R Training 4

Martin Chan

23 February 2019

Contents

Introduction - R Training 4

This is the training notes for 4th Rainmakers R Training session, where the aim is to introduce R with a focus on analysing survey data, or data types commonly found in market or strategy research. The document will cover more advanced topics in R (*advanced* may sound intimidating - but I guarantee they're highly valuable!), including:

- simulating your own data
- for loops
- `apply()` family of functions
- how to write your own custom functions.

These tricks and concepts will still be introduced with *practical usage in research* in mind and therefore will focus more on data examples close to what we find in research. However, if you'd like to do a bit of further reading do check out this blogpost available on R-bloggers https://www.r-bloggers.com/r-tutorial-on-the-apply-family-of-functions/.

Like in the previous training documents, the R code used here will largely follow the tidyverse / dplyr conventions of using pipe operators (`%>%`). The principle behind this is the cleaner, more readable code that results from using these conventions. Despite the fact they make the code marginally slower (literally by micro-seconds), the bottlenecks in data analysis tends to be in thinking and planning (coding) rather than computation, and hence the trade-off for more readable code is surely justified.

1. Simulating your own dataset

To completely eliminate any possibility of infringeing GDPR or other privacy issues that comes from using a 'real' dataset, we will try to simulate our own datasets for the purpose of this training. The main workhorse functions that we will use to do this are:

- `sample()`
- `tibble()` / `data.frame()` (doesn't really matter which one you use - I'd recommend `tibble`) Read this if you want the details
- `rpois()` (I've found most useful for generating age data)
- `matrix()` - useful for generating a large number of random binary variables

Before you run the following, ensure you start with loading tidyverse with `library(tidyverse)`. The first step is to create some vectors with a consistent size of 5000, which we will use as the columns in our data frame / tibble. With the `sample()` function, the `x` argument specifies what you would like to have as values, and the `prob` argument specifying the probability of occurrence for each of those values (omitting this argument will return an even split). Here is an example created for gender, which I've labelled as `q2_gender`:

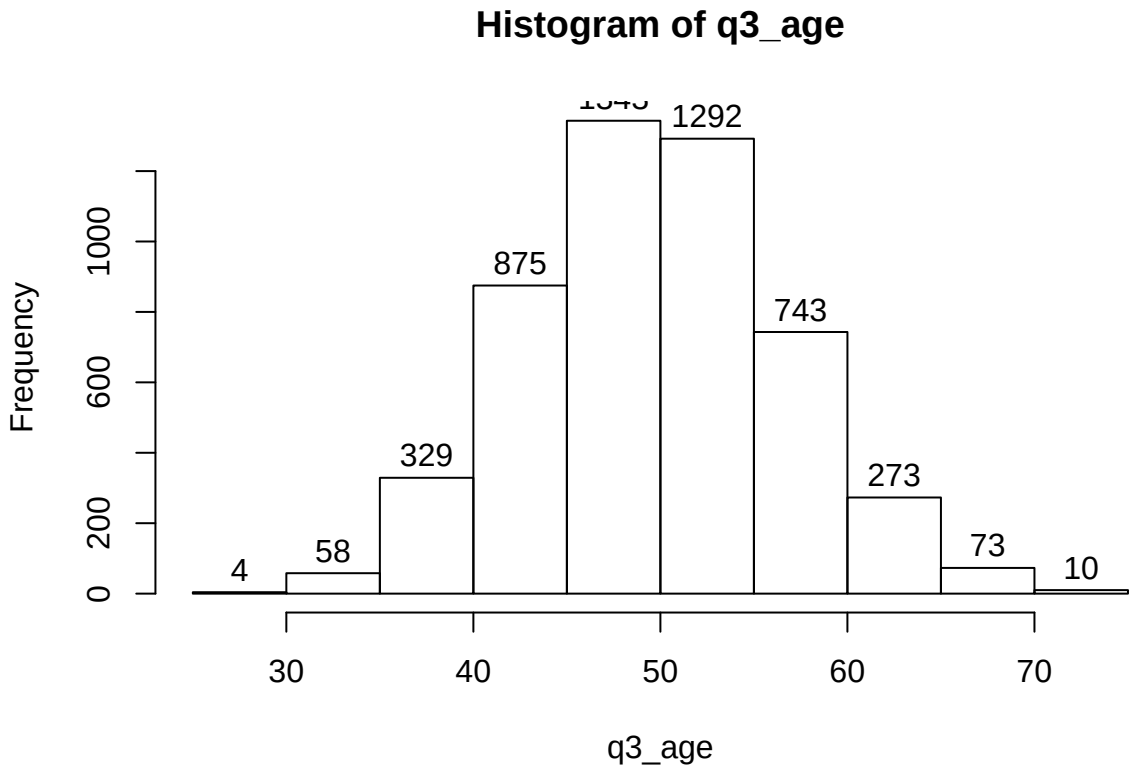
```
q2_gender <- sample(x = c("Male","Female","Other"),
                    size = 5000,
                    prob = c(.48,.51,.01),
                    replace = TRUE)
```

