A black and white logo

Description automatically generated

**StratoBayes Workshop**

**18th – 20th March 2024, Durham, UK**

**Worksheet for Day 1 Session 1:**

**Introduction to R and RStudio**

Adapted by Andrew Millard from materials prepared by Dr Sally Street,   
Department of Anthropology, Durham University.

This work is licensed under a   
[Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](http://creativecommons.org/licenses/by-nc-sa/4.0/).

[Creative Commons License](http://creativecommons.org/licenses/by-nc-sa/4.0/)

# Session 1: Introduction to R and RStudio

## Launching the software

You should already have installed R from <https://www.r-project.org/> and RStudio from <https://posit.co/downloads/>

R is the software that does all the work behind the scenes, while RStudio provides a more convenient interface. You do not need to launch R separately – it will run automatically in the background when you launch RStudio.

## Interacting with the software

You should see a window open like the one below, containing three smaller windows: the larger one on the left is the **console** containing the **command line**, the top right one contains information about the **workspace** and **history**, while the bottom right one is used primarily to view **plots** and **files.**

A screenshot of a computer

Description automatically generated

**Workspace & history**

**Plots & files**

**Command line**

**Console**

For now, the most important window to focus on is the **console**. In R you will be interacting with the program by entering commands rather than clicking on menu options, as you may be used to with statistical software such as SPSS. Commands are entered on the **command line** immediately after the chevron and blinking cursor.

Interacting with R is a little like a one-sided conversation – we will be feeding it instructions, and it will return results. A large part of learning to use R is adapting to its own particular programming language and its syntax, as well as its quirks – similarly to learning a human language!

To get used to the interface, we’ll start by entering a few basic instructions. Try, for example, typing some simple arithmetic using numbers, mathematical symbols (+, -, \*, /) and round brackets on the command line. Press enter, and R should return the result on the line below, e.g.

(35 \* 7) / 40

And R will return the answer, 6.125.

So far, so good – but of course R can do many, much more useful things for you than basic maths. In order to make best use of R we’ll need to get to grips with some of the basics of its language.

## Functions

**Functions** are specific actions that we can instruct R to do, such as calculating a mean, plotting a graph, running a particular statistical test and so on. R comes ready installed with a huge library of functions (referred to as **base** functions), but you can expand its capabilities further by installing specialised **packages** of functions or by writing your own.

Functions in R need to be entered using commands structured in the following format:

*function(arguments)*

Where ‘function’ will be the name of the function, and the round brackets will contain items (called **arguments**) you want to perform the function on (such as a list of numbers, a set of data, etc.).

For example, we can use the sum function to add up a list of numbers. Enter the command below to try this out (**Note:** you can copy-paste commands directly from the worksheet into R):

sum(1, 2, 3, 4, 5)

The argumentsinside the brackets need to be entered in various ways specific to each particular function. They should always, however, be separated by commas.

## Objects

One of the first things that is important to understand about R is that we primarily interact with the software by creating and storing **objects** in the software’s memory. Objects can contain a huge variety of different things, from individual numbers or words to entire datasets, plots and statistical results.

To create an object, we need to enter commands in this format:

*object <- items*

Where ‘object’ stands in for your chosen name for the object, and ‘items’ refers to whatever you want to store inside it. The ‘arrow’ sign (<-) indicates assignment – so R is assigning any items to the right of the arrow to the named object on the left of the arrow. You can call objects whatever you like, but you should use alphanumeric characters only (i.e. no spaces or special characters, although full-stops and underscores are ok).

For example, we could create an object called ‘x’ that contains the number 100 like this:

x <- 100

When you create an object successfully, it will appear as if nothing has happened. This is because R has done as you’ve instructed and stored the object in its memory, with nothing further to report yet. If you want to check the object really is there, just enter the name of the object and R will return whatever is inside it on the line below. Try this out now for ‘x’ to see for yourself.

The vast majority of commands in R use functions and objects together, using the following format:

*object <- function(arguments)*

For example, bringing all the above together we could create an object called ‘y’ containing the sum of a list of numbers as follows:

y <- sum(1, 2, 3, 4, 5)

**Tip:** to scroll through previous commands, click on the command line and press the up/down keys.

## Scripts

It is not usually a good idea to type commands directly into the console window, because it is very easy to lose track of what you have been doing and make mistakes. Instead, you should create a **script** file to store all the commands you will use for a particular project. A script is simply a text file where you can type and save your commands so that you have a record of what you have done, which you can return to at a later date and pick up where you left off.

To create a new script file, click File>New File>R Script, and you will see a new window open in the top left-hand corner (see below). Then, click File>Save As to save a copy of your script file.

A screenshot of a computer

Description automatically generated

**Script window**

To run commands from the script window, first type your command on a new line in the script window. Then, place your cursor on the same line as the command and press **control+enter**, and the command will shoot down from the script to the console window.

