LiteRacy Quiz

Alex Slavenko & Evatt Chrigwin

2023-08-17

Contents

This is a simple quiz designed to test your literacy in the R programming language, and teach you some basic skills if you're not really familiar with R. After completing this quiz you should have R and RStudio installed on your computer, and have basic understanding of data structures, annotation, and simple functions in R. Once you have successfully completed the quiz, you will be ready to take part in the Cesar R workshop.

The quiz shouldn't take you any longer than 30 minutes, and likely a lot less than that if you already have basic familiarity with R. If you have any questions or need help, don't hesitate to ask either of us (Alex or Evatt) for help.

Installation

The first step is to make sure you have R installed on your computer. If you already have R version 4 installed (any R version above 4.0.0), feel free to skip directly to the start of the quiz.

You can download the latest version of R from https://cloud.r-project.org. Choose the appropriate installer based on your operating system (macOS or Windows) and run it.

You will now have the programming language **R** installed on your computer. Well done! However, you will notice if you try to open it that you only have a single window open, which will probably look different from how you've seen R on your colleague's computers. This is because R doesn't have a graphical interface - that's where **RStudio** comes in. You can download the latest version of RStudio from https://posit.co/download/rstudio-desktop.

RStudio is an integrated development environment (IDE), which will allow you to use R with ease. When you open RStudio, you will notice three panes:

- On the top right you will see the Environment pane. This shows you which objects currently exist in your R environment, and should be empty for now. This will become populated as we go through the quiz. You can go ahead and ignore it for now.
- On the bottom right you will see the Files pane. This shows you your current working directory and which files are currently there. This is a lot like a file explorer on your computer. In this pane you'll also see generated plots (we'll get there), and documentation for R packages and functions (we'll get there too).
- On the left, you will see the Console pane. This is similar to what you see when you just open R without RStudio. This is where you actually run R code.

Copy the following bit of code and paste it into your console. Press *Enter* to run it.

```
2 + 3
```

Do you see the result of the calculation? You'll notice that while this is easy to use, you do have to go one line at a time. This can be difficult.

Now, click on File -> New File -> R Script. This will open a new, blank pane on the top left. In this pane you can view scripts, which are saved in the .R file format. Here you can write lines

of code, and then either run them one by one (by pressing *Ctrl+Enter*) or the entire script (by pressing *Ctrl+Shift+Enter*).

Copy the following bit of code into your script. Press *Ctrl+Shift+Enter* to run the entire script.

```
print("the answer to 2 + 3 is")
2 + 3
```

You will see the output of the script showed up in the console, where R is run. So, you can write your code in the script pane, allowing you to write many lines of code, keep track of them, try them out, and change them. And then you run them and see the output show up below in the console.

QUIZ BEGINS HERE!

Throughout this section, you will have several tasks to complete. If you are having trouble with a task, there will be clues and hints *in italics*. You can always see the documentation for a function you're unfamiliar with by switching the bottom right pane to the Help tab and searching for the function. Some tasks will also be followed by long-winded explanations. You can skip these if you managed to solve the task.

If you're still having difficulty, you can ask one of us (Alex or Evatt) for help.

There are many, **many** things you can do in R as is, or as we call it, base R. But there are a lot of more advanced functions that have been built by the wider R community. Functions can be stored in a **package**, and packages can be downloaded and installed in your own personal R library. A lot of packages which have been extensively tested and validated can be downloaded from CRAN - the Comprehensive R Archive Network.

Task 1.

Do you have the following packages installed: `tidyr`, `dplyr`, `magrittr`, `readr`, `ggplot2`?

If the answer is no, go ahead and install them. You won't need most of them for the quiz, but you will for the workshop!

Hint: you can use the 'installed.packages()' function to see which packages are installed on your system. The latest versions of packages can be installed from CRAN using the 'install.packages()' function

Task 2.

Load the `magrittr` package

Hint: use the 'library()' function

Task 3.

Import the example dataset (data.csv) and store it as an object named data

Hint: use the 'read.csv()' function, and use <- to store an object

Hint: are you looking in the right directory?

Explanation:

Like other programming languages, R needs to be told where files are located. If a full pathway to the file is not provided, then R will search for the file in its current working directory.

There are two ways in which you can find your working directory. First, you can use the the `getwd()` function, which prints out the pathway to the working directory to help you figure out where R is searching for the file. Second, you can look in the console pane (bottom right) in RStudio - at the top of the console pane there is a bar that shows you the working directory.

Note that the working directory isn't only where R will look for files you're trying to import, but also where it will put files you are trying to export. So you may need to provide full pathways whenever you are trying to import or export a file.