Another useful function for simulating data, particularly for integers, is the `rpois()` function. This generates a Poisson distribution, where `n` specifies the number of random values to return and `lambda` specifies the mean of the distribution (must be non-negative). To get a sense of what you've actually simulated, use the `hist()` function to plot a histogram of the simulated vector; it's also good practice to use `summary()` to review some summary statistics.

As you'll see below, `rpois()` generates a somewhat 'realistic' age distribution that you would expect from an actual dataset:

```
q3_age <- rpois(n = 5000, lambda = 50)

hist(q3_age, labels = TRUE) # Plot histogram, show value labels
```




```
summary(q3_age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    28.0    45.0    50.0    50.2    55.0    75.0
```

All of these newly simulated vectors can then be specified as columns in a tibble / data frame. Note that you don't need to create the simulations **outside** the `tibble()` function; you can always condense your code making all the simulation calls within `tibble()`. For instance, the `ID` column below is generated directly within the `tibble()` function. In some occasions however, you may choose to explicitly declare the vectors to enhance the readability of the code.

The `sim_data` tibble object created below will have 5000 rows and 3 variables, containing an ID, gender, and an age variable. You can use `glimpse()` or `View()` to explore what you have created. Note that because these variables are randomly generated, you will not get the same dataset everytime you run the same code. (`set.seed()` is a trick to overcome this, which you can read more about here)

```
sim_data <- tibble(ID = 1:5000,
                   Q2_GENDER = q2_gender,
                   Q3_AGE = q3_age)
```

Now, let us use the following code to simulate a dataset that represents the consumption of snacks over the three snacking occasions of "pre-lunch" (**PRELUN**), "post-lunch" (**POSLUN**), and "late-afternoon" (**LATAFT**). To make the data slightly more interesting, the probabilities of some variables are pre-specified. The survey question capturing these consumption variables can be:

> Please select all of the following snack types that you have consumed around [INSERT INTERVAL] in the past week.

Note that all the brand consumption variables generated below are in the form of binary variables taking on only the values of 0 and 1, where 0 is assumed to be 'Not Selected' and 1 is assumed to be 'Selected'. Imagine that the snack types 1, 2, and 3 represent Jaffa cakes, Rich Tea biscuits, and 'Posh chocolates' respectively.

```
snack_data <- tibble(ID = 1:5000,
                     Q2_GENDER = q2_gender,
                     Q3_AGE = q3_age,
                     PRELUN_1 = sample(x = c(0,1), 5000, replace = TRUE, prob = c(.45, .55)), # Jaffa
                     PRELUN_2 = sample(x = c(0,1), 5000, replace = TRUE, prob = c(.65, .35)), # Tea Bis
                     PRELUN_3 = sample(x = c(0,1), 5000, replace = TRUE), # Posh chocs

                     POSLUN_1 = sample(x = c(0,1), 5000, replace = TRUE), # Jaffa
                     POSLUN_2 = sample(x = c(0,1), 5000, replace = TRUE), # Tea Biscuits
                     POSLUN_3 = sample(x = c(0,1), 5000, replace = TRUE, prob = c(.45, .55)), # Posh ch

                     LATAFT_1 = sample(x = c(0,1), 5000, replace = TRUE), # Jaffa
                     LATAFT_2 = sample(x = c(0,1), 5000, replace = TRUE, prob = c(.90, .10)), # Tea Bis
                     LATAFT_3 = sample(x = c(0,1), 5000, replace = TRUE)) # Posh chocs
```

