Getting started with R

Antonio Gasparrini

01 November 2023

Contents

This session provides the basic notions for starting with R. The content is deliberately described in a non-technical and general fashion, focusing on practical matters more than on theoretical issues. However, the basic concepts underpinning the structure of the R software and language are introduced, also providing some definitions. The first part of the session provides information about how to interact with R, running very simple commands in the R console. Then two key concepts, *objects* and *functions*, are introduced, although avoiding technical details. The next section summarizes important features of R as a statistical program, describing how to set the working directory, load and save R files, and R scripts. The last section illustrates how R is structured in *packages*, and how they can be installed and loaded into the R session. This session can easily be run independently as a very short and compact introduction to R. It contains essential information for starting with the software, and each part will be complemented in other sessions.

Running R code

When opening RStudio or R, you are presented with a symbol '>' in the Console window. This step represents the most critical and intimidating moment for the new user. Therefore, we start simple and use R just as a calculator, executing simple arithmetic computations. We first manually input some command at the prompt, for example:

```r
3 + 2/3
```

```
## [1] 3.666667
```

Here, R *evaluates* the *expression* `3+2/3` and, in this case, returns the result. The results are printed in the console, as shown above. The number `[1]` in square brackets identifies the index of the first element printed in the corresponding line. In this case, the result is a single number. We will see later how results with multiple elements in multiple lines are printed.

A line of code can also be run from a *script*. In RStudio, a new script can be opened from the FILE menu (or using the corresponding icon below and selecting the type of file). The command can be written in a line, and run by placing the cursor on the line and pressing CTRL+R (or additionally CTRL+ENTER in RStudio) on the keyboard. Alternatively, the RUN button on the top left of the Editor panel can be used. Scripts will be discussed in more detail in the next sections.

The same line can also contain an expression including multiple commands, divided by a semi-colon '`;`'. In this case, multiple results are returned:

```
sin(3) ; 3^4
```

```
## [1] 0.14112
```

```
## [1] 81
```

Note here the use of functions and operators such as '`sin()`' and '`^`', respectively (more on this later). However, usually commands are simply divided by a newline. In this case, composite expressions with multiple commands can be created by including them between curly brackets '`{}`'. If a command is not complete at the end of a line, the expression remains unevaluated, and the user can continue in the subsequent line(s) until the command is syntactically complete:

```
(3 + exp(2)
- 2 *
sqrt(5))
```

```
## [1] 5.91692
```

In this case, R gives a *continuation prompt* '+' for the lines corresponding to incomplete expressions, which is omitted in this document and replaced by simple indenting for clarity. In a script, a portion of the line or multiple lines can be run by selecting the related part of code with the cursor. It is customary to use indentation to identify multi-line expressions.

Simple text can also be inputted in R, but it needs to be included in double or single quotes. An example:

```
"this is just text"
```

```
## [1] "this is just text"
```

Unquoted text is used to refer to objects, as shown in the next section.

In a line, code written after a hash mark '`#`' is not evaluated, providing a simple way to insert comments, both at the beginning of the line or after some code. For example:

```
3.6 + 5 # this is an addition
```

```
## [1] 8.6
```

Comments are especially valuable for documenting the code in scripts, before running specific commands.

In your interaction with the R evaluator, you can also receive messages, other than inputting commands and being shown results. The two main messages in R are *errors* and *warnings*. An error arises from the evaluation of expressions which do not make sense, both for syntax typos or absence of logical meaning. For instance, if you type in R:

```
3 + "elvis"
```

An error message will be returned, specifying that the addition does not make sense with non-numeric elements.

A warning is instead a message used to signal something unusual in the computation, although not necessarily an error. For example, if you type

```
log(-2)
```

R will return a missing-type result (discussed later), with a warning message, as the logarithm of negative numbers does not actually exist.