Alternatively, you can change the working directory using `setwd()`. This function allows you to set the working directory for your R session, and avoid having to provide full pathways. However, it is important to note two things:

1. Pathways set using `setwd()` are specific to the computer (and user). Therefore, if you share your code with someone else, they will almost certainly have to change the pathway written in `setwd()`.
2. `setwd()` only sets the working directory for **the current session**. This means that if you restart R, you will need to run the function again.

Finally, R has environments, used to store objects and to search for stored objects. We won't be getting into the intricacies of environments, because it's pretty complicated. The important thing to know is that you can store objects, such as imported files or newly generated objects, into your global environment. You can see what is stored in the global environment in the top right pane of RStudio.

So, if you just run `read.csv("data.csv")` the table will be printed in the console, but it won't be saved anywhere. You will have to import the table every time you want to do anything with it, which is not only tedious, but also impossible for many things we would want to do. But if you store it in the global environment using `data <- read.csv("data.csv")`, you can then access it and run other functions on it! You should now be able to see the object `data` in the Environment pane.

Task 4.

Find the mean of the second column

Did that work? Why not?

Hint: you can find the mean using the 'mean()' function

Hint: try to use the 'str()' function

Explanation:

R objects can generally come in three main varieties: vectors, matrices, and data.frames (we'll ignore the more complicated ones like arrays and lists for now).

Vectors have one dimension. They can be single value, or a series of values. For instance, `c(1)` is a vector, but so is `c(2, 4, 6)`. The differences is only in their length.

You can extract any single element of a vector with the following indexing function: `vec[i]` which extracts the *i*th element of the vector *vec*.

Matrices and data.frames both have two dimensions.

For instance, copy the following code into your console and press *Enter*:

```
matrix(c(1,2,3,4,5,6), ncol = 2)
```

You've just created a matrix! We've put in six numbers, and asked to generate a matrix with 2 columns. Therefore, there are 3 rows. What happens if you try to run the following code?

```
matrix(c(1,2,3,4,5,6,7), ncol = 2)
```

All columns of matrix (and a data.frame) must have the same length, as do all rows. In this chunk of code we tried to create a matrix with 2 columns from 7 numbers - and 7 is not divisible by two. The number of rows was then rounded up to 4. But 2*4 is 8, and our vector has only 7 elements. So what happened here?

The input for the `matrix()` function is a vector (note a vector is always defined using `c()`). By default, the dimensions of the matrix (how many rows and how many columns) are defined by the optional arguments `nrow` and `ncol`, and then the matrix is populated with the elements of the input vector by columns - from top to bottom, and then from left to right. Once our matrix reached its eighth element (column 2, row 4) it no longer had values from the input matrix to plug in - so it started over from the top. Hence, the bottom right element of the matrix is 1, the first element of our vector.

No what happens if you run the following code?

```
matrix(c(1,2,3,4,5,6))
```

This looks like a vector, but is actually a matrix. This is because the default value for `ncol` is 1 - meaning by default, there is only a single column, and as many rows as there are elements in the input vector. This ties in to an important feature of matrices - they are essentially combinations of vectors. You can extract each column or row of a matrix as a vector.

Run the following code:

```
mat <- matrix(c(1,2,3,4,5,6), ncol = 2)

mat[,2]
```

The output you have just printed is a vector, which is also the second column of the matrix. The function `mat[i,]` extracts the *i*th row of matrix *mat*, and the function `mat[,k]` extracts the *k*th column of matrix *mat*. Try it yourself.

What happens if you run `mat[3,2]`? You've just extracted the element of the matrix in row 3 and column 2 - which is the same as extracting the second element of the third row (`mat[3,][2]`), or the third element of the second column (`mat[,2][3]`).

A data.frame is similar to a matrix in being two dimensional, but with two key differences.

First, run the following code:

```
df <- data.frame(c(1,2,3,4,5,6), ncol = 2)

df
```

You'll notice you still have two columns, but otherwise it is very different to the matrix you previously generated. That is because a data.frame is created very differently from a matrix - the function `data.frame()` accepts vectors as input. Each vector is a separate column, and all are coerced to be the same length (like in matrices, all columns must be of the same length). Columns in data.frames are also named, and if a name is not provided explicitly, it is generated from the vector.

So what happened here? We created a data.frame, made up of the vectors `c(1,2,3,4,5,6)` and `c(2)`. The second one we named `ncol` by writing `ncol = 2`, whereas the first one we did not name (hence the weird name R generated). Secondly, `ncol` only had one element provided - so R automatically duplicated it six times to match the length of the first column.

Data.frames can be indexed like matrices, but we can also make use of the fact that columns are named by using `$`. So, if we want to extract the second column of the data.frame, we do this either by indexing its position (`df[,2]`) or its name (`df$ncol`).

