Solutions to Exercises from Chapter 02

Contents

1 Exercises on R vectors

1.1 Exercise 01

Generate and display:

1. A vector of values $-2, -1, 0, 1, 2, 3, 4, 5, 6$.
2. A vector of values $10, 9, 8, 7, 6$.
3. A vector of values $0, 0.5, 1, 1.5, 2, 2.5, 3$, starting from the vector with values $0, 1, 2, 3, 4, 5, 6$.

SOLUTION

1. As the interval between a number in the series of components is of length 1, we can use the R syntax `Vini:Vfin`, where `Vini` and `Vfin` are the initial and final values of the series.

```
Vini <- 0
Vfin <- 6
V <- Vini:Vfin
print(V)
#> [1] 0 1 2 3 4 5 6
```

2. The series goes backward, but the interval between values has still length 1. Therefore, we can use the `Vini:Vfin` syntax here too.

```
Vini <- 10
Vfin <- 6
V <- Vini:Vfin
print(V)
#> [1] 10  9  8  7  6
```

3. The series required can be obtained from the first one, simply dividing, elementwise, by 2.

```
# First series
W <- 0:6

# Second series
V <- W/2
print(V)
#> [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0
```

1.2 Exercise 02

Generate a regular grid between $-\pi$ and $+\pi$ of 100 points. Find out the length Δx between two contiguous points of the grid.

SOLUTION

A regular grid can be generated between any two values x_L, x_R, with $x_R > x_L$, by simply requesting to the function `seq` the number of grid points needed. The function works out the regular spacing by itself. For the specific example suggested:

```
# Extremes of the interval
xL <- -pi
xR <- pi

# Grid
xgrid <- seq(xL,xR,length=100)

# First three grid points
print(xgrid[1:3])
#> [1] -3.141593 -3.078126 -3.014660

# Spacing of the grid
print(xgrid[2]-xgrid[1])
#> [1] 0.06346652
```

The length requested is $\Delta x \approx 0.06346652$.

1.3 Exercise 03

Create the following pattern,

1 1 1 2 2 3

using function `rep`.

SOLUTION

The three numbers repeated are $1, 2, 3$. The first is repeated three times, the second tw times, and the third just one time. Thus:

```
x <- rep(c(1,2,3),times=c(3,2,1))
print(x)
#> [1] 1 1 1 2 2 3
```

1.4 Exercise 04

Create the numeric pattern,

$$1\ 2\ 3\ 2\ 2\ 1\ 2\ 3\ 2\ 2$$

using function `rep`.

SOLUTION

This is a two-times repetition of 1 2 3 2 2. Therefore:

```
x <- rep(c(1,2,3,2,2),times=2)
print(x)
#>  [1] 1 2 3 2 2 1 2 3 2 2
```

1.5 Exercise 05

Create the numeric pattern,

$$1\ 1\ 1\ 2\ 2\ 3\ 1\ 1\ 1\ 2\ 2\ 3\ 1\ 1\ 1\ 2\ 2\ 3$$

using function `rep`.

SOLUTION

This is a repetition, three-times, of the same pattern of Exercise 03. We can therefore use `rep` in a nested way.

```
x <- rep(rep(c(1,2,3),times=c(3,2,1)),times=3)
print(x)
#>  [1] 1 1 1 2 2 3 1 1 1 2 2 3 1 1 1 2 2 3
```

1.6 Exercise 06

Create a vector `x` of length 30 using the following expression:

```
set.seed(123)
x <- sample(seq(0,1,length=101),size=30,replace=TRUE)
```

Print the value of the 9th, 18th and 27th component of the vector `x`. The `set.seed` function fixes the generation of the pseudo-random numbers so that the above code will always output the same numbers.

SOLUTION

This is the code involved, in one chunk.

```
# Creation of the vector
set.seed(123)
x <- sample(seq(0,1,length=101),size=30,replace=TRUE)

# Access and print value of 9th, 18th and 27th component
```

```r
print(x[c(9,18,27)])
#> [1] 1.00 0.08 0.35
```

Therefore, the components requested are 1.00, 0.08 and 0.35.

1.7 Exercise 07

Consider the vector `x` with the following components:

$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$$

Select and print only the components of `x` which contain even numbers. Then print the other components.

SOLUTION

The *filtered* selection of a vector's components can be carried out using appropriate indices. For example, `x[c(2,4,6)]` selects the second, fourth and sixth component of `x`, while `x[-1]` selects all components of `x` with the exception of the first. For the exercise given here, we can create a set of indices with only odd integers, and use them both as they are and with a minus sign to carry out the filtered selection.

```r
# Create vector
x <- 0:9

# Index for direct and filtered selection
idx <- c(1,3,5,7,9)

# Straight selection
print(x[idx])
#> [1] 0 2 4 6 8

# Filtered selection
print(x[-idx])
#> [1] 1 3 5 7 9
```

1.8 Exercise 08

Using a regular grid `x` of 361 values in the range $[0, 2\pi]$, calculate the corresponding values of the function

$$2\sin(x) - \cos(x),$$

and store them in a vector called `y`. Find the indices of `x` corresponding to 0, π and 2π, and verify that the values of `y` at these positions are -1, 1 and -1.

SOLUTION

The grid suggested essentially covers the whole range between 0 degrees and 360 degrees, with each grid point associated to an integer degree. So the grid starts as $0, 1, 2, 3, \ldots$, where these values have to be transformed in radians.

```r
# Grid (in radians)
x <- seq(0,2*pi,length=361)
print(x[1:5])
#> [1] 0.00000000 0.01745329 0.03490659 0.05235988 0.06981317

# Each grid point is one degree (remember the conversion)
print(x[1:5]*180/pi)
#> [1] 0 1 2 3 4
```

We can, next, calculate the values, bearing in mind that operations on vectors act in a parallel fashion.

```
# Values of function at grid points
y <- 2*sin(x)-cos(x)

# Print first few values
print(y[1:5])
#> [1] -1.0000000 -0.9649429 -0.9295918 -0.8939576 -0.8580511
```

The indices of the values to check suggested are easily derivable from the fact that the grid has a regular step size of one degree. Therefore 0 is associated with component 1, π, which is half-way in the full range, is associated with component 181 and 2π with component 361.

```
# Values requested
idx <- c(1,181,361)
print(y[idx])
#> [1] -1  1 -1
```

They are, indeed, equal to $-1, 1, -1$, as expected.

1.9 Exercise 09

Consider the two vectors `x` and `y` of different lengths:

$$x: \quad 1\ 2\ 3\ 4\ 5, \qquad y: \quad 2\ 4$$

What do you expect the result of `x+y` to be? Justify your answer and verify its correctness with R.

*__SOLUTION__

As vector `y` is shorter than vector `x`, it will be recycled to match the length of `x`. As this last vector has length 5, recycling `y` yields the vector

$$2\ 4\ 2\ 4\ 2$$

Therefore we expect `x+y` to return

$$1+2=3,\ 2+4=6,\ 3+2=5,\ 4+4=8,\ 5+2=7$$

Indeed:

```
x <- 1:5
y <- c(2,4)
print(x+y)
#> Warning in x + y: longer object length is not a multiple of shorter object
#> length
#> [1] 3 6 5 8 7
```

The warning is automatically returned by R to make sure the user is conscious about the recycling of components.

2 Exercises on simple graphics

2.1 Exercise 10

Using the function `plot`, draw an empty square with vertices at $(0,0)$, $(0,1)$, $(1,1)$ and $(1,0)$, in black. Draw also the two diagonals in red.