Try this now using one of the commands we have encountered so far, such as:

y <- sum(1, 2, 3, 4, 5)

**Always** make sure you save your script file regularly, or you’ll lose your work if (or when!) R crashes.

## Annotation

Even if you save all your commands in the script window and feel confident that you know what you’re doing at the time, you will probably find it difficult to make sense of things if you return to a script after several weeks or months. Therefore, you should always **annotate** your code. This means adding notes to your script that explain what the commands are doing, in a way that is readable by a human rather than a computer. Be kind to your future self and write whatever will be most helpful!

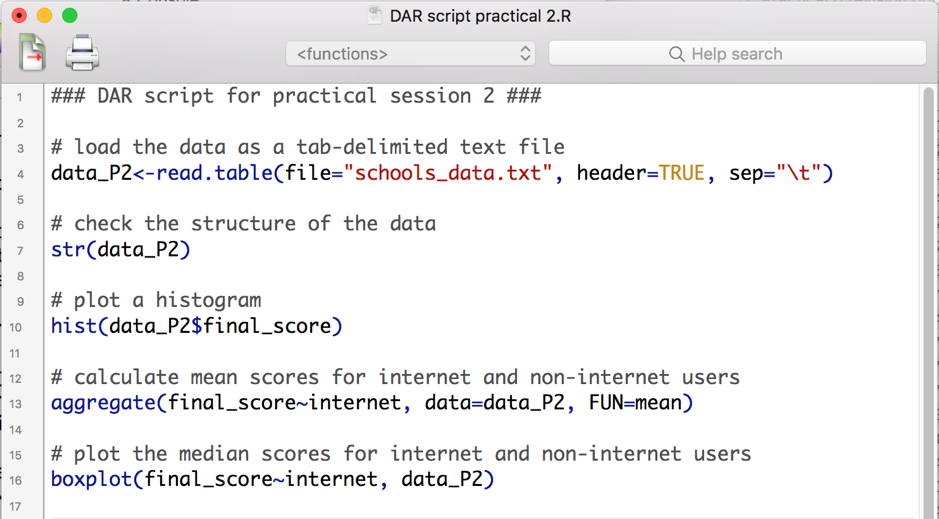
You can annotate your commands in the script window by preceding text with a hash (#). For example, try running the command below. Use your cursor to highlight both lines together before running them, so that they will be sent down to the command line at the same time.

# sum the list of numbers and assign them to the object y

y <- sum(1, 2, 3, 4, 5)

You’ll see that while R reads the command, it ignores the text you’ve entered preceded by the hash key. Because the text is not read by R, you can write whatever you like (including special characters).

You should **annotate all of your commands** during the workshop and for any project you do in R. The screenshot below shows you an example of a well-annotated script from an undergraduate class in anthropology, so you can get an idea of what to aim for. You’ll see that while the commands won’t make much sense to you yet, you’ll be able to understand something of what the student was doing with the data based on their annotations. In a similar vein, using meaningful object names makes reading scripts easier.



## Vectors

Vectors are one of the most fundamental types of object in R. A **vector** is a sequence of elements which are all of the same type (such as numbers or characters). In R, the function c (for ‘concatenate’) is used to combine multiple elements into a vector. For example, try using the command below to create a vector ‘z’ containing round numbers from 1 to 10:

z <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Alternatively, we could do the same using the colon operator to generate a sequence of integers from X to Y (inclusive):

z <- c(1:10)

When creating a vector containing character strings, all elements must be contained in double quotation marks, for example:

fruits <- c("mango", "papaya", "watermelon")

R’s functions are typically **vectorized**, which means that if you apply them to a vector, the function will be performed on all individual elements contained within the vector. For example, if you multiply z by 10 you should see the following output returned:

[1] 10 20 30 40 50 60 70 80 90 100

## Indexing

Once you’re able to create and store objects, you’ll often find yourself needing to select specific items from them based on various criteria – this is called **indexing.**

Indexing requires a command in the following format:

*object[criteria]*

Where ‘object’ stands in for the name of your object (e.g. a vector), and ‘criteria’ the criteria by which you want to select the items. **Note**: you must use square rather than round brackets for indexing.

One way to index is by the numeric position of elements within the vector. For example, the first command below creates a new vector ‘v’, while the second selects the second and fifth elements:

v <- c("A", "B", "C", "D", "E")

v[c(2, 5)]

**Note** the use of c within the square brackets on the second line – this is because the selection criteria themselves need to be combined into a vector, when there is more than one!

As before, we could also use the colon operator for the criteria if we wanted to select a list of elements from one number through to another, e.g.

v[c(1:4)]

Alternatively, you can select elements that meet certain criteria using **relational operators**, i.e.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |
| == | exactly equal to |
| != | not equal to |

**Note:** in R, ‘is equal to’ is indicated by double, rather than single, equals signs. The single equals sign can be used in place of the arrow (<-) to indicate assignment, but this is potentially confusing.

