Chapter 3: Part 1, The Basics of Programming (in R)

Tudor Cristea

Table of Contents

# Main Objective for Chapter 3

When I started my PhD, I had no experience in programming or working with educational data. Over the course of my PhD, I learned these skills and this chapter aims to help others in a similar position. The target audience for this chapter includes researchers working in educational fields, particularly in Learning Analytics (LA), who have little or no experience with student trace data.

Beginners can often provide better tutorials and explanations on a subject compared to experts. This is because beginners can offer fresh perspectives, use relatable language, and provide simplified explanations that resonate with other novices. They also tend to remember the challenges they faced while learning, which helps them anticipate and address common obstacles. Additionally, beginners are less likely to overlook basic concepts that experts might take for granted.

In this chapter, I will present all the steps I followed in small, digestible chunks, supplemented with dataset examples and code snippets. I will use data collected from students using a specific Learning Management System (LMS), Canvas, and employ R software for the analysis. However, the concepts and explanations can be applied to other LMSs or programming languages. While programming is not the main focus of this tutorial, it is necessary. Part 1 focuses more on understanding the basics of R (and programming in general) and can be skipped by experienced programmers; for this purpose it is not included in my dissertation. Part 2 focuses on cleaning and pre-processing the trace data in order to make it ready for the final analysis, thus, it is included in my dissertation.

This chapter reflects my personal journey, from the first time I opened an educational dataset to the publication of scientific papers based on it. If you are seeking specific analyses and expert explanations, I highly recommend this source, which I consider the best guide to Learning Analytics through R available today (<https://lamethods.github.io/>).

Another purpose of this chapter is to introduce the methodology employed in the subsequent two chapters, Chapter 4 and Chapter 5. Although each chapter includes its own methodology section, this chapter provides a comprehensive overview of the steps taken before reaching that point, detailing my personal process and understanding.

# Getting Started

First off, you need to get the R Software from here (<https://www.r-project.org/>). Nowadays, most people also use RStudio, which you can get here (<https://www.rstudio.com/categories/rstudio-ide/>). RStudio is an integrated development environment (IDE) for the R programming language. It provides a user-friendly interface for writing, executing, and debugging R code. Get them in this order. Looking directly at RStudio you can see the:

* **Script Editor (top left)**: Write and save your R code here.
* **Console (bottom left)**: Execute your R code here.
* **Environment Pane (top right)**: View all objects in memory, such as vectors and datasets.
* **Support Pane (bottom right)**: Access plots, files, and packages.

## General Framework for Problem-Solving

In R, as in most programming languages, you will frequently use functions. These functions are often grouped and shared through packages. Most of your work will involve using the functionality provided by various packages, written and shared by others for free. Below you can see a general framework for solving problems when programming.

1. **Define the task**: What do you want to achieve? Write it down in natural language.
2. **Break down the task into natural language steps**: Identify the steps needed to accomplish it. Think about the logical sequence of actions and write them down as clearly as possible using human language.
3. **Find relevant functions and packages**: Search for existing functions and packages that can accomplish each step. Use resources like R documentation, online forums, and communities.
4. **Write and test code**: Start writing your code in small, testable chunks. Test each part as you go to ensure it works as expected.
5. **Debug and refine**: Check for errors and unexpected behavior. Use debugging tools and techniques to refine your code.
6. **Document your code**: Add comments and documentation to explain your code by adding **#** before the text; this ensure the code following it will not be executed. Try to leave as many comments as possible, such that a stranger can easily understand what you are doing at any time. This helps others understand your work and can be useful for your own future reference.
7. **Review and Share**: Review your entire process and results to ensure accuracy and completeness. Prepare your analysis for sharing, whether through reports, visualizations, or publishing your code and findings. Use tools like R Markdown for dynamic reports or GitHub for code sharing.

Over time, you will learn how to improve each step individually, but also how to better integrate them. As you gain experience, thinking about and solving problems with R will become second nature. Finally, it is important to understand that these are not hard steps, you will go through them again and again in order to adjust and add new things. Let’s try and go through the steps with a very easy task in R.

## Applying the Framework

### Calculate the average of five randomly generated numbers

1. **Define the task**: Calculate the average of five randomly generated numbers.
2. **Break down the task into natural language steps**:
   * Generate five random numbers.
   * Sum them.
   * Divide by five.

What is the range of numbers that we can choose from? Should the numbers be integers only? While assumptions might be clear between humans, in code, you need to be very explicit. In our case, we want five integer numbers within the 1-100 range, with repetitions allowed.