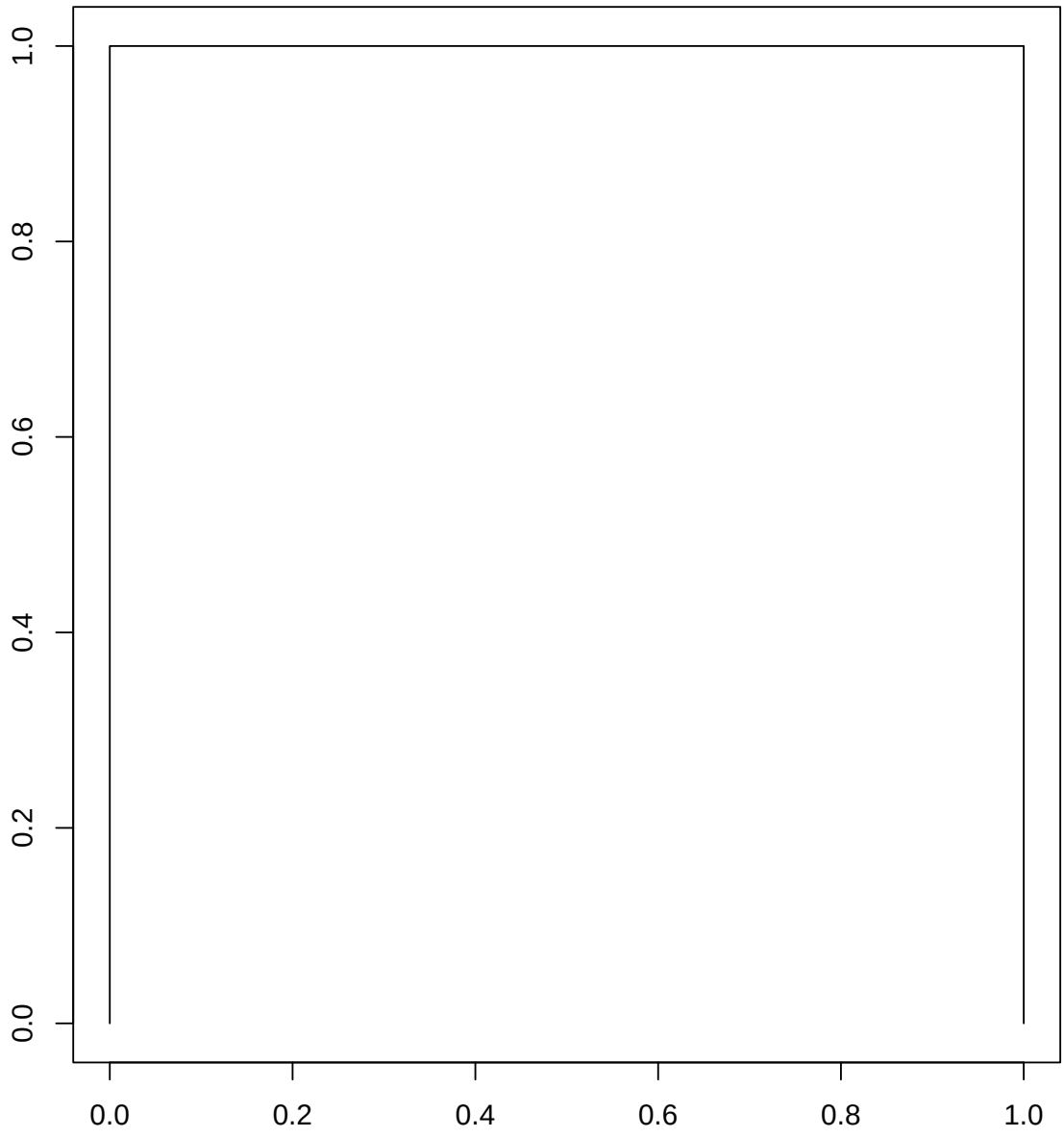
Repeat the same pattern two times, using function `rect` first, and `polygon` second. In both cases, the diagonals can be drawn using the function `segments`.

Make sure to use inline help and/or information on the internet for all functions needed.

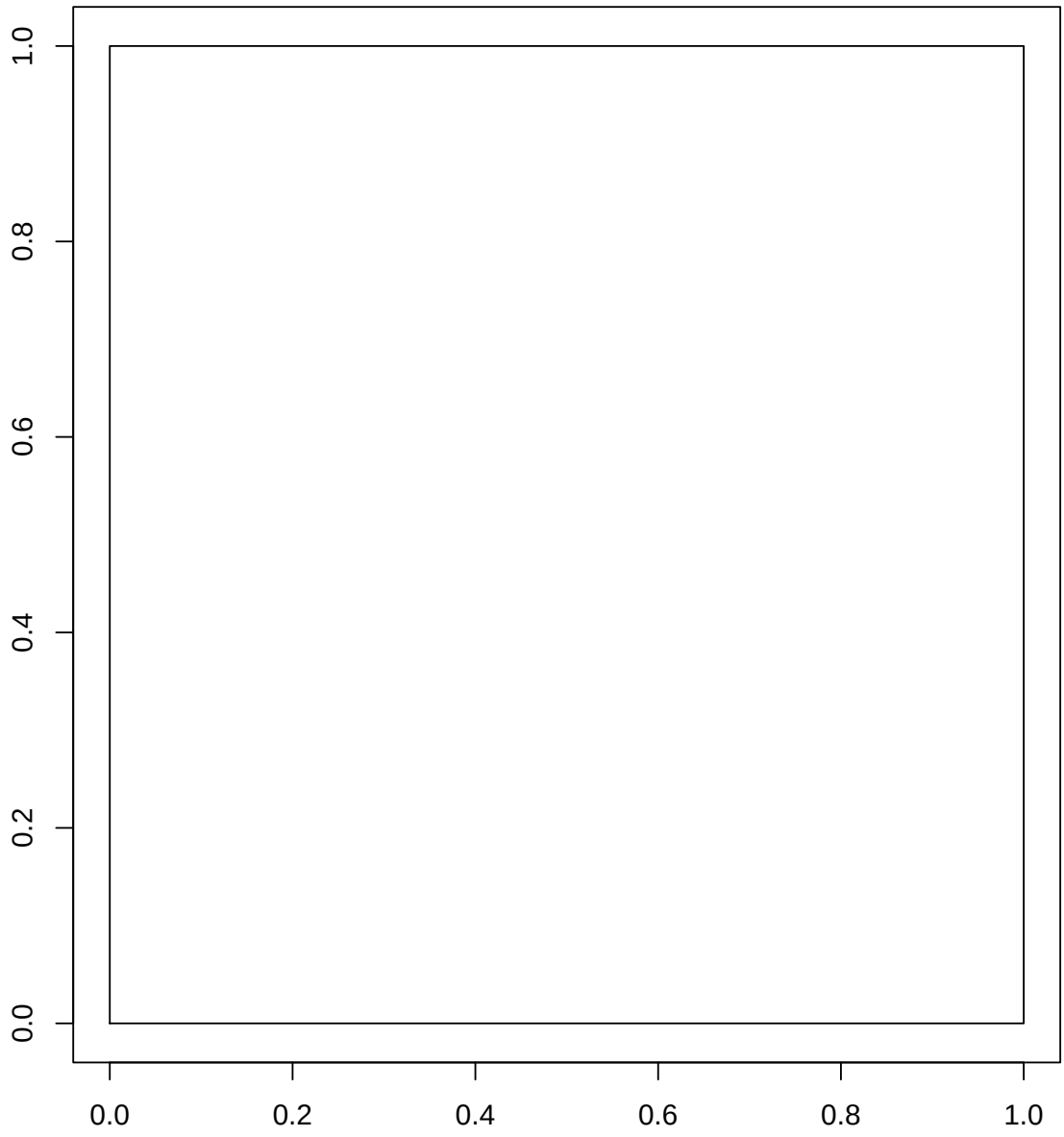
SOLUTION

When using `plot`, we need to decompose the box into four lines. Each line is a straight segment between two points. We can therefore use `plot` with just four points and with `type` set to `l` ("el'').

```
# x and y are the coordinates of the four points
# They need to be in the same order
x <- c(0,0,1,1)
y <- c(0,1,1,0)

# Now plot with type="l". Surprise: it's open
plot(x,y,type="l",xlab="",ylab="")
```



