First steps in R

Claudius Gräbner-Radkowitsch

2/12/2022

Contents

1 Issue commands to your computer **1**

2 Objects, functions, and assignments **2**

3 R packages **4**

4 Taking stock **7**

In this post we will learn about the basic *syntax* of R. The syntax basically refers to the grammatical rules you must adhere to when communicating with your computer in the language `R`: if you you do not follow the right syntax, i.e. you 'speak' gramatically incorrect, your computer will not understand you and communicate this to you by throwing up an error message.

To learn about these important basics, the post follows the following structure:

- Commands
- Objects, functions, and assignments
- R packages

1 Issue commands to your computer

There are two ways we can communicate with our computer in R Studio: either issuing commands directly via the *console*, or by executing a script.

Lets start by using the console and use R as a simple calculator first: we first want to add the numbers `2`and `5`. To this end, simply type `2 + 5` into the console and press `Enter`. Since the expression `2 + 5` is syntactically correct R code, the computer 'understands' what we want from it and returns the result:

```
2 + 5
```

```
#> [1] 7
```

The `#>` at the beginning of the line indicates that what is written on this line is the output of an R command (but the concrete sign might be different on your computer).

The result of `2+5` is a number (more precisely: a 'scalar'). In R, scalars are always represented as a *vector* of length 1. The `[1]` here indicates that the first element on this line is the first element of the vector. If the result of our calculation was a very long vector that needs to span several lines, at the beginning of the next line R would show us the index of the first number displayed on this line.[1]

In this way we can use R as a simple calculator, because for all simple mathematical operations we can use certain symbols as operators. At this point it should be pointed out that the symbol `#` in R introduces a

[1] You may try this out by typing `1:100` into your console and see what happens: this returns a vector of length 100, which certainly will contain some line breaks.

comment, that means everything in a line after `#` will be ignored by the computer and you can make notes in the code that only help *you* (or other humans) to understand what you have written.

```
2 + 5 # Addition
```

```
#> [1] 7
```

```
2/2 # Division
```

```
#> [1] 1
```

```
4*2 # Multiplication
```

```
#> [1] 8
```

```
3**2 # Exponentiation
```

```
#> [1] 9
```

As an alternative to typing the commands into the console and then press `Enter` to execute them, we can write down the commands in a script, and then execute this script. While the interaction via the console is useful to test the effects of certain commands, scripts are useful if we want to develop more complex operations, and save what you have written for later, or to make them accessible to other people: we can save scripts as a file on our computer, and then use them later.

The operations that we have conducted so far and not particularly exciting, to be honest. Before we proceed with more complex operations, however, we need to understand the ideas of `objects`, `functions`, and `assignments`.

2 Objects, functions, and assignments

> To understand computations in R, two slogans are helpful: Everything that exists is an object. Everything that happens is a function call. John Chambers

The statement 'Everything that exists is an object.' means that every number, function, letter, or whatever there is, is an object that is stored somewhere in the physical memory of your computer. For instance, in the computation `2 + 3`, the number `2` is as much an object as the number `3` and the addition-function, which we call via the operator `+`.

The statement 'Everything that happens is a function call.' means that whenever we tell our computer to do something via R, we are effectively calling a *function*.

Functions are algorithms that apply certain routines to an *input* and produce an *output*. The addition function we called in the calculation `2 + 3` took as input the two numbers `2` and `3`, applied to them the addition routine and produced the number `5` as output. The output `5` is an object in R just like the inputs `2` and `3`, as well as the addition function.

A 'problem' is that in the present case R prints the output of the calculation but we have no access to it afterwards:

```
2 + 3
```

```
#> [1] 5
```

It is stored, for some time, on the physical memory of our computer, but we basically have no way to find it. To address this problem we can issue an *assignment*: whenever we want to keep using the output of an operation, we may give the output a *name*. This name works effectively as a kind of pointer, which points to the place on the computer memory where the output is saved. This way, we can access, and reuse it whenever we call the name. The process of giving a name to an object is called *assignment*, and it is effectuated via the function `assign`:

```
assign("intermediate_result", 2 + 3)
```

We explain the process of calling a function in more detail below. Here we focus on the process of assignment instead. What the function `assign` does is the following: it assigns the name `intermediate_result` to the result of the operation `2 + 3`. We can now call this result by writing its name into the console and press `Enter`:

```
intermediate_result
```

```
#> [1] 5
```

Since making assignments happens so frequently in practice, there is a shortcut to the use of the function `assign`, namely the operator `<-`. Thus, the following two commands to effectively the same thing:

```
assign("intermediate_result", 2 + 3)
intermediate_result <- 2 + 3
```

From now on, we will only use the `<-` operator, which also represents quite nicely the idea of assignments as pointers to certain objects.[2]

> **Digression: why `<-`?** The use of the string `<-` as an assignment operator is at first sight unintuitive, uncomfortable, and rather unique in the world of programming languages. Much more common is the use of `=`. Where does this particularity of R come from? Besides practical reasons – in contrast to `=`, the use of `<-` makes explicit the unidirectionality of an assignment – the main reason is historical: `R` originated from the programming language `S`. This in turn has taken over the `<-` from the language `APL`. And APL, in turn, was developed on a keyboard layout, where `<-` had its own key. Moreoever, since the operator `==` was not commonly used at that time, `=` was already assigned as test for equality (which, today, is basically always done by using `==`). And so one has decided to use `<-` as an assignment operator and while since 2001 you can also make assignments in R using `=`, `<-` remains strictly recommended for the sake of readability as well as some technicalities.