Based on the structure created above, we can even create a larger dataset with more binary variables (brand consumption) which will be a closer resemblance to larger, wider data in Usage & Attitude / U&A surveys . The `matrix()` function below creates a 5000 by 30 matrix (specified by the arguments `nrow` and `ncol`), taking on the values of either 0 or 1 generated by the `sample()` function. In other words, we are asking the `sample()` function to generate 15,000 binary values to fill a 5000 x 30 table.

```
matrix(data = sample(x = c(0,1),30 * 5000, replace = TRUE),
       nrow = 5000,
       ncol = 30) %>%
  as_tibble() -> brand_con_sim # Simulated Brand Consumption, 30 columns x 5000 rows

# Give column names to the simulated brand consumption data
names(brand_con_sim) <- c(paste0("PRELUN_",1:10),
                          paste0("POSLUN_",1:10),
                          paste0("LATAFT_",1:10))

# Simulate ID, Gender, and Age variables
# Use cbind() to combine with the simulated brand consumption data

tibble(ID = 1:5000,
       Q2_GENDER = q2_gender,
       Q3_AGE = q3_age) %>%
  cbind(brand_con_sim) -> big_snack_data

glimpse(big_snack_data[,1:15]) # Glimpse first 15 columns only
```

```
## Observations: 5,000
## Variables: 15
## $ ID        <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 1...
## $ Q2_GENDER <chr> "Male", "Male", "Male", "Female", "Female", "Male", ...
## $ Q3_AGE    <int> 49, 56, 54, 54, 46, 44, 64, 52, 53, 44, 57, 52, 55, ...
## $ PRELUN_1  <dbl> 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0...
## $ PRELUN_2  <dbl> 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1...
## $ PRELUN_3  <dbl> 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0...
## $ PRELUN_4  <dbl> 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0...
## $ PRELUN_5  <dbl> 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1...
## $ PRELUN_6  <dbl> 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0...
## $ PRELUN_7  <dbl> 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0...
## $ PRELUN_8  <dbl> 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0...
## $ PRELUN_9  <dbl> 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0...
## $ PRELUN_10 <dbl> 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0...
## $ POSLUN_1  <dbl> 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1...
## $ POSLUN_2  <dbl> 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0...
```

Having simulated a suitable dataset that has a fair resemblance to U&A data, we can look at how for loops and `apply()` functions can come into use.

2. For-loops

The purpose of using for-loops, or any kind of iterative feature (e.g. apply()), is to save the need from explicit coding through controlled repetition. What a for-loop does is to allow a specific piece of code to be executed repeated *for* a specified number of iterations.

Try running the following piece of code: (`paste()` is a function that concatenates strings together to form a single string)

```
for(i in 1:10){ # For i starting from the value 1, all the way to value 10
  two_to_power <- 2^i
  statement <- paste("2 to the power of", i, "is equal to", two_to_power)
  print(statement)
}
```

```
## [1] "2 to the power of 1 is equal to 2"
## [1] "2 to the power of 2 is equal to 4"
## [1] "2 to the power of 3 is equal to 8"
## [1] "2 to the power of 4 is equal to 16"
## [1] "2 to the power of 5 is equal to 32"
## [1] "2 to the power of 6 is equal to 64"
## [1] "2 to the power of 7 is equal to 128"
## [1] "2 to the power of 8 is equal to 256"
## [1] "2 to the power of 9 is equal to 512"
## [1] "2 to the power of 10 is equal to 1024"
```