For example, using the vector z we created earlier we could find out which numbers in the list are 5 or more by entering:

z >= 5

And we could extract these numbers from the list using:

z[z >= 5]

The utility of indexing will become more obvious in the second session, where we will cover how to select specific observations from datasets based on various criteria.

## Error and warning messages

When things go wrong, R will (usually!) tell you by reporting an error or warning message in red text. Don’t worry if this happens – it just means that you have probably not used the appropriate language in your commands. **Errors** mean that R cannot do what you have asked it to do, while **warnings** mean that R has been able to do what you have asked, but thinks this might cause problems.

For example, try typing in some random text into the command line and entering it. You’ll see an error message returned like this:

Error: object 'sdfds' not found

R is telling you here that it cannot find any object called ‘sdfds’, because no such object exists.

To see an example of a warning, try the command below. The function as.numeric can be used to convert an object to a number or series of numbers.

as.numeric("apples")

R will return the following result and associated warning message:

[1] NA

Warning message:

NAs introduced by coercion

Here, we get a warning rather than an error because while syntactically correct, this command does not make much sense. We’ve asked R to transform the character string “apples” into a number but this is not a very clear instruction – how should R know which number we want to assign to “apples”? Should this be based on the number of letters in the word, the position of the first letter in the alphabet, etc.? So, instead it returns NA (‘not available’) and warns you because this probably isn’t what you wanted it to do.

## Trouble-shooting

Dealing with errors, warnings and other unexpected behaviour in R can be frustrating, but is all part and parcel of learning to code. This section contains some useful tips for problem-solving when things go wrong.

*Read the message*

This might sound obvious, but the first thing to do when you get an error or warning is to read it carefully. R will tell you precisely what the problem is, although it will do so in its own language, which takes some getting used to. Try to read the message in extremely literal terms and ask yourself ‘what have I told R to do’ and ‘what does R think I want to do’?

*Double-check your commands*

99% of the time, problems in R are caused by human input errors rather than anything going wrong with the software itself. Before doing anything else, simply read and re-read your command back to yourself to see if you can spot the problem[[1]](#footnote-2). Pay special attention to commas, quotes, brackets, and capitalisation as these are very easy to get wrong.

*Ask a friend*

It is very easy to go ‘blind’ to our own coding errors, especially if we have been staring at a script for a while. Quite often a pair of fresh eyes solves the problem quickly, so it can be really helpful to ask a peer to check your code for you (and be prepared to do the same for them in return!).

*Use the help function*

In many cases, things go wrong because you are not totally sure what a function does or what arguments it needs to work. If you need to remind yourself of this, enter a question mark immediately followed by the name of a function, for example:

?mean

You will see R’s help page for the function pop up in the bottom right window. Unfortunately, these help files are not super user-friendly for beginners as they make heavy use of R jargon, so do not worry if you can’t make much sense of them. This does help, however, if you simply want to remind yourself what a particular function does and what arguments it requires. If you scroll down to the bottom of the help page you will also see some reproducible examples of the function in action.

*Ask Google or ChatGPT*

The vast majority of the time, someone else will have experienced and resolved the same problem as you and posted it on a statistics forum such as Stack Overflow or Cross Validated. Just by typing in some key words or copy-pasting an error message into Google, you will likely find something helpful.

ChatGPT can be very useful as it often can provide a solution in less time than it would take to check multiple Google search results.

**Note –** do not worry if you cannot make sense of the forums, they can be very technical and take some getting used to. I also would not necessarily recommend you post on them yourself while getting to grips with R – some of the respondents can be a bit harsh with newbie queries…

## Session 1: questions and exercises

**1.** A typical command in R involves creating an object using a function and arguments. Which of the options below correctly represents the format of a typical command in R?

**A.** *object -> function(arguments)*

**B.** *object <- function(arguments)*

**C.** *object <- function[arguments]*

**D.** *object(function(arguments))*

**2.** Find the square root of 35, using the function sqrt

**3.** Which key should you use to preface annotations (“comments”)?

**A.** #

**B.** $

**C.** /

**D.** @

**4.** The following command will return an error. Why is this?

z <- (14, 25, 36)

**5.** How could you resolve the following error?

Error in sum(1532, 5435, 7385, ) : argument 4 is empty

**6.** What is the mean of 42, 79, 11 and 85? Find your answer by first assigning the numbers to a vector, then using the mean function on that vector.

**7.** What is the c function used for?

**A.** calling a function

**B.** combining different data types

**C.** cancelling a command

**D.** concatenating a sequence

**8.** What kind of brackets are used for indexing?

**A.** square

**B.** round

**C.** curly

**9.** Create a character vector (i.e. containing a sequence of items, each of which is a character string) and select a subset of the items based on their numeric position (e.g. the third and fourth elements).

**10.** Create a numeric vector (i.e. containing a sequence of numbers only) and select a subset of the items based on a relational operator such as equal to, greater than, less than, etc.

1. See also the ‘rubber duck debugging’ method: <https://rubberduckdebugging.com/> [↑](#footnote-ref-2)