You are not allowed to give names to objects as you wish. All syntactically correct names in R...

- only contain letters, numbers, or the symbols `.` and `_`
- do not start with `.` or a number

Moreover, there are some reserved words that you must not (and cannot) use as names, e.g. `function`, `TRUE`, of `if`. You can have a look at the complete list of forbidden words by calling `?Reserved`.

There is, however, nothing to remember since whenever you try to give an object a name that conflicts with the rules just described, R immediately throws an error message:

```
TRUE <- 5
```

```
#> Error in TRUE <- 5: invalid (do_set) left-hand side to assignment
```

There are, however, some rules that determine what is a *good* name and that you should adhere to whenever possible:

- Names should be short and informative; `sample_mean` is a good name, `vector_2` not so much
- You should **never use special characters**, especially *Umlaute*
- R is *case sensitive*, meaning that `mean_value` is a different name than `Mean_Value`
- Even if this is possible you should never use names that are already used for function provided by R. For instance, an assignment such as `assign <- 2` is possible, but it effectively prevents you from using the function `assign` without further complications.

[2] In theory we can use `<-` also the other way around: `2 + 3 -> intermediate_result`. At first sight this is more intuitive and respects the sequence of events: first, the result of `2 + 3` gets created, i.e. a new object gets defined. Then, this object gets the name `intermediate_result`. However, the code that results from such practice is usually much more difficult to read, so it is common practice to use `<-` rather than `->`.

Note: You can have a look at all current assignments in the `Environment` pane in R-Studio, or list them by calling `ls()`

Note: One object can have more than one name, but no name can ever point to two object. If you re-assign a name, the old assignment will be overwritten:

```
x <- 2
y <- 2 # The object 2 now has two names
print(x)
```

```
#> [1] 2
```

```
print(y)
```

```
#> [1] 2
```

```
x <- 4 # The name 'x' now points to '4', not to '2'
print(x)
```

```
#> [1] 4
```

Note: As you might have experienced, R does not return results after making an assignment:

```
2 + 2 # No assignment, R returns the result in the console
```

```
#> [1] 4
```

```
x <- 2 + 2 # Assignment, R does not return the results in the console
```

If you want to remove an assignment you can use the function `rm()`:

```
x <- 2
rm(x)
x
```

```
#> Error in eval(expr, envir, enclos): object 'x' not found
```

You can remove all assignment by clicking on the broom in the upper right environment panel in R-Studio or by calling the following command:

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

3 R packages

Packages are a combination of R code, data, documentation and tests. They are the best way to create reproducible code and make it available to others.The fact that many people solve problems by developing routines, then generalizing them and making them freely available to the whole R community is one of the main reasons for success and wide applicability of R.

While packages are often made available to the public, e.g. via GitHub or CRAN, it is equally useful to write packages for private use, e.g. to write functions implementing certain routines that you use frequently across different projects, document them, and make them available too use in different projects.^ [Wickham and Bryan (2022) provide an excellent introduction to the development of R packages].

When one starts R on our computer we have access to a certain number of functions, predefined variables and data sets. The totality of these objects is usually called `base R`, because we can use all the functionalities easily immediately after installing R on our computer.

The function `assign`, for example, is part of `base R`: we start R and can use it without further ado. Other functions, such as `Gini()` are not part of `base R`: they were written by someone else, and before using them we need to install the package that contains the function definition on our computer.

To use a package in R, it must first be installed. For packages that are available on the central R package platform CRAN, this is done with the function `install.packages()`.[3] For example, if we want to install the package `ineq` this is done with the following command:

```
install.packages("ineq")
```

The package collects a number of functions that allow us to compute common inequality indicators, such as the Gini index.

After having installed the package, we have to options to access the objects that are defined within this project. The first option is to use the operator `:::`

```
x <- c(1,4,5,6,12.9)
y <- ineq::Gini(x)
y
```

```
#> [1] 0.3570934
```

Here we write the name of the package, directly followed by `::` and then the name of the object that we want to use. In this example we want to use the function `Gini()`, which computes the Gini index.

If we ommited the `::`, R would not look into the package `ineq` and, therefore, was not able to find the function, returning an error:

```
y <- Gini(x)
```

```
#> Error in Gini(x): could not find function "Gini"
```

Using `::` is the most transparent and safest way to access objects defined in a package: you immediately see where the object is coming from. At the same time it can be tedious to write the package name so many times, especially if you use many objects from the same package. In this case we can make available all objects from the package by calling the function `library()`:

```
library(ineq)
y <- Gini(x)
```

This process is called *attaching a package*. For the sake of clarity, you should always add a call of `library()` for all packages used within a script *at the very top of the script*. This way you can see immediately which packages have to be installed for the script to work.