1. **Find relevant functions and packages**:

A significant part of data science involves knowing where to find information. For simple tasks, online searches can be very helpful. For example, searching for “generating random numbers in R” or “average in R” can provide quick answers. Remember to add “R” to your search queries, as many results might otherwise refer to Python. For more complex problems, communities such as Stack Overflow (<https://stackoverflow.com/>) are invaluable. Additionally, natural language models like ChatGPT (<https://chatgpt.com/>) or specialized programming tools like GitHub Copilot (<https://github.com/features/copilot>) can assist you by understanding your problem in human language and providing code solutions and explanations.

* In order to generate random numbers we can use the sample() function. We can specify the range (1-100), how many numbers to select (5), and if we can repeat the same number (“replace= TRUE”).

sample(1:100, 5, replace = TRUE)

## [1] 30 31 20 76 29

In programming, it is essential to save the generated values so you can reuse them later. In R, we use the “<-” operator for assignment, which assigns a value to a variable. Simply running the sample function will generate the numbers and display them, but they won’t be saved. Considering that the function “generates” random numbers, these could be different every time. If we assign the numbers to a variable (“random\_integers”), we can use them later. However, in this case, the code only saves the numbers. You can simply write the name of the variable or use print() to actually see the values. You can also check them in the upper right corner of R Studio, in the environment pane.

#Save the numbers  
random\_integers <- sample(1:100, 5, replace = TRUE)   
  
#See the numbers  
random\_integers

## [1] 45 48 76 98 23

* Next, we can use the sum() function to add up all the generated numbers.

#Save the sum  
random\_integers\_sum <- sum(random\_integers)   
  
#See the sum  
random\_integers\_sum

## [1] 290

* We can divide the sum by 5 in order to get the average

#Divide by 5 and see the final number  
random\_integers\_average <- random\_integers\_sum/5   
random\_integers\_average

## [1] 58

As you can imagine, there is already a function that calculates averages, mean(), which we can use instead:

#Use the mean() function instead  
random\_integers\_mean <- mean(random\_integers)   
random\_integers\_mean

## [1] 58

As you can see, the final steps we talked about earlier (writing and testing the code, debugging, adding comments, and reviewing) are not explicitly shown. This is because the programming process is a series of interconnected steps that are not completely separate but rather iterative and often overlapping; we have already used them at each point in order to successfully achieve our goal.

### Function arguments

In the context of functions, there are two important types of arguments: mandatory and default.

* **Mandatory arguments** are parameters that must be provided when calling a function for it to execute correctly. For instance, in the sample() function, specifying the number of values (5) to be selected is a mandatory argument as the function can not work without it.
* **Default arguments** are values assigned to parameters within a function’s definition. When the function is called, if a value for that parameter is not explicitly provided, the default value is used. Typically, these default values are set by the creator of the function. For instance, in the sample() function, the default behavior is replace = FALSE, meaning that values are sampled without replacement. Since we wanted the function to accept repetitions, we had to change it to replace = TRUE.

Before using a function, it’s essential to refer to the R Documentation to understand its parameters and usage. This documentation can be accessed easily in the R console using the question mark followed by the function name (?mean), or by searching for the function online. This ensures that you’re utilizing the function correctly and leveraging its capabilities effectively.

## Data Types

In R, data is categorized into different types, each suited for specific kinds of analysis and manipulation. The primary types include:

* **Numeric** for numbers
* **Character** for text
* **Logical** for TRUE/FALSE values
* **Factor** for categorical data

These types can be organized into various data structures, allowing for efficient data handling and processing. Below you can see some examples of the main data structures in Learning Analytics.

## Data Structures

**Vectors**: Vectors are the most basic data structures in R, consisting of a sequence of elements of the same type, such as numeric, character, or logical, typically representing one column of data. In Learning Analytics, for instance, you might have a vector containing all the datastamps or one containing all the student IDs. When these vectors are combined, they form a data table or data frame, which is made up of multiple vectors of different types. They are created using the c() function. For example:

c(1, 2, 3, 4) #shows a numeric vector

## [1] 1 2 3 4

**Data Frames**: Data frames are two-dimensional, heterogeneous data structures, where each column can be of a different type. They are similar to tables in databases and are commonly used for storing datasets. You can create a data frame using the data.frame() function. For example,:

data.frame(name = c("John", "Jane"), age = c(25, 30), scores = c(85, 90)) #a dataframe with columns for names, ages, and scores

## name age scores  
## 1 John 25 85  
## 2 Jane 30 90

**Lists**: Lists are flexible, heterogeneous data structures that can contain elements of different types and lengths. They are created using the list() function and can hold a variety of objects, such as vectors, data frames, and even other lists. For example:

list(name = "John", age = 25, scores = c(85, 90, 92)) #list with a character element, a numeric element, and a numeric vector

## $name  
## [1] "John"  
##   
## $age  
## [1] 25  
##   
## $scores  
## [1] 85 90 92

These data structures are fundamental to data manipulation and analysis in R, providing versatile tools for handling diverse data types and formats.