Error or warning messages can appear sometimes obscure to the inexperienced user. When an error arises, the evaluator stops and the command is not executed. Instead, the computation is normally evaluated when a warning message is returned, although the result can be different than that expected.

Previous commands can be re-executed, using the vertical arrow keys in the keyboard to scroll forward and backwards through the *command history*. The list is also displayed in the History window of RStudio, where commands can be searched, and then moved (also in groups) to the console or a script with the buttons To Console and To Source, respectively. Once located, the command can also be edited.

Other than interactively, as shown above, R code can also be run in batch mode, using the Windows shell command. This method is often used for automated use of the software but is not relevant for new users.

R objects

The evaluation of expressions, *per se*, can be rather limited for computing purposes. When an expression is used as a simple command, its value is printed and lost. The other type of command you can perform in R is the *assignment* of the result of an evaluated expression to an *object*, which from here on is referred to through a name.

An assignment consists of three parts: the *name* of the object, the *assignment operator* '<-' (an arrow formed by the less-than and the minus signs), and an *expression*. An example:

```
x <- 12 - 9
```

Here, R first evaluates the expression `12 - 9` but, instead than simply returning the result `3`, it assigns it to the new object, which is univocally associated with the name `x`. You can obtain the same result with `12 - 9 -> x`, where the arrow is still pointing to the object receiving the value. In R (differently than S-plus), the assignment can also be carried out with '=', although this use is discouraged as the same operator is applied in other contexts.

The name of an object must start with a letter and can include letters, numbers, dots ('`.`'), or underscores ('`_`'). R is *case-sensitive*, meaning that `x` and `X` represent different objects. The object can also be a string, such as:

```
singer <- "Eddie Vedder"
```

In this case, you must have clear in mind the distinction between the name of the object (`singer`) and its value (`Eddie Vedder`). In R, strings are always reported as text in (single or double) quotes, while objects are referred to as unquoted text.

The name is used as an unambiguous reference to the object. Typing the name of an object results in its value being printed, the same which happens with a simple evaluation of expressions seen in the previous section. An example:

```
x ; singer
```

```
## [1] 3
```

```
## [1] "Eddie Vedder"
```

You can also assign the same value to multiple objects:

```
y <- w <- 6
```

In this case, two objects `y` and `w` are simultaneously created. You can also re-assign a new value to an existing object:

```
x
```

```
## [1] 3
```

```
(x <- 6)
```

```
## [1] 6
```

and in this case, the old value is erased. As demonstrated here, including an assignment command between round brackets causes the object being printed. As explained above, previously created objects can be used in further commands, such as:

```
z <- 4
x + z
```

```
## [1] 10
```

Despite the silly examples illustrated above, this process reveals the flexibility of R as a computing and programming language, where different objects are progressively created by evaluation and assignment of expressions. This model is complemented and fully exploited by the application of proper *functions*, as shown later.

R functions

Most (actually, all) of the computations in R are carried out by *functions*. At the moment, you can think of a function as a piece of code which performs a specific task, most of the time related to specific computations. A more technical description will be given in other sessions. Many functions are available to the user through different packages, as described later.

You have already seen some functions in some previous examples, such as `sin()` or `exp()`. The syntax for a given function `somefun()` is an expression such as `somefun(arg1,arg2,arg3)`. The elements within the brackets, divided by commas, are *arguments* of the function and provide the initial information. Arguments are often expressions returning some values compatible with the arguments themselves, or R objects previously created. They may represent data, starting values or options related to the computation. The behaviour of the function is usually flexibly modulated through different values for the arguments. A function usually returns some values, which can be assigned to a new object.

As an example, we can use the function `rnorm()`, which returns a sequence of numbers randomly sampled from a normal distribution:

```
obs <- rnorm(n=5, mean=3, sd=2)
obs
```

```
## [1] 3.8438564 1.5947221 8.5437601 3.4686320 0.1200963
```

This function has 3 arguments, defining the number of observations to be sampled, and the mean and standard deviation of the normal distribution. The arguments can be named using the operator '=', or otherwise inputted unnamed in the right order (that is, the expression `rnorm(5,3,2)` is identical to that above). If named, the order of the arguments is not important. Abbreviations are allowed, as long as conflicts with other arguments are avoided.