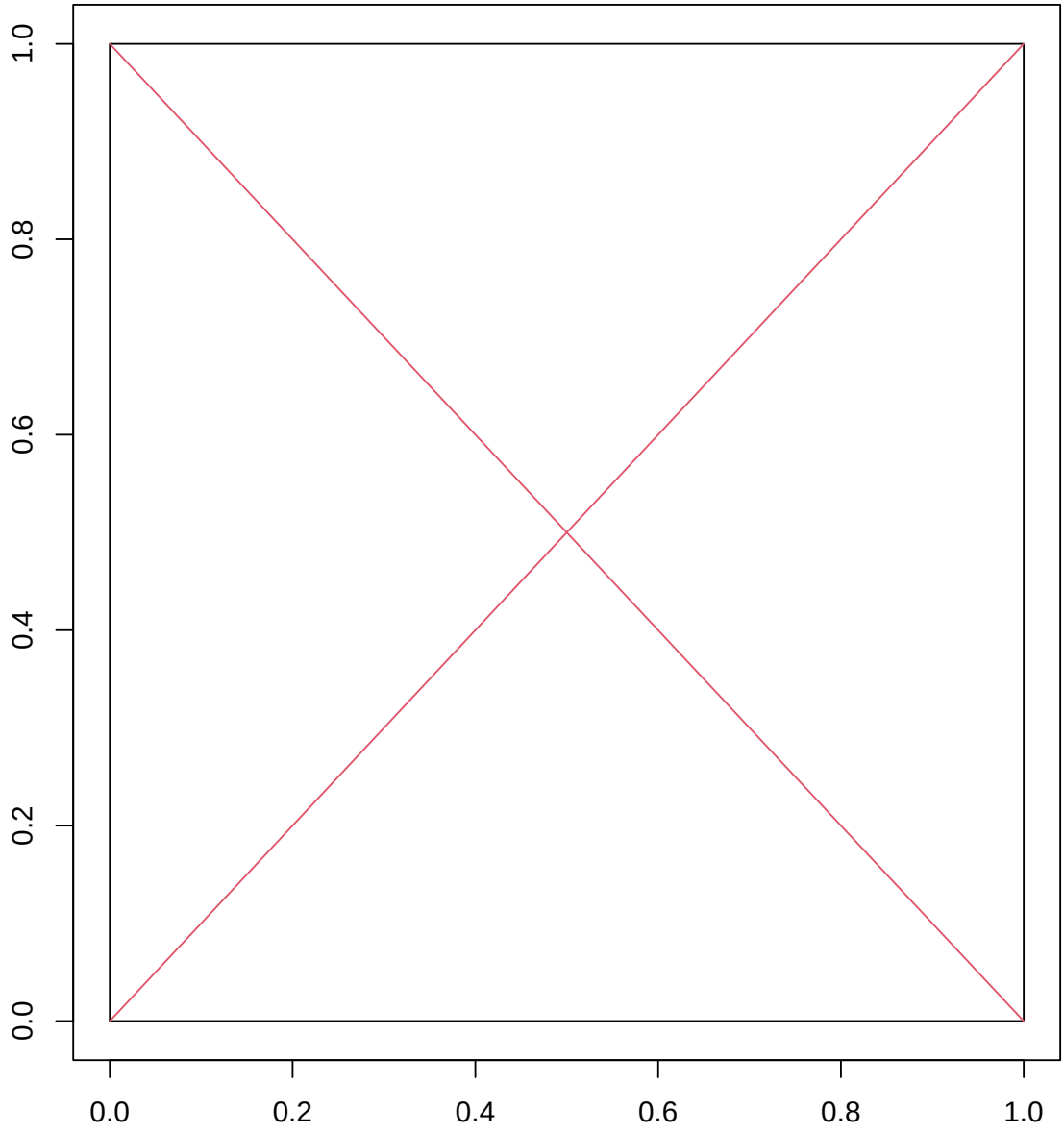

```r
# To close the square, repeat first point after the last
x <- c(0,0,1,1,0)
y <- c(0,1,1,0,0)
plot(x,y,type="l",xlab="",ylab="")
```



Finally, we can add the *cross* in red using the same concept of points joined by segments.

```
# Repeat what done earlier
x <- c(0,0,1,1,0)
y <- c(0,1,1,0,0)
plot(x,y,type="l",xlab="",ylab="")

# Plot each segment of cross, in turn
# Notice that as the plot has been already created,
```

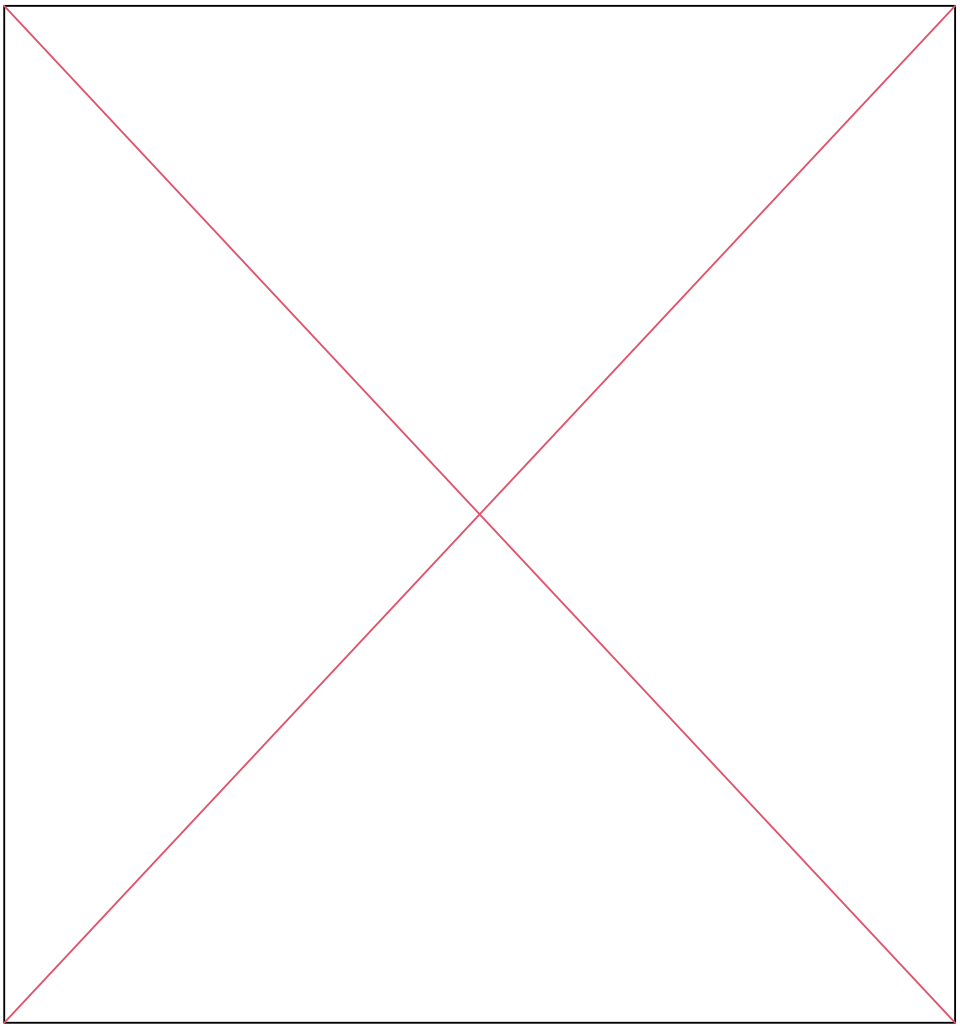
```r
# we use "points", rather than "plot"
points(x=c(0,1),y=c(0,1),type="l",col=2)
points(x=c(1,0),y=c(0,1),type="l",col=2)
```



A square can also be drawn using the function `rect`. A rectangle with sides parallel to the x-axis and y-axis is completely defined once the bottom-left and top-right corners are defined. The colour of the border of the rectangle is given by `border`. A plot should already be present before using `rect`. This is accomplished using `plot.new`. Observe that in this case the plot is not annotated. The segments for the cross can also be

produced using `segments`.

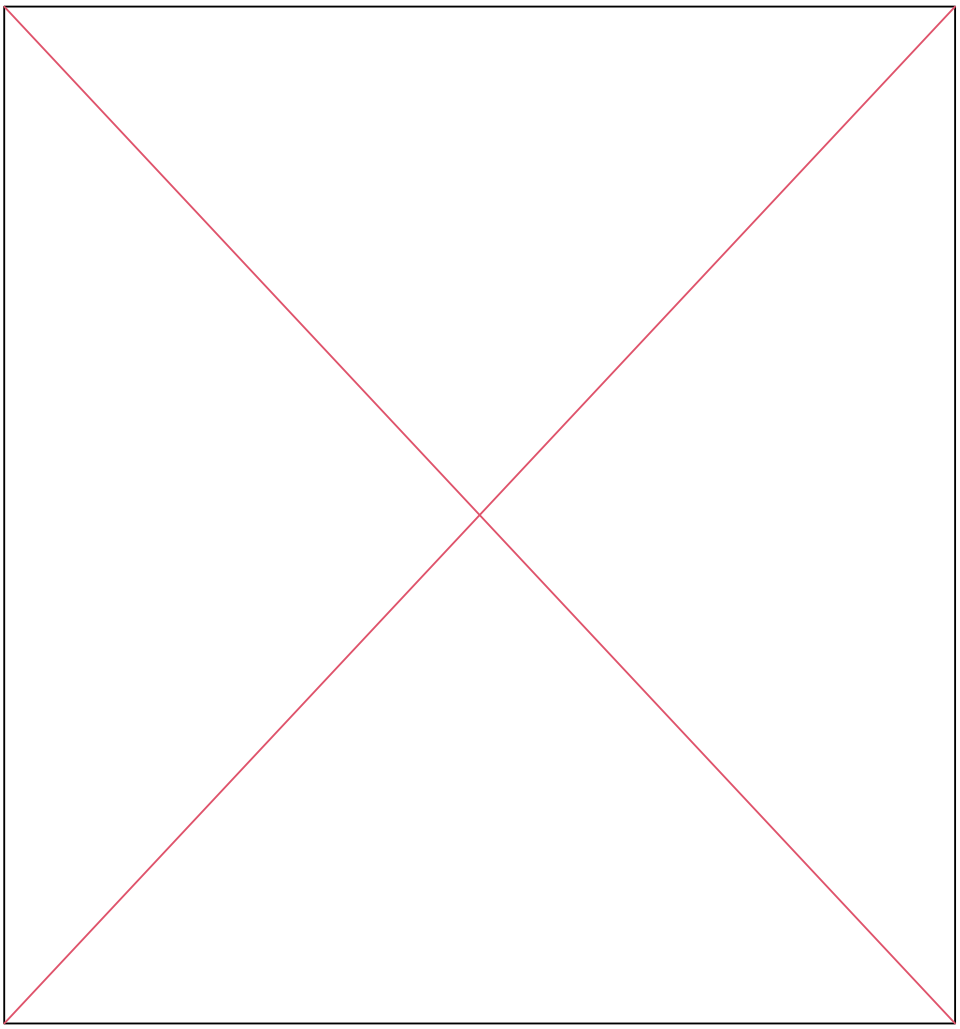
```
# Create an empty plot
plot.new()

# The whole square using "rect"
rect(xleft=0,ybottom=0,xright=1,ytop=1,border="black")

# Red cross
segments(x0=0,y0=0,x1=1,y1=1,col=2)
segments(x0=1,y0=0,x1=0,y1=1,col=2)
```