## Checking for Data Types: **class(), is.x()**

To check the type or structure of data you can use the class() function, or use is.x() functions (e.g., is.numeric(), is.characteThis preparation ensures that the data is ready for the more advanced analyses presented in the subsequent chapters.This preparation ensures that the data is ready for the more advanced analyses presented in the subsequent chapters.This preparation ensures that the data is ready for the more advanced analyses presented in the subsequent chapters.r(), is.vector()) to verify if an element is a specific type. Many coding problems arise when functions do not operate correctly on a given data structure, so it’s essential to check data types early in the debugging process.

#Check the type of one value  
class(3) #number 3 is numeric

## [1] "numeric"

class("3") #adding quotation marks make it a character type

## [1] "character"

#Use the class() function  
class(random\_integers\_mean)

## [1] "numeric"

#Check if the element is a vector and what type of vector  
is.vector(random\_integers\_mean)

## [1] TRUE

is.numeric(random\_integers\_mean)

## [1] TRUE

## Packages

Finally, we need to talk about packages. When looking up the documentation for the mean() function (?mean), you will see in the top left “mean {base}”. This shows that the mean function is part of the base package, which comes with R. However, many times you will need other specific functions that are part of other packages. For this, when you find the function you want to use, you first need to install the package. After this, you need to load it.

For example, I want to use the ggplot function which is great for data visualization.

#Install the "ggplot2" package. We will not run this command now since it might cause some problems  
install.packages("ggplot2")  
  
#Load the package. Now we can use all functions within the package  
library(ggplot2)

You only need to install the package once, however, you need to load it every time you open R and use it. It is best practice to not load all the packages but only those you use. Try to only load the packages that are used within the script (usually at the beginning of the script).

# Importing Data

Once you’ve obtained the necessary software and data (for which I can’t provide assistance), the next step is to import/load the dataset into your R environment, which is displayed in the top right pane. Fortunately, RStudio simplifies this process, particularly for beginners. It’s crucial to pay attention to the file extension of your dataset. To import it, navigate to “RStudio” -> “File” -> “Import Dataset” and select the appropriate extension corresponding to your dataset. Every time you interact with the interface, keep an eye on the console. It displays the code executed behind the interface, offering valuable insights. Over time, you will learn to avoid the user interface as it can be restrictive, and use the console only.

Alternatively, we can employ functions such as read.excel() to load datasets into memory. However, when dealing with numerous datasets, such as in the case of big data, individually importing each one can be cumbersome. Therefore, a more efficient approach involves saving the folder path as a variable (in the form of a character string) and referencing it as needed. Firstly, let’s examine the default path that R uses.

### Importing data using the default path

#This means "get working directory"  
getwd()

## [1] "C:/Users/Me/OneDrive/Papers/Methodology/Guides and data for the Method chapter"

#Show the files present in the working directory  
list.files()

## [1] "Chapter-3,-Part-1,-The-Basics.docx" "Chapter-3,-Part-1,-The-Basics.html"  
## [3] "Chapter-3,-Part-1,-The-Basics.Rmd" "Chapter-3,-Part-2,-Advanced.docx"   
## [5] "Chapter-3,-Part-2,-Advanced.html" "Chapter 3, Part 1, The Basics.Rmd"   
## [7] "Chapter 3, Part 2, Advanced.Rmd" "example\_assignments.xlsx"   
## [9] "example\_courses.xlsx" "example\_discussions.xlsx"   
## [11] "example\_files.xlsx" "example\_parquet.snappy.parquet"   
## [13] "example\_quizzes.xlsx" "example\_requests.xlsx"   
## [15] "final\_requests.csv" "req\_crs\_disc\_assgn\_quiz.csv"

If the dataset is in the same directory as the default path, we can load it using the dataset name alone:

#Load the package with the read\_excel() function  
library(readxl)   
  
