, ', it is equal to ', v[pmin]))
```

```
[1] "minimum is at 100, it is equal to 1"
```

There are some redundant braces `{}`

While (condition) {...}

Question

Find the location of the minimum in a vector `v`

Solution

```r
v <- sample(100, 100)

pmin <- 1   # Minimum in v[1:1]
i <- 2

# q: find le location of the minimum in  vector v

while (i <= length(v)) {
  # loop invariant: v[pmin] == min(v[1:i])
  if (v[i]<v[pmin]) {
    pmin <- i
  }
  i <- i + 1
}

print(stringr::str_c('minimum is at ', pmin, ', it is equal to ', v[pmin]))
```

```
[1] "minimum is at 37, it is equal to 1"
```

```r
which.min(v); v[which.min(v)]
```

```
[1] 37
[1] 1
```

Question

Write a loop that checks whether vector `v` is non-decreasing.

Solution

```r
result <- TRUE

for (i in 2:length(v))
  if (v[i] < v[i-1]) {
    result <- FALSE
    break
  }

if (result) {
  print("non-decreasing")
} else {
  print("not non-decreasing")
}
```

```
[1] "not non-decreasing"
```

Functions

To define a function, whether named or not, you can use the `function` constructor.

```r
foo <- function(arg1, arg2=def2) {
  # body

}
```

Question

Write a function that checks whether vector v is non-decreasing.

Solution

```
is_non_decreasing <- function(v) {
  for (i in 2:length(v))
    if (v[i] < v[i-1]) {
      return(FALSE)
    }
  return(TRUE)
}

is_non_decreasing(v)
```

```
[1] FALSE
```

```
is_non_decreasing(1:10)
```

```
[1] TRUE
```

A function is an object like any other

```
is_non_decreasing
```

```
function(v) {
  for (i in 2:length(v))
    if (v[i] < v[i-1]) {
      return(FALSE)
    }
  return(TRUE)
}
<bytecode: 0x5ef78ca52c70>
```

```
body(is_non_decreasing)
```

```
{
    for (i in 2:length(v)) if (v[i] < v[i - 1]) {
        return(FALSE)
    }
    return(TRUE)
}
```

```
args(is_non_decreasing)
```

```
function (v)
NULL
```

Question

Write a function with integer parameter n, that returns the Pascal Triangle with $n+1$ rows.

Solution

```r
triangle_pascal <- function(n) {
  m <- n+1
  T <- matrix(c(rep(1, m), rep(0, m*(m-1))), nrow=m, ncol=m)

  for (i in 2:m)
    T[i, 2:i] <- T[i-1, 1:i-1] + T[i-1, 2:i]

  for (i in 1:(m-1))
    T[i, (i+1):m] <- NA

  colnames(T) <- 0:n
  rownames(T) <- 0:n

  T
}

print(triangle_pascal(10), na.print=" " )
```

```
   0  1  2   3   4   5   6   7  8  9 10
0  1
1  1  1
2  1  2  1
3  1  3  3   1
4  1  4  6   4   1
5  1  5 10  10   5   1
6  1  6 15  20  15   6   1
7  1  7 21  35  35  21   7   1
8  1  8 28  56  70  56  28   8  1
9  1  9 36  84 126 126  84  36  9  1
10 1 10 45 120 210 252 210 120 45 10  1
```

Sanity check: `R` provides us with function `choose`

```r
n <- 5
map(0:n, ~ choose(., 0:.))
```

```
[[1]]
[1] 1

[[2]]
[1] 1 1

[[3]]
[1] 1 2 1

[[4]]
[1] 1 3 3 1

[[5]]
[1] 1 4 6 4 1

[[6]]
[1]  1  5 10 10  5  1
```

```r
t10 <- triangle_pascal(10)

for (n in 0:10)
  for (p in 0:n)
    stopifnot(t10[as.character(n), as.character(p)] == choose(n, p))
```

Question

How would you generate a Fibonacci sequence of length n ?

Recall the Fibonacci sequence is defined by

$$F_{n+2} = F_{n+1} + F_n \qquad F_1 = F_2 = 1$$

Solution

```r
fibo <- function(n) {
  res <- integer(n)
  res[1:2] <- 1
  for (k in 3:n) {
    res[k] <- res[k-1] + res[k-2]
  }
  return(res)
}

fibo(5)
```

```
[1] 1 1 2 3 5
```

Question

Write a function that perform binary search in a non-decreasing vector.

Solution