Everything can be repeated using `polygon`, which can be used only if a plot already exists (one can use `plot.new` for that). This acts similar to `plot`, the way we have used it to draw the rectangle. We can then add the cross using, again, `segments`.

```
# Create an empty plot
plot.new()

# Polygon (in this case a square)
polygon(x=c(0,0,1,1,0),y=c(0,1,1,0,0),border="black")
```

```r
# Red cross
segments(x0=0,y0=0,x1=1,y1=1,col=2)
segments(x0=1,y0=0,x1=0,y1=1,col=2)
```



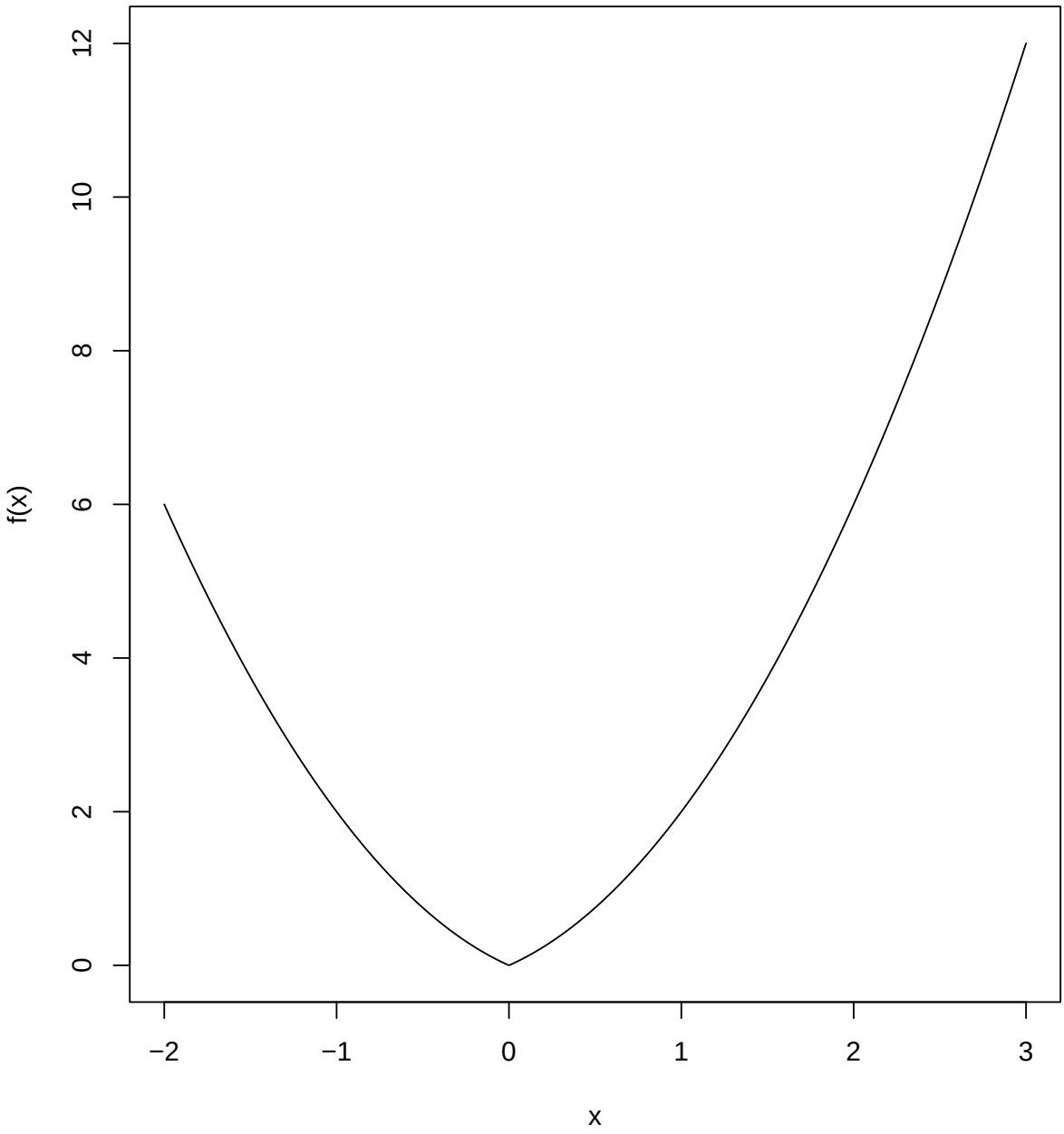
2.2 Exercise 11

Plot the function $f(x) = |x| + x^2$ in the interval $x \in [-2, 3]$, on two separate plots using, respectively, `plot` and `curve`. For the second plot, use a green, dashed line.

SOLUTION

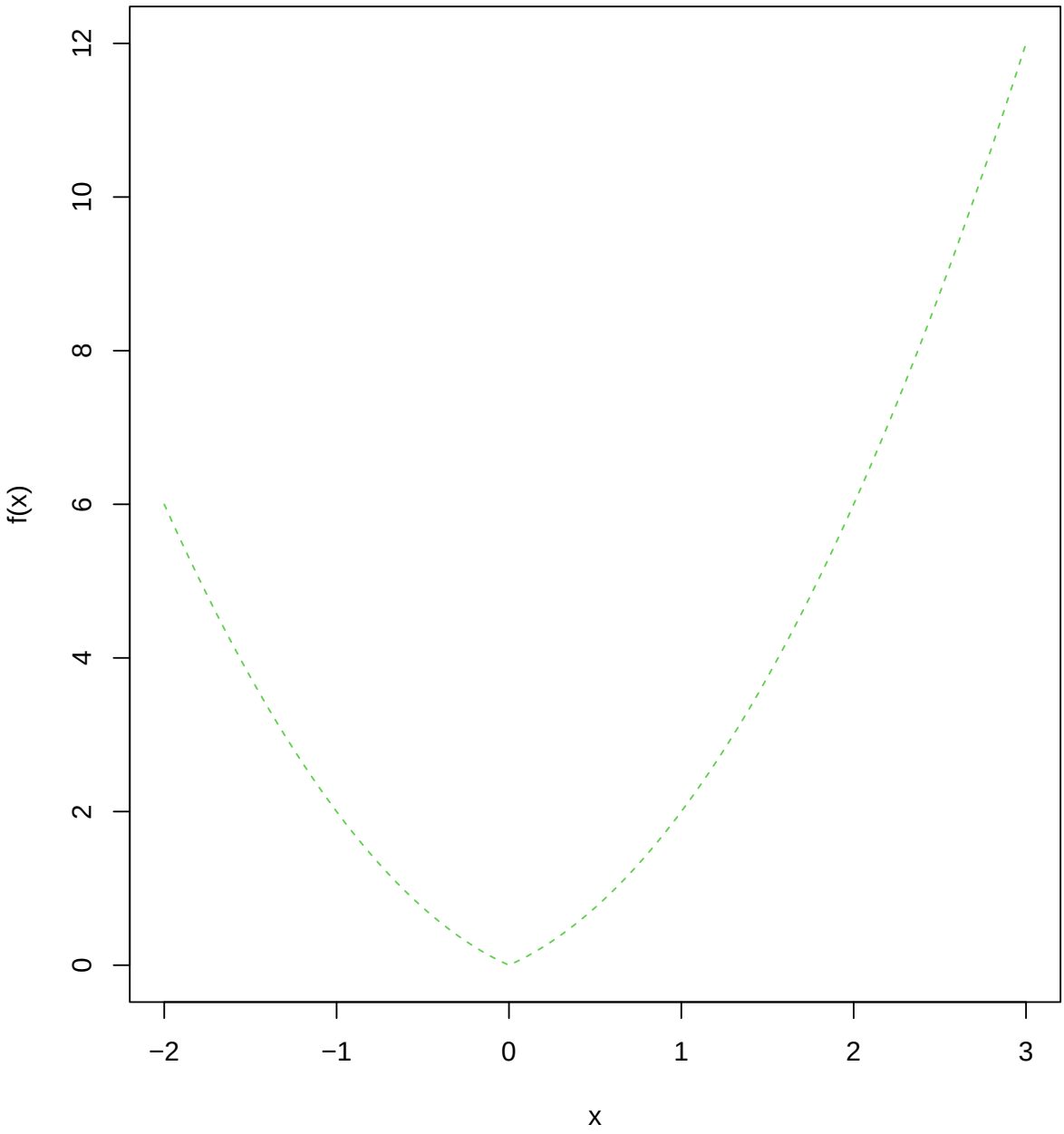
Without `curve`, we need first to create a grid in $[-2, 3]$ and calculate the function at all grid points.

```
# x grid
x <- seq(-2,3,length=1000)

# Values of function
y <- abs(x)+x^2

# Plot
plot(x,y,type="l",xlab="x",ylab="f(x)")
```