#Import the data just by using the dataset name  
requests <- read\_excel("example\_requests.xlsx")   
  
#Print the first few rows of the data to confirm it's loaded  
head(requests)

## # A tibble: 6 × 9  
## student\_id course\_id timestamp url assignment\_id quiz\_id discussion\_id  
## <chr> <dbl> <chr> <chr> <dbl> <dbl> <dbl>  
## 1 A 10456 2024-03-20 13:… /cou… 476 NA NA  
## 2 A 10456 2024-03-20 13:… /cou… 476 NA NA  
## 3 B 10456 2024-03-25 08:… /api… NA 798 NA  
## 4 C 10987 2024-07-30 08:… /cou… NA NA 2356  
## 5 D 10987 2024-08-15 11:… /cou… NA NA 2356  
## 6 A 16628 2024-08-17 11:… /cou… NA NA 2478  
## # ℹ 2 more variables: web\_application\_action <chr>,  
## # web\_application\_controller <chr>

### Importing data by changing the default directory path

When sharing R scripts, variations in default directories across different computers can complicate the process. A common solution is to change the default working directory to the one containing your data. Let’s say the dataset is in the following folder: “C:/Users/example\_path/”. To set the working directory and load the data:

#Set the new default path  
setwd("C:/Users/example\_path/")   
  
#Import just by using the dataset name  
requests <- read\_excel("example\_requests.xlsx")

### Importing data by using a string variable for the path

The final option is to save the folder path as a character string variable, which can specify the correct path when importing each dataset.

#Path to folder saved as a character string  
folder\_path<- "C/:Users/example\_path/" #Don't forget the quotation marks!  
  
#Check the name   
folder\_path

## [1] "C/:Users/example\_path/"

#Make sure it is a character string  
class(folder\_path)

## [1] "character"

By defining a “folder\_path” variable at the start, you can reference it every time you load a dataset. This avoids repetitive path changes and simplifies the whole process, especially when sharing scripts. Since the read\_excel(), as most other functions only accept one value we need to concatenate the strings into one. This is done using the paste0() (check ?paste0) function:

#The complete path to the folder   
requests\_path\_string<- paste0(folder\_path, "example\_requests.xlsx")   
  
#View the string composed of the path and dataset name as a character string  
requests\_path\_string

## [1] "C/:Users/example\_path/example\_requests.xlsx"

requests <- read\_excel(requests\_path\_string)

While setting the default working directory might seem convenient, it’s generally better practice to define and use a specific path variable for loading data. This approach avoids affecting other scripts that might rely on the default path.

## Data Overview

A clear understanding of your data before diving into analysis can prevent many issues later on. We will start with a data overview to ensure you have a good grasp of the dataset structure and content.

### Key elements in Learning Management Systems (LMS) trace data

At its core, trace data captures the requests/clicks students make when using an LMS. Each click is a data point that typically includes: \* **Timestamp: The exact time the click occurred.** \* **Student and Course: Information about who made the click.** \* **LMS Feature: Details on what the student interacted with.**

While these three columns are fundamental, modern LMS data often includes much more information. For instance, the Canvas LMS data structure can be found here (<https://portal.inshosteddata.com/docs>), which might seem overwhelming regardless of your experience level. Remember, even when faced with extensive data, focusing on the following key variables can help solve most problems.

The **timestamp** provides temporal context to the data, allowing you to track the sequence of events and understand how interactions unfold over time. It enables the study of patterns, trends, and behaviors within the learning environment, such as identifying peak activity periods, analyzing learning progression, or detecting anomalies. Most LMSs save datapoints in a clear Year:Month:Day:Hour:Second and even millisecond format, allowing great detail in any time related analysis. When looking at the second feature, “who made the click”, you will receive a **student ID**, usually accompanied by a course ID. Ideally, the IDs should be anonymized in some way to protect the student’s identity. Most times the IDs will remain untouched during the analysis process. Finally, the **LMS feature** that the student clicked on can be the most confusing and difficult to decipher and may require some extra work before it becomes clear.

## Exploratory Functions

### Selecting one variable from a dataset

To work with a specific variable from a dataset, there are a few methods at your disposal. You can use the name of the dataset followed by the dollar sign operator “$” and the name of the variable, enclose the variable name within brackets, or reference the position of the variable within the dataset. The first method is typically the simplest, but the others have their own utility as well. It’s important to note that these methods are for selecting subsets, so their effectiveness depends on the context. For instance, if we’re working with a single dataset, employing these methods will extract one of its variables (also referred to as vectors). However, if we’re dealing with a list (multiple datasets together), these methods would select one of its datasets.