The `i` variable is simply a specification on the number of iterations that the loop should run, and you can call this something else if you want. It also doesn't need to be limited to integers; you can also iterate through a vector of character strings:

```
glee_team <- c("Belinda Blumenthal","Bella Ridley","Giselle Maarschalkerweerd de Klotz","Hazel")

for(glee_member in glee_team){
  n_char <- nchar(glee_member)
  to_print <- paste0(glee_member,"'s name has ",n_char," characters.")
```

```
  print(to_print)
  }
```

```
## [1] "Belinda Blumenthal's name has 18 characters."
## [1] "Bella Ridley's name has 12 characters."
## [1] "Giselle Maarschalkerweerd de Klotz's name has 34 characters."
## [1] "Hazel's name has 5 characters."
```

As the above demonstrates, for-loops make it very easy to run operations that need to be repeated many times. It also makes it very easy to dynamically adjust your input-outputs, e.g. if you'd wish to add 10 new members to the `glee_team` variable in the previous example.

This can be applied to data analysis operations.

Imagine if you wished to create a gender-split table three times, once for each snack occasion type; you'd have to repeat the code three times:

```
big_snack_data %>%
  group_by(Q2_GENDER) %>%
  summarise_at(vars(num_range("PRELUN_", 1:10)), ~sum(.)/length(.)) # can also use funs() form, dependi
```

```
## # A tibble: 3 x 11
##   Q2_GENDER PRELUN_1 PRELUN_2 PRELUN_3 PRELUN_4 PRELUN_5 PRELUN_6 PRELUN_7
##   <chr>        <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 Female       0.508    0.497    0.502    0.497    0.501    0.505    0.504
## 2 Male         0.525    0.506    0.502    0.484    0.503    0.498    0.508
## 3 Other        0.66     0.46     0.52     0.46     0.54     0.54     0.62
## # ... with 3 more variables: PRELUN_8 <dbl>, PRELUN_9 <dbl>,
## #   PRELUN_10 <dbl>
```

```
big_snack_data %>%
  group_by(Q2_GENDER) %>%
  summarise_at(vars(num_range("POSLUN_", 1:10)), ~sum(.)/length(.))
```

```
## # A tibble: 3 x 11
##   Q2_GENDER POSLUN_1 POSLUN_2 POSLUN_3 POSLUN_4 POSLUN_5 POSLUN_6 POSLUN_7
##   <chr>        <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 Female       0.501    0.480    0.480    0.485    0.502    0.512    0.5
## 2 Male         0.494    0.507    0.495    0.497    0.500    0.492    0.476
## 3 Other        0.42     0.56     0.48     0.54     0.56     0.48     0.48
## # ... with 3 more variables: POSLUN_8 <dbl>, POSLUN_9 <dbl>,
## #   POSLUN_10 <dbl>
```

```
big_snack_data %>%
  group_by(Q2_GENDER) %>%
  summarise_at(vars(num_range("LATAFT_", 1:10)), ~sum(.)/length(.))
```

```
## # A tibble: 3 x 11
##   Q2_GENDER LATAFT_1 LATAFT_2 LATAFT_3 LATAFT_4 LATAFT_5 LATAFT_6 LATAFT_7
##   <chr>        <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 Female       0.506    0.508    0.492      0.5    0.502    0.512    0.489
## 2 Male         0.492    0.509    0.513      0.5    0.513    0.509    0.495
## 3 Other        0.48     0.54     0.46       0.6    0.54     0.5      0.48
## # ... with 3 more variables: LATAFT_8 <dbl>, LATAFT_9 <dbl>,
## #   LATAFT_10 <dbl>
```

The example below shows how the identical output can be replicated using a for-loop. The iteration is done through each member of the `vars_string` vector, which has been created to contain the variable headers

for each snack-time occasion. `summarise_at()` allows you to summarise all the 10 brand variables for each snack-time occasion at the same time, so effectively the loop only iterates 3 times for 30 columns of data.