Now, run this code:

```
mat <- matrix(c(1,2,3,"a","b","c"), ncol = 2)

df <- data.frame(col1 = c(1,2,3), col2 = c("a","b","c"))

mat

df
```

While these superficially seem similar, they are in fact different in another key aspect that differentiates matrices and data.frames. See how the elements in `mat` have quotation marks around

them whereas the ones in `df` don't? That's because all elements in a matrix (or vector) must be of the same type. In data.frames, each column can be of a different type.

What are data types (or classes)? There are several data types in R. The most basic ones, and which we will use the most, are numeric (1, 2.5, 77, etc.), character (blah, ok, "wow how cool", etc.), and logical (TRUE, FALSE). Quotation marks are used to show that something is a character - this can work for numerics that are coerced to be a character ("1" vs 1), or for longer strings that include spaces (like "wow how cool", which can be a single element of vector, as opposed to wow how cool which will cause an error - R really doesn't like spaces). Numerics can be coerced into characters, characters can't be coerced into numerics. So, since we supplied a mixed vector (some numbers, some characters) to `mat`, every element there was coerced into a character. We can see this by running the following code:

```
str(mat)
```

```
##  chr [1:3, 1:2] "1" "2" "3" "a" "b" "c"
```

This shows us that `mat` is a character matrix with three rows and two columns. Compare this to:

```
str(df)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ col1: num  1 2 3
##  $ col2: chr  "a" "b" "c"
```

And we can see that `df` is made up of a single numeric column, and a single character column.

We can also coerce vectors manually. For example, we can coerce the first column of `df` into a character, like in `mat`, by running `as.character(df$col1)`. We can also coerce the first column of `mat` into a numeric by running `as.numeric(mat[,1])`. Note that while we can make this change permanent in `df` by assigning the coerced values back to the first column (`df$col1 <- as.character(df$col1)`), trying to do the same thing with `mat` (`mat[,1] <- as.numeric(mat[,1])`) won't work, because matrices all elements of a matrix must be of the same type.

So, to bring this all back to our original question - why couldn't we get the mean of the second column of `data`? Try to run the following code:

```
str(data)
```

As you can see, the second column (column_b) is a character vector, and the function `mean()` only works on numeric vectors.

Task 5.

Coerce the second column into a numeric and find its mean

Did that work? Why not?

Hint: use the functions 'is.na()' and 'which()'

Hint: can the argument 'na.rm' help you find the mean?

Explanation:

Look closely at `data$column_b`. You'll notice that the 11th element of that column is `?`. That's the reason this column was a character in the first place (R automatically assigns data types to columns when importing a data.frame), causing the `mean()` function to fail originally. But what happens when we coerce the column into a numeric?

```
as.numeric(data$column_b)
```

```
## Warning: NAs introduced by coercion
```

```
##  [1]  5 97 23 11 46 89 23 55 41 82 NA 63  1
```

`?` cannot be coerced into a number, so it was turned into `NA`, which is how R designates missing values.

By default, `mean()` returns `NA` if the input includes any missing value. Luckily, we can set the `na.rm` argument to `TRUE` to tell the function to remove all missing values from the input before calculating the mean. Therefore, if you run the following code, you should get the correct mean for the column:

```
mean(as.numeric(data$column_b), na.rm = TRUE)
```

Task 6.

Run the following code:

```
mean(data$column c)
```

Did that work? Why not? Can you fix this code so it runs properly?

Hint: use 'names()' to see the column names

Explanation:

Remember, R **really** doesn't like spaces. Our original csv file had the column named column c - which R automatically changed into column.c. Follow iSOP #001 (Dataset Construction) for naming conventions for columns to avoid issues like this.

Task 7.

Run the following code:

```
this is code to calculate the mean

mean(data$column.c)
```

Did that work? Why not? Can you fix this code so it runs properly?

Hint: just like in Instagram, use hashtags

Explanation:

Sometimes, you will want to add comments or annotation to your code. Especially if you're writing long scripts, comments can help you keep track of what you're doing and why. This is especially useful when you're working with others, since sharing well-annotated code makes it easier for collaborators to understand how the code works. And remember - your main collaborator is future you, and future you will really appreciate it if you annotate your code properly now.

Placing # before a line tells R to ignore it. R will still print it, but it won't treat it as code and will not attempt to run it.

That's why this chunk:

```
this is code to calculate the mean
```

throws up an error (because there are spaces, and none of these words are recognised functions or objects), whereas this chunk:

```
# this is code to calculate the mean
```

Does no such thing.

CONGRATULATIONS!

If you've reached this far, you have successfully completed the Cesar R literacy quiz! This means you are now ready to participate in the Cesar R workshop. Until then, time to celebRate!