With `curve` there is no need to create the grid.

```
# For a gree dashed line we use "col" and "lty"
curve(abs(x)+x^2,from=-2,to=3,col=3,lty=2,xlab="x",ylab="f(x)")
```



2.3 Exercise 12

In the interval $x \in [-\pi, \pi]$, draw the following three experimental points,

x	$y = f(x)$
$-\pi$	-1
0	0
π	1

as black upward triangles and the following three curves:

a. $f(x) = \sin(x)$, in red

b. $f(x) = x^3 + (1/\pi + \pi^2)x$, in green

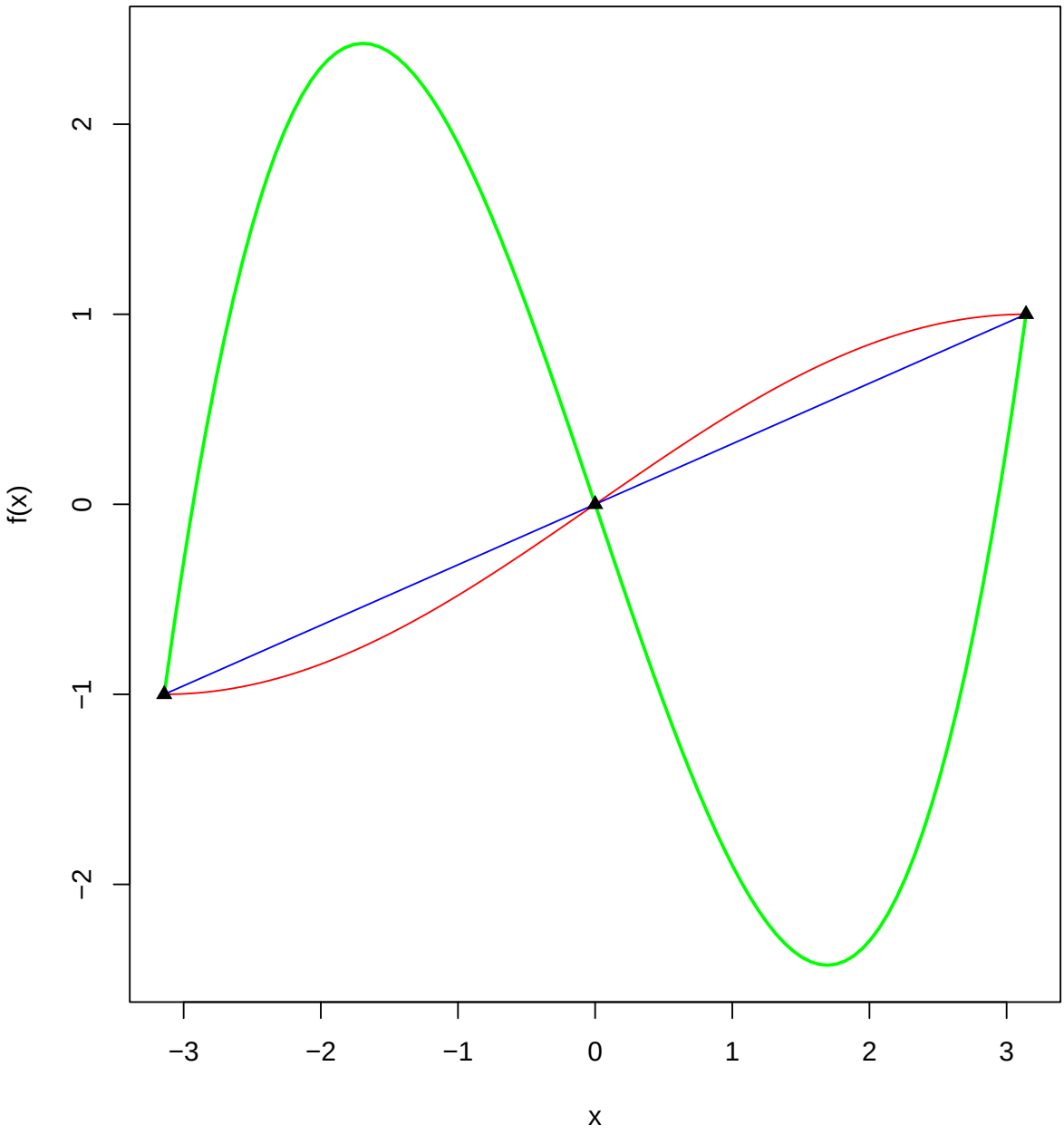
c. $f(x) = x/\pi$, in blue.

Make sure that the second curve as a line twice as thick as the other two curves.

SOLUTION

We can plot the curves with `curve` and add the triangles with `points`. But we first need to know the range of the three functions, so to represent all of them completely within one plot.

```
# Range of three functions
x <- seq(-pi,pi,length=1000)
y1 <- sin(x/2)
y2 <- 0.25*x^3+(1/pi-0.25*pi^2)*x
y3 <- x/pi
yy <- range(y1,y2,y3)

# Use "curve" to draw the functions
curve(sin(x/2),from=-pi,to=pi,col="red",xlab="x",ylab="f(x)",ylim=yy)
curve(0.25*x^3+(1/pi-0.25*pi^2)*x,from=-pi,to=pi,col="green",lwd=2,add=TRUE)
curve(x/pi,from=-pi,to=pi,col="blue",add=TRUE)

# Add points - The triangle is pch=17
points(c(-pi,0,pi),c(-1,0,1),pch=17,col="black")
```



3 Exercises on R functions

3.1 Exercise 13

Write a function with input `n` that outputs the first n integers as a vector.

SOLUTION

As the input is `n` the function's definition will contain `function(n)`. A simple implementation of the function

requested is:

```
# Definition
f <- function(n) {
  v <- 1:n

  return(v)
}
```

This is then easy to use:

```
# Generate and print vector of first 3 numbers
v <- f(3)
print(v)
#> [1] 1 2 3

# Generate and print vector of first 5 numbers
v <- f(5)
print(v)
#> [1] 1 2 3 4 5
```

3.2 Exercise 14

Modify the previous function (remember to change the name in order to preserve the previous function) to create a vector between two integer numbers.

SOLUTION

This task can be achieved using `n1:n2` rather than `1:n`.

```
# New function
g <- function(n1,n2) {
  v <- n1:n2

  return(v)
}

# Use the function
v <- g(-3,5)
print(v)
#> [1] -3 -2 -1  0  1  2  3  4  5
```

3.3 Exercise 15

Write a function `spc` to sum the sine and cosine of any angle `x` and a function `smc` that calculates the difference between the sine and cosine of any angle `x`. Finally, write a function that calculates the ratio

$$T(x) = \frac{\sin(x) + \cos(x)}{\sin(x) - \cos(x)},$$

for any given angle. Verify that the result is identical to the one returned by the function

$$\frac{\tan(x) + 1}{\tan(x) - 1}.$$

Finally, use the input `x <- c(0,pi/6,pi/4,pi/3,pi/2)` in the last function. What do you think is happening?