Some of the arguments may be given *default* values when not specified by the user. For example, in `rnorm()` by default `mean=0` and `sd=1`, meaning that observations will be sampled from a standard normal distribution. Let's see:

```
rnorm(3)
```

```
## [1] -0.6726141 -0.1319468 -0.9529946
```

In this case, the object `3` is assigned to the first argument `n`, while the other arguments are given their default values. All the arguments with no default values need to be specified.

Working directory, workspace, and scripts

A session in R has always a reference to a *working directory*, basically where the files to be loaded into the session are searched for, and where the saved files are stored. The default location of the working directory depends on the operating system. In Windows, if R is started using the desktop icon or the Start Menu, the working directory is usually set to the MY DOCUMENT folder. The working directory can be displayed by typing `getwd()`, and is shown in RStudio at the top of the Console window.

```
## [1] "C:/Users/emsuagas/OneDrive - London School of Hygiene and Tropical Medicine/Work/teaching/Rcour
```

By default, the content of the working directory is shown in the Files window of RStudio. However, this window provides a browser for navigating through different folders and directly opening, deleting, or renaming files.

The location of the working directory can be changed by the function `setwd()`, specifying a path, such as:

```
setwd("H:/Rcourse")
```

In this case the new working directory will be RCOURSE in the H drive. Note how in R the forward slash '/' is used to specify the path, while the backward slash is used by convention in the Windows system. Alternatively, the double backward slash '\\' can be used in R. This is to bear in mind when copying paths from Windows.

In RStudio, life is much easier and the working directory can be easily changed using SET WORKING DIRECTORY → CHOOSE DIRECTORY in the SESSION menu. Alternatively, the working directory can be set to the folder displayed in the Files window, using the MORE menu.

In R, the objects which can be accessed during a session are stored in different *environments*. In particular, the objects created during a session are kept in the *global environment*. The collection of these objects constitutes the *workspace*. A list of objects available can be displayed by the function `ls()` (or similarly by `objects()`), typing:

```
ls()
```

```
## [1] "hook_output" "obs"         "singer"      "w"           "x"           "y"           "z"
```

This function returns a sequence of strings corresponding to the names of the objects present in the global environment, in this case the objects created during this session. In RStudio, these objects are also listed in the Environment window (have a look), and you can also click on them to view their content. Some of these objects can be erased from the global environment with the function `rm()` (or similarly by `remove()`). For example:

```
rm(x, z)
ls()
```

```
## [1] "hook_output" "obs"         "singer"      "w"           "y"
```

while all the objects in a session can be erased by the workspace using:

```
rm(list=ls())
```

or using the CLEAR button in the Environment window in RStudio. However, all the objects created during a given session are lost when the session is closed unless the workspace is saved. The function `save()` can be used to save one object or a subset of objects in a specific file with extension .RDATA, including the specific path:

```r
x <- y <- 1
save(x, y, file="files/x.RData")
```

The whole workspace can be saved at any point during the analysis through the function `save.image()`. For instance, the current workspace can be saved in the folder FILES with:

```r
save.image("files/mysession.RData")
```

In this case, a new file MYSESSION.RDATA is saved in the FILES folder that must be present in the working directory. This file, including all the objects (and the command history), can then be re-loaded in following sessions, by:

```r
load("files/mysession.RData")
```

provided such file exists in the working directory. Again, paths to different folders can be specified. These actions can be more easily produced in RStudio by using the SAVE and LOAD icons in the Environment window, or the SESSION menu. Always remember to include the extension .RDATA when saving. A workspace can also be loaded by clicking on the related file in the Files window of RStudio. An R session can be closed by typing `q()`, or simply closing the GUI.