In principle, only the packages that are actually used should be read into each script with `library()`. Otherwise you will unnecessarily load a lot of objects and lose track of where a certain function actually comes from. In addition, it is more difficult for others to use the script because many packages have to be installed unnecessarily.

Since packages are produced decentrally by a wide variety of people, there is a danger that objects in different packages get the same name. Since in R a name can only belong to one object, names may be overwritten or 'masked' when loading many packages. While R informs you about this happening when you attach a package, it is easily forgotten and can result in very cryptic error messages.

We will illustrate this briefly using the two packages `dplyr` and `plm`:

```
library(dplyr)
```

```
library(plm)
```

Both packages define objects with the names `between`, `lag` and `lead`. When attaching packages using `library()`, the later package masks the objects of the earlier package. You see this by calling the objects by

[3] PAckages not released on this platform can also be installed directly from repository they were published, e.g. Github. To this end, the package `remotes` must be installed first, then you can use functions such as `install_github()`. A short manual is provided here.

name:

```
lead
```

```
#> function (x, k = 1L, ...)
#> {
#>     UseMethod("lead")
#> }
#> <bytecode: 0x7f999c8d1020>
#> <environment: namespace:plm>
```

The last line informs is about the fact that the function was defined in the package `plm`. If we now want to call the function `lead` from the package `dplyr`, we must use `:::`

```
dplyr::lead
```

```
#> function (x, n = 1L, default = NA, order_by = NULL, ...)
#> {
#>     if (!is.null(order_by)) {
#>         return(with_order(order_by, lead, x, n = n, default = default))
#>     }
#>     if (length(n) != 1 || !is.numeric(n) || n < 0) {
#>         msg <- glue("`n` must be a positive integer, not {friendly_type_of(n)} of length {length(n)}
#>         abort(msg)
#>     }
#>     if (n == 0)
#>         return(x)
#>     if (vec_size(default) != 1L) {
#>         msg <- glue("`default` must be size 1, not size {vec_size(default)}")
#>         abort(msg)
#>     }
#>     xlen <- vec_size(x)
#>     n <- pmin(n, xlen)
#>     inputs <- fix_call(vec_cast_common(default = default, x = x))
#>     vec_c(vec_slice(inputs$x, -seq_len(n)), vec_rep(inputs$default,
#>         n))
#> }
#> <bytecode: 0x7f993d0dc8d0>
#> <environment: namespace:dplyr>
```

This can be very confusing. Thus, I *strongly recommend* to *always* use `::` when it comes to masking, no matter whether it is stricly necessary or not. In this case, always use `plm::lead` and `dplyr::lead`, even it it is not required in the first case. Otherwise, your code becomes very difficult to understand and breaks completely once you change the sequence of the library calls in the beginning.

> **Hint**: You can show all object that are affeceted by conflicting names via the function `conflicts()`.

For the sake of transparency I will always use the notation with `::` whenever I refer to an object that is not defined in `base R`. Only in the case of objects that are part of base I will stick to only writing the object name.

> **Digression**: In order to check the order in which R searches for objects, the function `search()` can be used. When an object is called by its name R first looks in the first element of the vector, the global environment. If the object is not found there, it looks in the second, and so on. As you can also see here, some packages are read in by default. If an object is not found anywhere, R gives an error. In the present case, the function shows us that R only looks in the package `plm` for the function `lead()`, and not in the package `dplyr`:

```
search()
```

```
#>  [1] ".GlobalEnv"        "package:plm"       "package:dplyr"
#>  [4] "package:ineq"      "package:bit64"     "package:bit"
#>  [7] "package:tufte"     "package:stats"     "package:graphics"
#> [10] "package:grDevices" "package:utils"     "package:datasets"
#> [13] "package:methods"   "Autoloads"         "package:base"
```

> **Further information**: To better understand masking you might want to learn about the concepts of *namespaces* and *environments*. Wickham and Bryan (2022) is an excellent source to do so.

4 Taking stock

Lets recap what we have learned so far about issuing commands, names and assignments:

- We can issue commands to the computer in R by (a) typing R code into the console and press `Enter`, or (b) write the code into a script and then execute it
- Everything that *exists* in R is an *object*, everything that *happens* is a function call
- A function is an object, that takes an input, applies a certain routine, and returns an output
- We can assign an object a name by using `<-`. Then we can call this object by typing its name. The process of giving a name to an object is called *assignment*, and we can have a look at all names currently given to objects by calling `ls()`
- R packages are bundles of objects and functions created by others and made available to the R community. After installing packages, we can access their objects via `PackageName::ObjectName`, or by attaching the package via `library(PackageName)`

Finally, I want to point your attention to the function `help()`, which can provide you with additional information about the object a name points to. For instance, if you want to get more information about the function with the name `assign`, then just type the following:

```
help(assign)
```