```r
binary_search <- function(v, x) {
  n <- length(v)
  i <- 1
  j <- n
  while (i <= j) {
    k <- (i+j) %/% 2
    if (v[k] == x) {
      return(k)
    } else if (v[k] < x) {
      i <- k + 1
    } else {
      j <- k - 1
    }
  }
  return(NA)
}
```

The provided code defines a function binary_search in R, which implements the binary search algorithm to find the position of a target value x within a sorted vector v. The binary search algorithm is efficient, operating in O(log n) time complexity, making it suitable for large datasets.
The function begins by determining the length of the vector v and storing it in the variable n. It then initializes two variables, i and j, to represent the lower and upper bounds of the search range, respectively. Initially, i is set to 1 (the first index of the vector), and j is set to n (the last index of the vector).
The core of the function is a while loop that continues as long as i is less than or equal to j. Within the loop, the midpoint k of the current search range is calculated using integer division (%/%) to ensure it is an integer. The value at index k in the vector v is then compared to the target value x. If the value at v[k] is equal to x, the function returns k, indicating that the target value has been found at this position. If v[k] is less than x, the search range is adjusted by setting i to k + 1, effectively discarding the lower half of the current range. Conversely, if v[k] is greater than x, the search range is adjusted by setting j to k - 1, discarding the upper half of the current range.
If the loop completes without finding the target value, the function returns NA, indicating that the target value x is not present in the vector v. This implementation assumes that the input vector v is sorted in ascending order.

Code and explanation provided by https://www.techiedelight.com/binary-search-in-r/ through copilot

!

Read Chapter on functions in Advanced R
In R, argument evalution is surprising, powerful but taming argument evaluation is real work.

Functional programming

In R, functions are first class entities, they can be defined at run-time, they can be used as function arguments. You can define list of functions, and iterate over them.

Try to use https://purrr.tidyverse.org.

! **Anonymous functions**

```
\(x) body
```

is a shorthand for

```
function (x) {
  body
}
```

Operators `purrr::map_???`

Question

Write truth tables for `&`, `|`, `&&`, `||`, `!` and `xor`
Hint: use `purrr::map`, function `outer()`

Solution

```r
vals <- c(TRUE, FALSE, NA)
ops <- c(`&`, `|`, `xor`)

truth <- purrr::map(ops, \(x) outer(vals,vals, x))

names(truth) <- (ops)
truth
```

```
$`.Primitive("&")`
      [,1]  [,2]  [,3]
[1,]  TRUE FALSE    NA
[2,] FALSE FALSE FALSE
[3,]    NA FALSE    NA

$`.Primitive("|")`
     [,1]  [,2] [,3]
[1,] TRUE  TRUE TRUE
[2,] TRUE FALSE   NA
[3,] TRUE    NA   NA

$`function (x, y) \n{\n    (x | y) & !(x & y)\n}`
      [,1]  [,2] [,3]
[1,] FALSE  TRUE   NA
[2,]  TRUE FALSE   NA
[3,]    NA    NA   NA
```

Question

Write a function that takes as input a square matrix and returns `TRUE` if it is lower triangular.

Solution

```r
lt <- function(A){
  n <- nrow(A)
  all(purrr::map_lgl(1:(n-1), \(x) all(0 == A[x, (x+1):n])))
}
```

Question

Use `map` , `choose` and proper use of pronouns to deliver the `n` first lines of the Pascal triangle using one line of code.
As far as the total number of operations is concerned, would you recommend this way of computing the Pascal triangle?

Solution

```
n <- 5

tp5 <- matrix(unlist(map(0:n,
          \(x) c(choose(x, 0:x), rep(0L, n-x)))),
       nrow=n+1,
       byrow=T)

rownames(tp5) <- 0:n

colnames(tp5) <- 0:n

tp5
```

```
  0 1  2  3 4 5
0 1 0  0  0 0 0
1 1 1  0  0 0 0
2 1 2  1  0 0 0
3 1 3  3  1 0 0
4 1 4  6  4 1 0
5 1 5 10 10 5 1
```

No. Using `map` and `choose`, we do not reuse previous computations. The total number of arithmetic operations is $\Omega(n^3)$, it should be $O(n^2)$.

! Read Chapter on Functional Programming in Advanced R

Further exploration

This notebook walked you through some aspects of `R` and its packages. We just saw the tip of the iceberg.

We barely mentioned:

- (Non-standard) Lazy evaluation
- Different flavors of object oriented programming
- Connection with `C++`: `RCpp`
- Connection with databases: `dbplyr`
- Building modeling pipelines: `tidymodels`
- Concurrency
- Building packages
- Building interactive Apps: `Shiny`
- Attributes (metadata)
- Formulae `formula`
- Strings `stringi`, `stringr`
- Dates `lubridate`
- and plenty other things ….

References

- https://www.statmethods.net/index.html
- https://www.datacamp.com/courses/free-introduction-to-r
- dplyr videos

- ggplot2 video tutorial
- cheatsheets

⚠ Readers who really want to learn `R` should spend time on

- R for Data Science by Wickham, Çetinkaya-Rundel, and Grolemund.
- Advanced R 2nd Edition by Wickham
- Advanced R Solutions by Grosser and Bumann
- Hands-On Programming with R by Grolemund

Don't go without Base R cheatsheet