SOLUTION

The creation of the first two functions is implemented in the following code:

```
# Function spc
spc <- function(x) {
  y <- sin(x)+cos(x)

  return(y)
}

# Function smc
smc <- function(x) {
  y <- sin(x)-cos(x)

  return(y)
}
```

These two functions, like any new function, must be tested to check they return the expected output. We can try them now on an angle, $\pi/6$, for which it is easy to calculate both the sine and the cosine. In the first case we expect:

$$\sin\left(\frac{\pi}{6}\right)+\cos\left(\frac{\pi}{6}\right)=\frac{1}{2}+\frac{\sqrt{3}}{2}=\frac{\sqrt{3}+1}{2}\approx 1.366$$

and

$$\sin\left(\frac{\pi}{6}\right)-\cos\left(\frac{\pi}{6}\right)=\frac{1}{2}-\frac{\sqrt{3}}{2}=\frac{1-\sqrt{3}}{2}\approx -0.366.$$

```
# Test on x=pi/6
print(spc(pi/6))
#> [1] 1.366025
print(smc(pi/6))
#> [1] -0.3660254
```

So, the function defined, work. Next, we define a new function, `T`, that uses the previously-defined functions. As these are present in the working memory, they will be recognised:

```
# New (ratio) function
T <- function(x) {
  y <- spc(x)/smc(x)

  return(y)
}
```

To test this function we can still use the angle $\pi/6$. We should have:

$$\frac{\sin(\pi/6)+\cos(\pi/6)}{\sin(\pi/6)-\cos(\pi/6)}=\frac{(1+\sqrt{3})/2}{(1-\sqrt{3})/2}=\frac{1+\sqrt{3}}{1-\sqrt{3}}\approx -3.732.$$

Indeed:

```
# Test with x=pi/6
print(T(pi/6))
#> [1] -3.732051
```

So, this function works. Incidentally, dividing the original expression with sines and cosines by $\cos(x)$, we obtain an expression with tangents:

$$\frac{\sin(x)+\cos(x)}{\sin(x)-\cos(x)}=\frac{\tan(x)+1}{\tan(x)-1}.$$

This should give us again the same result when $x=\pi/6$:

```r
# Use tangents
print((tan(pi/6)+1)/(tan(pi/6)-1))
#> [1] -3.732051
```

Let us complete the exercise on the values given:

```r
# Values given
x <- c(0,pi/6,pi/4,pi/3,pi/2)

# Feed them to T
y <- T(x)

# Print
print(y)
#> [1] -1.000000e+00 -3.732051e+00 -1.273810e+16  3.732051e+00  1.000000e+00
```

The function built acts in parallel fashion on the five values provided. The third value returned is particularly high; this is in line with $\sin(\pi/4) - \cos(\pi/4)$ being zero and the ratio being, accordingly, infinity. The function has not returned `Inf` because `pi/4` is stored with finite precision.

4 Exercises on R objects and data handling

4.1 Exercise 16

a. Create the following objects in R:

```r
a <- 42
b <- "42"
c <- TRUE
d <- charToRaw("Hello")
e <- as.raw(c(0x48, 0x65, 0x6C, 0x6C, 0x6F))
```

b. Use `typeof` to determine the storage type of each object. What is the difference between `a` and `b`? What is the difference between `d` and `e`? For the raw objects `d` and `e`: convert them back to a character string using `rawToChar`.

c. Use `typeof` on a data frame and on one of its columns. Why are the results different? What does this tell you about how R stores data?

SOLUTION

a. The first task is easily carried out simply by typing in all variables in a console:

```r
# Clear workspace to check next set of objects is created
# (This is an interesting way of doing it)
rm(list=ls(all=TRUE))
ls()
#> character(0)

# Create objects
a <- 42
b <- "42"
c <- TRUE
d <- charToRaw("Hello")
e <- as.raw(c(0x48, 0x65, 0x6C, 0x6C, 0x6F))
ls()
#> [1] "a" "b" "c" "d" "e"
```

b. Check each type:

```
# a
print(typeof(a))
#> [1] "double"

# b
print(typeof(b))
#> [1] "character"

# c
print(typeof(c))
#> [1] "logical"

# d
print(typeof(d))
#> [1] "raw"
print(d)
#> [1] 48 65 6c 6c 6f

# e
print(typeof(e))
#> [1] "raw"
print(e)
#> [1] 48 65 6c 6c 6f
```

So, although `a` and `b` hve been typed with a same '`42''`, the first is a number in double precisionwhile the second is a character; they are two completely different objects.cis clearly a logical object. The last two are raw objects. By printing them out one can see their hexadecimal representation, which coincides. So, even if generated differently,dandecontain the same raw data. Let try and convert them to characters using the built in functionrawToChar':

```
# Convert raw data to character strings
Cd <- rawToChar(d)
print(Cd)
#> [1] "Hello"
Ce <- rawToChar(e)
print(Ce)
#> [1] "Hello"
```

Being `d` and `e` the same raw data, when converted they produce the same character string, the word "Hello''.

c. Here we could use one of the existing data frames, say `mtcars`.

```
# typeof applied to data frame
print(typeof(mtcars))
#> [1] "list"

# mtcars column name
print(colnames(mtcars))
#>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#> [11] "carb"

# typeof applied to data frame columns
print(typeof(mtcars$mpg))
#> [1] "double"
```

Thus, a data frame is a list with some attributes that make it a data frame. The command `typeof` in essence reveals how data are stored. So, data are stored as in a list for the whole data frame, but as numbers for one of its columns.

4.2 Exercise 17

Type in the following (exact) syntax in a console:

```
G <- list(mtcars,diag(5),solve(diag(5)),LETTERS[1:10])
```

a. What is the `typeof` for `G` and `G[[1]]`? Explain the difference.

b. What data structure have `G[[2]]` and `G[[3]]`? Print the objects to verify this is true.

c. What is the data structure and data type of `G[[4]]`?

SOLUTION

Typing is straightforward:

```
# Clear workspace to check next set of objects is created
rm(list=ls(all=TRUE))
ls()
#> character(0)

# Create object provided
G <- list(mtcars,diag(5),solve(diag(5)),LETTERS[1:10])
ls()
#> [1] "G"
```

a. Data types are assessed with `typeof`:

```
# Data type of G
print(typeof(G))
#> [1] "list"

# Data type of G[[1]]
print(typeof(G[[1]]))
#> [1] "list"
```

Both objects are lists. But the second is, in fact, a built-in data frame, `mtcars`, which is recognised as a list data type because all data frames are lists with additional attributes.

```
# Attributes reveal internal structure
print(attributes(G))
#> NULL
print(attributes(G[[1]]))
#> $names
#>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#> [11] "carb"
#>
#> $row.names
#>  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
#>  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
#>  [7] "Duster 360"          "Merc 240D"           "Merc 230"
#> [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
#> [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
#> [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
#> [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
```

```
#> [22] "Dodge Challenger"    "AMC Javelin"          "Camaro Z28"
#> [25] "Pontiac Firebird"    "Fiat X1-9"            "Porsche 914-2"
#> [28] "Lotus Europa"        "Ford Pantera L"       "Ferrari Dino"
#> [31] "Maserati Bora"       "Volvo 142E"
#>
#> $class
#> [1] "data.frame"
```

So, `G` has no attributes, it's a pure list. Differently, one of the attributes of `G[[1]]` is `class` which, in this specific instance, '`classify''`mtcars' as a data frame.

b.

```
# Data structures are revealed by str
# G[[2]]
str(G[[2]])
#>  num [1:5, 1:5] 1 0 0 0 0 0 1 0 0 0 ...

# G[[3]]
str(G[[3]])
#>  num [1:5, 1:5] 1 0 0 0 0 0 1 0 0 0 ...
```

So, both objects are 2D arrays (matrices), containing numbers. Being a '`matrix'' is made possible by the attribute`dim':

```
# Attributes of G[[2]] and G[[3]]
print(attributes(G[[2]]))
#> $dim
#> [1] 5 5
print(attributes(G[[3]]))
#> $dim
#> [1] 5 5

# Print actual objects
print(G[[2]])
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    0    0    0    0
#> [2,]    0    1    0    0    0
#> [3,]    0    0    1    0    0
#> [4,]    0    0    0    1    0
#> [5,]    0    0    0    0    1
print(G[[3]])
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    0    0    0    0
#> [2,]    0    1    0    0    0
#> [3,]    0    0    1    0    0
#> [4,]    0    0    0    1    0
#> [5,]    0    0    0    0    1
```

These two matrices are identical, the 5×5 identity matrix, I_5. The function `solve`, when applied to a matrix, returns its inverse, and the inverse of I_5 is still I_5.

c. Let us explore `G[[4]]` now:

```
print(typeof(G[[4]]))
#> [1] "character"
print(attributes(G[[4]]))
```

```
#> NULL

# What's in G[[4]]?
print(G[[4]])
#>  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

`G[[4]]` is a vector of characters. As all vectors, it does not have attributes, to start with. In fact, the object `LETTERS` is a built-in vector containing the whole alphabet in uppercase:

```
# Uppercase alphabet
print(LETTERS)
#>  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
#> [20] "T" "U" "V" "W" "X" "Y" "Z"
```

4.3 Exercise 18

Select a built-in data frame different from `mtcars` and transform it into a matrix. Is it possible straight away? Or, you need to take some decisions on the variables contained, first?

SOLUTION

One of the data frames seen earlier is `iris`. It includes five columns, the last one being a column of factors.