As you can see, code using for-loops are more succinct, elegant, and allows less room for human error, such as forgetting to change all the code chunks when changing the grouping variable. In order to make the above code practical, we will also need to find a way to assign the outputs to something that we can manipulate, instead of just printing them out in the console. An easy way to do this is to initialise an empty list object, and then assign each output as the n-th member of the list.

In the example below, an alternative way of iteration is used, where the iteration is run through the numbers 1 to 3, as opposed to each member in the `vars_string` vector. The benefit of iterating this way is to make it easier to assign the outputs to their corresponding 'slots' in the list.

```
vars_string <- c("PRELUN_","POSLUN_","LATAFT_") # Variable characters to iterate

brandcon_list <- list() # Initialise empty list

for(i in 1:length(vars_string)){ # length(vars_string) is 3 in this case
  big_snack_data %>%
    group_by(Q2_GENDER) %>%
    summarise_at(vars(num_range(vars_string[[i]], 1:10)), ~sum(.)/length(.)) %>%
    data.frame() -> brandcon_list[[i]]
}

brandcon_list
```

```
## [[1]]
##   Q2_GENDER  PRELUN_1  PRELUN_2  PRELUN_3  PRELUN_4  PRELUN_5  PRELUN_6
## 1    Female 0.5081712 0.4968872 0.5015564 0.4972763 0.5011673 0.5046693
## 2      Male 0.5252101 0.5063025 0.5021008 0.4840336 0.5033613 0.4978992
## 3     Other 0.6600000 0.4600000 0.5200000 0.4600000 0.5400000 0.5400000
##    PRELUN_7  PRELUN_8  PRELUN_9 PRELUN_10
## 1 0.5038911 0.4988327 0.5233463 0.5050584
## 2 0.5075630 0.4915966 0.4962185 0.5012605
## 3 0.6200000 0.5200000 0.5400000 0.5000000
##
## [[2]]
##   Q2_GENDER  POSLUN_1  POSLUN_2  POSLUN_3  POSLUN_4  POSLUN_5  POSLUN_6
## 1    Female 0.5007782 0.4801556 0.4797665 0.4848249 0.5019455 0.5116732
## 2      Male 0.4941176 0.5071429 0.4953782 0.4974790 0.5004202 0.4920168
## 3     Other 0.4200000 0.5600000 0.4800000 0.5400000 0.5600000 0.4800000
##    POSLUN_7  POSLUN_8  POSLUN_9 POSLUN_10
## 1 0.5000000 0.5035019 0.5046693 0.5023346
## 2 0.4760504 0.4945378 0.5264706 0.4894958
## 3 0.4800000 0.5400000 0.5600000 0.5200000
##
## [[3]]
##   Q2_GENDER  LATAFT_1  LATAFT_2  LATAFT_3 LATAFT_4  LATAFT_5  LATAFT_6
## 1    Female 0.5062257 0.5077821 0.4918288      0.5 0.5019455 0.5124514
## 2      Male 0.4915966 0.5088235 0.5134454      0.5 0.5134454 0.5092437
## 3     Other 0.4800000 0.5400000 0.4600000      0.6 0.5400000 0.5000000
##    LATAFT_7  LATAFT_8  LATAFT_9 LATAFT_10
## 1 0.4894942 0.4988327 0.4922179 0.5003891
## 2 0.4945378 0.4869748 0.5075630 0.4941176
## 3 0.4800000 0.4400000 0.5800000 0.4600000
```

After the outputs have been assigned to a list, each individual member of a list can be called by referencing the index in square brackets, for instance `brandcon_list[[2]]`. A list is perhaps the most suitable object for storing these outputs, not only because this would avoid cluttering up your environment by explicitly declaring the outputs as objects, but also because data frames in lists can be quite handily exported as Excel sheets in a single Workbook using the `write_xlsx()` in the `writexl` package (see here)

Below is a sample of the same for-loop used in iterative plotting. Try copying and running this and see what you can figure out what the code does! (don't worry if you don't understand the ggplot syntax at this point)

3. apply() family of functions

Although for-loops are generally easy to read and are fairly intuitive, R possesses some special tools for iterative repetition which allow largely similar operations to be performed more efficiently due to its vectorisation features (If you're curious, read more here). The standard example of such a 'tool' is the `apply()` function, but since there are multiple variations of this (e.g. `lapply()`,`sapply()`,`mapply()`) with the main difference being the type of output they return, these functions are often referred to as the "apply() family" in the literature (e.g. for instance, see here).