#Select one variable with "$"  
requests$web\_application\_controller   
  
#Select one variable with []  
requests["web\_application\_controller"]   
  
#Select one variable by its location in the dataset  
requests[9]

### Getting the variable/column names: **colnames()**

Now that we have the dataset loaded, let’s explore it a bit. We can start with the col\_names() function. We can use this function to get all the names of the variables in the dataset.

#Show column names  
colnames(requests)

## [1] "student\_id" "course\_id"   
## [3] "timestamp" "url"   
## [5] "assignment\_id" "quiz\_id"   
## [7] "discussion\_id" "web\_application\_action"   
## [9] "web\_application\_controller"

### Exploring data dimensions: **dim(), nrow(), ncol()**

The dim() function shows the “dimensions” of the dataset, the number of rows and columns. This can also be seen in the environment pane (top right), or by running nrow() and ncol(), respectively.

#Dimensions of the dataset  
dim(requests)

## [1] 35 9

#Rows and columns  
nrow(requests)

## [1] 35

ncol(requests)

## [1] 9

### Previewing data: **head(), tail()**

Two good functions are head() and tail() which provide the first six and last six datapoints in the dataset by default. However, we can change the number of datapoints it shows you.

#First six datapoints   
head(requests)

## # A tibble: 6 × 9  
## student\_id course\_id timestamp url assignment\_id quiz\_id discussion\_id  
## <chr> <dbl> <chr> <chr> <dbl> <dbl> <dbl>  
## 1 A 10456 2024-03-20 13:… /cou… 476 NA NA  
## 2 A 10456 2024-03-20 13:… /cou… 476 NA NA  
## 3 B 10456 2024-03-25 08:… /api… NA 798 NA  
## 4 C 10987 2024-07-30 08:… /cou… NA NA 2356  
## 5 D 10987 2024-08-15 11:… /cou… NA NA 2356  
## 6 A 16628 2024-08-17 11:… /cou… NA NA 2478  
## # ℹ 2 more variables: web\_application\_action <chr>,  
## # web\_application\_controller <chr>

#Last two datapoints  
tail(requests, 4)

## # A tibble: 4 × 9  
## student\_id course\_id timestamp url assignment\_id quiz\_id discussion\_id  
## <chr> <dbl> <chr> <chr> <dbl> <dbl> <dbl>  
## 1 D 10456 2024-08-27 13:… /cou… NA NA NA  
## 2 D 10456 2024-08-27 13:… /N NA NA NA  
## 3 D 10456 2024-08-29 13:… /cou… NA NA NA  
## 4 D 10456 2024-08-29 13:… /cou… NA NA NA  
## # ℹ 2 more variables: web\_application\_action <chr>,  
## # web\_application\_controller <chr>

### Measuring data size: **length()**

The length() function shows the number of elements in an object. This is similar to dim(), as length() of a dataset show the number of variables, while length of a variable/vector shows the number of datapoints/rows. This function is especially useful when combined with the unique() function. Together, they show the “unique” number of values. For example, applied to the “course\_id” variable, it shows the number of different courses in the dataset.

#Checking how length works on a dataset as it is the same as the ncol() function  
length(requests)== ncol(requests)

## [1] TRUE

#Checking how length works on a variable as it is the same as the nrow() function  
length(requests$course\_id)== nrow(requests)

## [1] TRUE

We can see that length() of a dataset is equal to the number of variables it has, while length() of a variable (“requests$course\_id”) is equal to the number of rows.

#The number of values in the "requests$course\_id" variable   
length(requests$course\_id)

## [1] 35

#The number of unique values in the "requests$course\_id" variable  
length(unique(requests$course\_id))

## [1] 3

Here we can see that while there are 35 datapoints in the dataset, there are only 3 distinct “course\_id”.

### Data exploration essentials: **str(), summary()**

In general, I recommend starting any data overview with the str() function. This provides the types of variables and the first values of each variable.