```
# Data frame iris
str(iris)
#> 'data.frame':    150 obs. of  5 variables:
#>  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
#>  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
#>  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
#>  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

unique(iris$Species)
#> [1] setosa     versicolor virginica
#> Levels: setosa versicolor virginica
```

There are only three factor levels that could be turned into the integers $1, 2, 3$. We can therefore start by creating a vector of integers to transform the last column of `iris`

```
# Turn factors into integers
new_factors <- as.integer(iris$Species)

# Replace last column with integers (keep iris safe)
M <- cbind(iris[,1:4],new_factors)
str(M)
#> 'data.frame':    150 obs. of  5 variables:
#>  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
#>  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
#>  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
#>  $ new_factors : int  1 1 1 1 1 1 1 1 1 1 ...
```

The new data frame `M` contains only numbers and can be turned into a matrix with one command:

```
# Turn data frame into mtrix
M <- as.matrix(M)
str(M)
#>  num [1:150, 1:5] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
```

```
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:5] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" ...
```

There are some left over attributes. What are they?

```
# Attributes of M
print(attributes(M))
#> $dim
#> [1] 150   5
#>
#> $dimnames
#> $dimnames[[1]]
#> NULL
#>
#> $dimnames[[2]]
#> [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "new_factors"
```

Besides the `dim` attribute that characterises a matrix, there is another attribute, `dimnames`, which is a left over from when `M` was a data frame. We can get rid of it:

```
# Get rid of unnecessary attributes
attributes(M)$dimnames <- NULL

# New M (a pure matrix)
str(M)
#>  num [1:150, 1:5] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
```

The same task could have been achieved using different procedures but factors and attributes must be always taken into account.

4.4 Exercise 19

Create an $2 \times 3 \times 4$ array and fill it naturally (i.e. following the built-in order of your machine) with the first 24 integers.

SOLUTION

An array with three dimensions can be thought as a `cubic matrix'' made of slices, each one of them being a`square matrix''. Filling the array, if no special arrangements are made, follows whatever filling order is hard-coded in the language. Let us fill the array, which we can call `A`, and discover how the filling is done.

```
# Create the array
A <- array(1:24,c(2,3,4))

# Print to discover order of filling
print(A)
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
```

```
#> [2,]    8   10   12
#>
#> , , 3
#>
#>      [,1] [,2] [,3]
#> [1,]   13   15   17
#> [2,]   14   16   18
#>
#> , , 4
#>
#>      [,1] [,2] [,3]
#> [1,]   19   21   23
#> [2,]   20   22   24
```

In order to identify which part of the array is being filled, we can use the letters `i,j,k`. We can then see that the first to be filled is the bottom slice (`k=1`); the filling of this slice/matrix follows the columns order: rows (index `i`) change fast, columns (index `j`) change slow. This means that a different filling order will have to be thought appropriately.

The different filling ordering can be also visualised by extracting the indices from the array, with the command `arrayInd`:

```
# Extract indices from array
idx <- arrayInd(seq_along(A), dim(A))

# Merge these with array content.
# This work because it's a natural filling order
T <- cbind(idx,value=A)
print(T)
#>              value
#>  [1,] 1 1 1     1
#>  [2,] 2 1 1     2
#>  [3,] 1 2 1     3
#>  [4,] 2 2 1     4
#>  [5,] 1 3 1     5
#>  [6,] 2 3 1     6
#>  [7,] 1 1 2     7
#>  [8,] 2 1 2     8
#>  [9,] 1 2 2     9
#> [10,] 2 2 2    10
#> [11,] 1 3 2    11
#> [12,] 2 3 2    12
#> [13,] 1 1 3    13
#> [14,] 2 1 3    14
#> [15,] 1 2 3    15
#> [16,] 2 2 3    16
#> [17,] 1 3 3    17
#> [18,] 2 3 3    18
#> [19,] 1 1 4    19
#> [20,] 2 1 4    20
#> [21,] 1 2 4    21
#> [22,] 2 2 4    22
#> [23,] 1 3 4    23
#> [24,] 2 3 4    24
```

We can appreciate the fastest-changing index `i` (`121212`), the intermediate-changing index `j` (`112233`), and the slowest-changing index `k` (`111111`).

4.5 Exercise 20

Generate an Hermitian matrix. Recall: an Hermitian matrix, H, obeys the property

$$H^\dagger = H,$$

where † indicates a complex-conjugate transpose of the original matrix. This property means that H has real numbers along the diagonal, while for the other components h_{ij} (in general complex numbers), we have:

$$h^*_{ij} = h_{ji}.$$

SOLUTION

One way of creating the Hermitian matrix is indicated here.

```
# Fix random seed for results replication
set.seed(1234)

# Random set of 9 complex numbers with real
# and imaginary parts between -2 and 2
h <- complex(real=sample(-2:2,size=9,replace=TRUE),
             imaginary=sample(-2:2,size=9,replace=TRUE))

# Create a matrix with complex values
A <- matrix(h,ncol=3)

# Hermitian is sum of A and transpose of complex conjugate
H <- 0.5*(A+t(Conj(A)))
print(H)
#>           [,1]      [,2]    [,3]
#> [1,] 1.0+0.0i  0.0+0.5i  1.5+0i
#> [2,] 0.0-0.5i -2.0+0.0i  0.5+1i
#> [3,] 1.5+0.0i  0.5-1.0i -1.0+0i
```

The matrix is Hermitian. Besides having all values along the diagonal as real, all off-diagonal components enjoy $h^*_{ij} = h_{ji}$. For example:

$$h_{23} = \frac{1}{2} + \mathrm{i} \;\Rightarrow\; h^*_{23} = \frac{1}{2} - \mathrm{i} = h_{32}.$$

4.6 Exercise 21

A quick way of generating all combinations of finite factors is with the command `expand.grid`. Use the help facility in R to understand how this command works. Then create a data frame for all possible combinations of:

- `colour`: values `"g","r","b"`;
- `size`: values `"xs","s","m","l","xl"`;
- `gender`: value `"m","f"`.

Finally, create a list of the three vectors `colour`, `size`, and `gender`.

SOLUTION

The `expand.grid` command is quite easy to use. Simply, one include all vectors (with relative values) as arguments. In our case:

```r
# Content of data frame
colour <- c("g","r","b")
size <- c("xs","s","m","l","xl")
gender <- c("m","f")

# Generate data frame of all combinations of jumpers
jumpers <- expand.grid(colour=colour,size=size,gender=gender)
str(jumpers)
#> 'data.frame':    30 obs. of  3 variables:
#>  $ colour: Factor w/ 3 levels "g","r","b": 1 2 3 1 2 3 1 2 3 1 ...
#>  $ size  : Factor w/ 5 levels "xs","s","m","l",..: 1 1 1 2 2 2 3 3 3 4 ...
#>  $ gender: Factor w/ 2 levels "m","f": 1 1 1 1 1 1 1 1 1 1 ...
#>  - attr(*, "out.attrs")=List of 2
#>   ..$ dim     : Named int [1:3] 3 5 2
#>   .. ..- attr(*, "names")= chr [1:3] "colour" "size" "gender"
#>   ..$ dimnames:List of 3
#>   .. ..$ colour: chr [1:3] "colour=g" "colour=r" "colour=b"
#>   .. ..$ size  : chr [1:5] "size=xs" "size=s" "size=m" "size=l" ...
#>   .. ..$ gender: chr [1:2] "gender=m" "gender=f"
print(jumpers[1:5,])
#>   colour size gender
#> 1      g   xs      m
#> 2      r   xs      m
#> 3      b   xs      m
#> 4      g    s      m
#> 5      r    s      m
```

The three vectors of characters (they are transformed into factors by `expand.grid`) can be easily accommodated into a list with the standard construction:

```r
# Build a list of factors
features <- list(colour=colour,size=size,gender=gender)
print(features)
#> $colour
#> [1] "g" "r" "b"
#>
#> $size
#> [1] "xs" "s"  "m"  "l"  "xl"
#>
#> $gender
#> [1] "m" "f"
```

4.7 Exercise 22

Consider the dataset `penguins`. A `help(penguins)` reveals the following description:

Data on adult penguins covering three species found on three islands in the Palmer Archipelago, Antarctica, including their size (flipper length, body mass, bill dimensions), and sex.

Use `aggregate` to find the average of the penguins' flipper length, body mass, and bill dimension, aggregating data by island, species, and sex. Where can you find the penguins with highest body mass index? And are they male or female?

SOLUTION

It is always a good idea to explore the content of a data frame using `str`.