The function that aligns most well with we are attempting to achieve in our example (i.e. summarise some data by group, then iterate through sets of variables, then return the outputs in a list) is `lapply()`. In essence, `lapply()` allows you to iterate some process (e.g. analysis, pasting text) and return the outputs in a list.`lapply()` has two main arguments:

- x: a vector or a list to iterate through.
- FUN: the function which you want x to pass as the argument.

Unlike the for-loop example in the previous section, the "process" to iterate is expressed by a function, which is labelled as "some_function" below. This function here takes `x` as the argument, calculates its square, and then returns a sentence that tells you the result of the calculation. What `lapply()` does here is essentially to apply the custom-defined function to "some_numbers":

```
some_numbers <- c(57,89,40,16)

some_function <- function(x){
  square_x <- x^2 # Square x
  paste(square_x,"is the square of",x)
}

lapply(some_numbers,some_function)
```

```
## [[1]]
## [1] "3249 is the square of 57"
##
## [[2]]
## [1] "7921 is the square of 89"
##
## [[3]]
## [1] "1600 is the square of 40"
##
## [[4]]
## [1] "256 is the square of 16"
```

You may also try to run `lapply(1:10,sqrt)`, which applies the base R square root function (`sqrt()`) to the numbers 1 to 10. How many members of that list have a value that is greater than 2.5? (Answers below)

To fully take advantage of the power of `lapply()`, you will need to learn how to write your own functions. Put simply, a function is something takes an input (the "argument(s)") and converts it into an output (there

are exceptions, but most functions have inputs and outputs). You will already know many R functions at this point, such as `mean()`, `sum()`,`sqrt()`, `nchar()`.

Creating your own function is simple. You will first need to give it a name, decide what you want as inputs and what you want it to do and return as outputs. Here's one that lets you input "some_word" as an argument, calculates the total number of characters in that word using the function `nchar()`, and returns a string of text.

```
count_characters <- function(some_word){
 total_char <- nchar(some_word)
 paste(some_word, "has", total_char,"characters.")
}

# count_characters("schadenfreude")
# [1] "schadenfreude has 13 characters."
```

Here's a silly one - no arguments are taken, but if you run `quick_maths()` it prints a succession of statements to the console, whilst pausing one second between each statement. This isn't really useful in real life, but gives you a sense of how a function works.

```
quick_maths <- function(){
  print("two plus two is four")
  Sys.sleep(1)
  print("minus one that's three")
  Sys.sleep(1)
  print("QUICK MATHS")
}

## RUN THIS:
quick_maths()
```

```
## [1] "two plus two is four"
## [1] "minus one that's three"
## [1] "QUICK MATHS"
```

A function can also take in multiple arguments. Here are two examples, one calculating index and the other calculating CAGR (Compound Annual Growth Rate):

```
index_it <- function(x, total){
  return(x / total * 100)
}

# index_it(.87,.43)
# [1] 202.3256


calculate_CAGR <- function(end_value, start_value, periods){

  step_1 <- end_value / start_value
  step_2 <- step_1 ^ (1 / periods)
  step_3 <- step_2 - 1
  return(step_3)
}

# calculate_CAGR(end_value = 5000,
#                start_value = 3200,
#                periods = 3)
```

```
# [1] 0.1603972
```

Answers

1. `sum(lapply(1:10,sqrt) > 2.5)` tells you how many members in the `lapply()` result is larger than 2.5.