#Some the types of variables and the first values  
str(requests)

## tibble [35 × 9] (S3: tbl\_df/tbl/data.frame)  
## $ student\_id : chr [1:35] "A" "A" "B" "C" ...  
## $ course\_id : num [1:35] 10456 10456 10456 10987 10987 ...  
## $ timestamp : chr [1:35] "2024-03-20 13:48:21" "2024-03-20 13:48:26" "2024-03-25 08:20:56" "2024-07-30 08:19:04" ...  
## $ url : chr [1:35] "/courses/10456/assignments/476" "/courses/10456/assignments/476" "/api/v1/courses/10456/quizzes/798/submissions?per\_page=100" "/courses/10987/discussion\_topics/2356" ...  
## $ assignment\_id : num [1:35] 476 476 NA NA NA NA NA NA NA NA ...  
## $ quiz\_id : num [1:35] NA NA 798 NA NA NA NA NA NA NA ...  
## $ discussion\_id : num [1:35] NA NA NA 2356 2356 ...  
## $ web\_application\_action : chr [1:35] "show" "show" "index" "create" ...  
## $ web\_application\_controller: chr [1:35] "assignments" "assignments" "quizzes/quiz\_submission\_api" "discussion\_topics" ...

#Some descriptive data about the variables  
summary(requests)

## student\_id course\_id timestamp url   
## Length:35 Min. :10456 Length:35 Length:35   
## Class :character 1st Qu.:10456 Class :character Class :character   
## Mode :character Median :10456 Mode :character Mode :character   
## Mean :11812   
## 3rd Qu.:10987   
## Max. :16628   
##   
## assignment\_id quiz\_id discussion\_id web\_application\_action  
## Min. :428.0 Min. :746.0 Min. :2356 Length:35   
## 1st Qu.:436.0 1st Qu.:759.0 1st Qu.:2356 Class :character   
## Median :444.0 Median :798.0 Median :2478 Mode :character   
## Mean :448.6 Mean :780.7 Mean :2424   
## 3rd Qu.:460.0 3rd Qu.:798.0 3rd Qu.:2478   
## Max. :476.0 Max. :798.0 Max. :2478   
## NA's :28 NA's :29 NA's :26   
## web\_application\_controller  
## Length:35   
## Class :character   
## Mode :character   
##   
##   
##   
##

Similarly, the summary() function shows the types of variables and some descriptive data. Based on the values, R has automatically assigned different classes to the variables. Since most IDs are numeric, they were interpreted as integers, resulting in calculated values such as means and medians. However, these computations do not hold significance at this stage. Nonetheless, utilizing functions like summary() can assist in determining the appropriate course of action, such as changing their class.

### Tabulating data: **table()**

Finally, a particularly useful function for gaining an overview of the data is table(). This function generates a tabulation of the counts for each unique value within a variable. In our dataset, it displays the frequency of requests per student.

#Shows the requests per each student  
table(requests$student\_id)

##   
## A B C D   
## 8 8 3 16

Additionally, you can include a second variable to create a contingency table which shows the count for each combination of values between the two variables. For example, by using both the student and the course variables, we can see the number of requests per unique student per unique course.

#Shows the requests per each student for each course  
table(requests$student\_id, requests$course\_id)

##   
## 10456 10987 16628  
## A 5 0 3  
## B 5 2 1  
## C 2 1 0  
## D 8 5 3

## Good Practices and Recap

Applying good programming practices is crucial for code quality and maintainability. In R, as in any programming language, adhering to these practices makes a significant difference. Firstly, maintain an 80-character line limit for better readability, especially when sharing code. This limit is typically marked in your script editor by a vertical line; press “Enter” when you reach it to continue on the next code-line while preserving function hierarchy. Use meaningful and consistent naming conventions for variables and functions (e.g., requests and random\_integers\_mean). Add comments using # to explain your code’s logic and enhance understanding. Break your code into modular, reusable functions, each performing a single task (as presented in the previous example when calculating the average of five random numbers). Use version control systems like Git or osf.io to track changes and collaborate effectively. In the end, be consistent when naming, commenting, and formatting across your projects, even if not all recommended practices are followed.

In this tutorial, we covered essential concepts and techniques for starting to work with students’ trace data, starting from setting up the environment to basic exploratory functions in R. We introduced a general framework for problem-solving when programming and demonstrated its application through practical examples. We explained how to use function arguments, different data types and structures, and provided methods to check them. We delved into importing data using various methods, gave an overview and highlighted key elements in LMS trace data. Finally, we explored data with essential functions such as colnames(), dim(), nrow(), ncol(), head(), tail(), length(), str(), summary(), and table(), equipping you with the foundational tools to effectively open and understand your data in R. We also described some good practices which you are highly encouraged to follow. The next part will continue this journey. We will use some example datasets which have the exact same structure as Canvas data, and pre-process them to prepare them for the final analysis.