```
# What;s in penguins?
str(penguins)
#> 'data.frame':    344 obs. of  8 variables:
#>  $ species    : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
#>  $ island     : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
#>  $ bill_len   : num  39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
#>  $ bill_dep   : num  18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
#>  $ flipper_len: int  181 186 195 NA 193 190 181 195 193 190 ...
#>  $ body_mass  : int  3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
#>  $ sex        : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
#>  $ year       : int  2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

So, we can see that `penguins` is a data frame with 344 observations of 8 variables. Three of these variables are factors, ideal for the aggregation. They are `species`, `island`, and `sex`. These are exactly the factors requested for aggregation in the question. We can then move to formulating the aggregation using `aggregate`.

```
# Aggregation by species, island, and sex
df <- aggregate(cbind(bill_len,bill_dep,flipper_len,body_mass) ~
                species+island+sex,data=penguins,FUN=mean)
print(df)
#>       species    island    sex bill_len bill_dep flipper_len body_mass
#> 1      Adelie    Biscoe female 37.35909 17.70455    187.1818  3369.318
#> 2      Gentoo    Biscoe female 45.56379 14.23793    212.7069  4679.741
#> 3      Adelie     Dream female 36.91111 17.61852    187.8519  3344.444
#> 4   Chinstrap     Dream female 46.57353 17.58824    191.7353  3527.206
#> 5      Adelie Torgersen female 37.55417 17.55000    188.2917  3395.833
#> 6      Adelie    Biscoe   male 40.59091 19.03636    190.4091  4050.000
#> 7      Gentoo    Biscoe   male 49.47377 15.71803    221.5410  5484.836
#> 8      Adelie     Dream   male 40.07143 18.83929    191.9286  4045.536
#> 9   Chinstrap     Dream   male 51.09412 19.25294    199.9118  3938.971
#> 10     Adelie Torgersen   male 40.58696 19.39130    194.9130  4034.783
```

The data frame returned contains all aggregated data (*via* mean). So, for example, the average body mass index of the male Gentoo penguins in Biscoe island, is 5484.836. In this case this is also the highest value of body mass index in our data. Therefore, the penguins with the highest body mass index are the Gentoo living in Biscoe island, and they are male.

4.8 Exercise 23

Carry out the same aggregation of the previous exercise, this time using function `by`.

SOLUTION

According to the `by` syntax explained in the text, the same aggregation is obtained as follows:

```
# Aggregation using by
ltmp <- by(penguins[,c("bill_len","bill_dep",
                       "flipper_len","body_mass")],
           list(penguins$species,penguins$island,penguins$sex),
           colMeans)
```

We know that `by` does not return a data frame but a list. We could see what's in the list using `str`:

```
# What's in the list?
str(ltmp)
#> List of 18
#>  $ : Named num [1:4] 37.4 17.7 187.2 3369.3
```

```
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : Named num [1:4] 45.6 14.2 212.7 4679.7
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : Named num [1:4] 36.9 17.6 187.9 3344.4
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : Named num [1:4] 46.6 17.6 191.7 3527.2
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : Named num [1:4] 37.6 17.6 188.3 3395.8
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : NULL
#>  $ : Named num [1:4] 40.6 19 190.4 4050
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : Named num [1:4] 49.5 15.7 221.5 5484.8
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : Named num [1:4] 40.1 18.8 191.9 4045.5
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : Named num [1:4] 51.1 19.3 199.9 3939
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : Named num [1:4] 40.6 19.4 194.9 4034.8
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : NULL
#>  - attr(*, "dim")= int [1:3] 3 3 2
#>  - attr(*, "dimnames")=List of 3
#>   ..$ : chr [1:3] "Adelie" "Chinstrap" "Gentoo"
#>   ..$ : chr [1:3] "Biscoe" "Dream" "Torgersen"
#>   ..$ : chr [1:2] "female" "male"
#>  - attr(*, "call")= language by.data.frame(data = penguins[, c("bill_len", "bill_dep", "flipper_len"
#>  - attr(*, "class")= chr "by"
```

So, the output is more complicted than the one from `aggregate`. We can output specific elements of the list, but we need a mapping between the specific element and the combination of `species`, `island` and `sex`. This could be studied considering that the ordering of the list follows the factors grid created as

$$\text{species} \times \text{island} \times \text{sex}.$$

Thus, as the first element of `species` is `"Adelie"`, the first of `"island"` is `"Biscoe"` and the first of `sex` is `"female"` (you can read that at the bottom of the previous output), the first element of the list `ltmp` will be the one corresponding to this combination. This should coincide with the same combination as derived with aggregate. Indeed:

```
# Element 9
print(ltmp[[1]])
#>    bill_len    bill_dep flipper_len   body_mass
#>    37.35909    17.70455   187.18182  3369.31818

# Corresponding combination with aggregate
print(df[df$species == "Adelie" & df$island == "Biscoe" &
           df$sex == "female",])
#>   species island    sex bill_len bill_dep flipper_len body_mass
```

```
#> 1  Adelie Biscoe female 37.35909 17.70455    187.1818  3369.318
```

The aggregate means are the same. The second element corresponds to the combination `"Chinstrap"`, `"Biscoe"`, and `"female"` (that is, the first factor is the fastest changing, the second is intermediate, and the third is the slowest changing). There exist no data for this combination. This is also evident by the `aggregate` data frame not having this combination. So the list `ltmp` returns `NULL` as second element:

```
# Data frame
print(df)
#>      species    island    sex bill_len bill_dep flipper_len body_mass
#> 1     Adelie    Biscoe female 37.35909 17.70455    187.1818  3369.318
#> 2     Gentoo    Biscoe female 45.56379 14.23793    212.7069  4679.741
#> 3     Adelie     Dream female 36.91111 17.61852    187.8519  3344.444
#> 4  Chinstrap     Dream female 46.57353 17.58824    191.7353  3527.206
#> 5     Adelie Torgersen female 37.55417 17.55000    188.2917  3395.833
#> 6     Adelie    Biscoe   male 40.59091 19.03636    190.4091  4050.000
#> 7     Gentoo    Biscoe   male 49.47377 15.71803    221.5410  5484.836
#> 8     Adelie     Dream   male 40.07143 18.83929    191.9286  4045.536
#> 9  Chinstrap     Dream   male 51.09412 19.25294    199.9118  3938.971
#> 10    Adelie Torgersen   male 40.58696 19.39130    194.9130  4034.783

# Element 2
print(ltmp[[2]])
#> NULL
```

A quick mapping can be done using some expressions in a single line.

```
# dimnames is an attribute of ltmp
print(dimnames)
#> function (x)  .Primitive("dimnames")

# A grid created systematicaly will follow the (natural)
# order of the list. This is the mapping needed.
print(expand.grid(dimnames(ltmp)))
#>          Var1      Var2   Var3
#> 1     Adelie    Biscoe female
#> 2  Chinstrap    Biscoe female
#> 3     Gentoo    Biscoe female
#> 4     Adelie     Dream female
#> 5  Chinstrap     Dream female
#> 6     Gentoo     Dream female
#> 7     Adelie Torgersen female
#> 8  Chinstrap Torgersen female
#> 9     Gentoo Torgersen female
#> 10    Adelie    Biscoe   male
#> 11 Chinstrap    Biscoe   male
#> 12    Gentoo    Biscoe   male
#> 13    Adelie     Dream   male
#> 14 Chinstrap     Dream   male
#> 15    Gentoo     Dream   male
#> 16    Adelie Torgersen   male
#> 17 Chinstrap Torgersen   male
#> 18    Gentoo Torgersen   male

# Mapping
```

```r
ltmp_map <- expand.grid(dimnames(ltmp))
print(ltmp_map)
#>          Var1      Var2   Var3
#> 1     Adelie    Biscoe female
#> 2  Chinstrap    Biscoe female
#> 3     Gentoo    Biscoe female
#> 4     Adelie     Dream female
#> 5  Chinstrap     Dream female
#> 6     Gentoo     Dream female
#> 7     Adelie Torgersen female
#> 8  Chinstrap Torgersen female
#> 9     Gentoo Torgersen female
#> 10    Adelie    Biscoe   male
#> 11 Chinstrap    Biscoe   male
#> 12    Gentoo    Biscoe   male
#> 13    Adelie     Dream   male
#> 14 Chinstrap     Dream   male
#> 15    Gentoo     Dream   male
#> 16    Adelie Torgersen   male
#> 17 Chinstrap Torgersen   male
#> 18    Gentoo Torgersen   male
```

It is clear that `by` creates a structure which is different from a data frame, differently from `aggregate`. Therefore, `by` can be used for different tasks.