As mentioned earlier, the commands used to perform the analysis are usually written in R *scripts*, instead than directly typed in the console. This allows the analysis to be stored and repeated in future sessions. Multiple scripts can be opened or created using the Editor window in RStudio.

The scripts can be saved in the working directory or elsewhere as text files with extension .R (or the small letter .R), using the main menu or the menu in the Editor window (again, remember to include the extension .R). An entire script can be run using the command `source()` with:

```r
source("files/myscript.R")
```

provided the file MYSCRIPT.R is present in the folder FILES in the working directory. Paths to different folders can also be specified (again, remember the use of the forward slash '/' or double back slash '\\'). However, it is usually more convenient to keep multiple scripts open, each of which can be entirely run using the SOURCE button in the top-right of the Editor window. A script can be opened by clicking on it in an external browser or the Files window in RStudio.

R can also be directly started by double-clicking on a previously saved .RDATA file or, in RStudio, on a .R script file. In this case, the working directory is set to the folder where the file is stored. This means you do not usually need to worry about changing the working directory. Also, the working directory can be set to the location of the script currently active in the Editor window by SET WORKING DIRECTORY $\rightarrow$ TO SOURCE FILE LOCATION in the SESSION menu, so that analyses involving scripts and data from multiple folders can be easily managed.

All the saved workspaces, scripts and other files can be browsed using the Files window in RStudio.

R packages

In R, all functions and internally-stored datasets are organized in *packages*. Packages are better described as *coherent functional environments*, instead than simple collections of compiled code, data and documentation. These environments extend the global environment, providing capabilities for specific statistical (and non-statistical) applications.

Packages can be divided in *standard* and *contributed*. The former are considered part of the R source code, and are maintained and extended directly by the R Development Core Team. Contributed packages are instead written by different authors. The thousands of packages now available cover pretty much all of the conventional and novel statistical applications.

The content of a package, specifically its collection of functions and data, is available when the package is loaded into the R session. Only standard packages are pre-loaded as soon as R is opened. The following command shows the *search path*, namely the environments present in the session when R is opened, including the standard packages (result not shown):

```r
search()
```

Installed packages, which can be loaded into the session, are listed in the Package window of RStudio (alternatively, the list is shown by typing `library()`). To load a contributed package, for example, the BOOT package to perform bootstrapping, you can type:

```r
library(boot)
```

or tick the package in the list provided in the Package window of RStudio. Now the package BOOT is listed as an additional environment in the search path, and its functions and data available to be used in commands. Some packages contain functions which depend on other packages: in this case, also the latter are automatically loaded into the session.

Only the contributed packages *recommended* by the R Development Core Team are installed through the main R distribution. However, most of the other contributed packages are stored in the CRAN, or other repositories. The function `install.packages()` can be used to download packages from CRAN, or alternatively (but more rarely) from a .ZIP file containing the pre-compiled binaries or from a .TAR.GZ file with the source code. For instance, we can install the package maps for doing basic mapping in R:

```r
install.packages("maps")
```

This step is easier in RStudio, where packages can be installed through a dialogue box opened with the button INSTALL PACKAGES in the Packages window of RStudio. Newly installed packages must then be loaded into the session as above. A package can be removed from a library with the `remove.packages()` function:

```r
remove.packages("maps")
```

The packages are placed in *libraries*, namely specific directories in your machine. Packages included in the main distribution (standard and recommended) are stored in the folder LIBRARY of the R installation, while other packages installed by the user are commonly stored (in Windows) in a WIN-LIBRARY folder. Other system-specific locations are sometimes used. The paths for all the libraries can be found with:

```r
.libPaths()
```

Contributed packages are extended and improved constantly with new versions, which can be updated by the function `update.packages()`, or again directly through the browser opened with the button UPDATES in the Packages window of RStudio:

```r
update.packages()
```