Introduction to R

UA R Users Group

May 14, 2019

Contents

0.1 Learning Objectives

- Familiarize participants with R syntax
- Understand the concepts of objects and assignment
- Understand the concepts of vectors and data types
- Get exposed to a few functions

0.2 The R syntax

0.3 Creating objects

You can get output from R simply by typing in math in the console

```
3 + 5
```

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

```
12/7
```

```
## [1] 1.714286
```

```
2*4
```

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

```
2^4
```

```
## [1] 16
```

We can also comment what it is we're doing

```
# I am adding 3 and 5. R is fun!
3 + 5
```

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

What happens if we do that same command without the # sign in the front?

```
I am adding 3 and 5. R is fun!
3 + 5
```

Now R is trying to run that sentence as a command, and it doesn't work. Now we're stuck over in the console. The `+` sign means that it's still waiting for input, so we can't type in a new command. To get out of this type `Esc`. This will work whenever you're stuck with that `+` sign.

It's great that R is a glorified caluculator, but obviously we want to do more interesting things.

To do useful and interesting things, we need to assign *values* to *objects*. To create objects, we need to give it a name followed by the assignment operator `<-` and the value we want to give it.

0.4 Assignment operator

For instance, instead of adding 3 + 5, we can assign those values to objects and then add them.

```
# assign 3 to a
a <- 3
# assign 5 to b
b <- 5
```

```
# what now is a
a
```

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

```
# what now is b
b
```

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

```
# Add a and b
a + b
```

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

`<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `x <- 3`, the value of `x` is `3`. The arrow can be read as 3 **goes into** `x`. You can also use `=` for assignments but not in all contexts so it is good practice to use `<-` for assignments. `=` should only be used to specify the values of arguments in functions, see below.

In RStudio, typing `Alt + -` (push `Alt`, the key next to your space bar at the same time as the `-` key) will write `<-` in a single keystroke.

To view which objects we have stored in memory, we can use the `ls()` command

```
ls()
```

```
## [1] "a" "b"
```

To remove objects we can use the `rm()` command

```
rm(a)
```

0.4.1 Exercise

- What happens if we change `a` and then re-add `a` and `b`?
- Does it work if you just change `a` in the script and then add `a` and `b`? Did you still get the same answer after they changed `a`? If so, why do you think that might be?
- We can also assign a + b to a new variable, `c`. How would you do this?

0.5 Notes on objects

Objects can be given any name such as `x`, `current_temperature`, or `subject_id`. You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid but `x2` is). R is case sensitive (e.g., `my_data` is different from `My_data`). There are some names that cannot be used because they represent the names of fundamental functions in R (e.g., `if`, `else`, `for`, see here for a complete list). In general, even if it's allowed, it's best to not use other function names (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). In doubt check the help to see if the name is already in use. It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. There are many functions in R with dots in their names for historical reasons, but because dots have a special meaning in R (for methods) and other programming languages, it's best to avoid them. It is also recommended to use nouns for variable names, and verbs for function names. It's important to be consistent in the styling of your code (where you put spaces, how you name variable, etc.). In R, two popular style guides are Hadley Wickham's and Google's.

When assigning a value to an object, R does not print anything. You can force to print the value by using parentheses or by typing the name:

0.6 Functions

The other key feature of R are functions. These are R's built in capabilities. Some examples of these are mathematical functions, like `sqrt` and `round`. You can also get functions from libraries (which we'll talk about in a bit), or even write your own.

0.7 Functions and their arguments

Functions are "canned scripts" that automate something complicated or convenient or both. Many functions are predefined, or become available when using the function `library()` (more on that later). A function usually gets one or more inputs called *arguments*. Functions often (but not always) return a *value*. A typical example would be the function `sqrt()`. The input (the argument) must be a number, and the return value (in fact, the output) is the square root of that number. Executing a function ('running it') is called *calling* the function. An example of a function call is:

```
sqrt(a)
```

Here, the value of `a` is given to the `sqrt()` function, the `sqrt()` function calculates the square root. This function is very simple, because it takes just one argument.

The return 'value' of a function need not be numerical (like that of `sqrt()`), and it also does not need to be a single item: it can be a set of things, or even a data set. We'll see that when we read

data files in to R.

Arguments can be anything, not only numbers or filenames, but also other objects. Exactly what each argument means differs per function, and must be looked up in the documentation (see below). If an argument alters the way the function operates, such as whether to ignore 'bad values', such an argument is sometimes called an *option*.

Most functions can take several arguments, but many have so-called *defaults*. If you don't specify such an argument when calling the function, the function itself will fall back on using the *default*. This is a standard value that the author of the function specified as being "good enough in standard cases". An example would be what symbol to use in a plot. However, if you want something specific, simply change the argument yourself with a value of your choice.

Let's try a function that can take multiple arguments `round`.

```
round(3.14159)
```

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

We can see that we get `3`. That's because the default is to round to the nearest whole number. If we want more digits we can see how to do that by getting information about the `round` function. We can use `args(round)` or look at the help for this function using `?round`.

```
args(round)
```

```
## function (x, digits = 0)
## NULL
```

```
?round
```

We see that if we want a different number of digits, we can type `digits=2` or however many we want.

```
round(3.14159, digits = 2)
```

```
## [1] 3.14
```

If you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

```
## [1] 3.14
```

However, it's usually not recommended practice because it's a lot of remembering to do, and if you share your code with others that includes less known functions it makes your code difficult to read. (It's however OK to not include the names of the arguments for basic functions like `mean`, `min`, etc...)

Another advantage of naming arguments, is that the order doesn't matter. This is useful when there start to be more arguments.

0.8 Vectors and data types

A vector is the most common and basic data structure in R, and is pretty much the workhorse of R. It's basically just a list of values, mainly either numbers or characters. They're special lists that you

can do math with. You can assign this list of values to a variable, just like you would for one item. You can add elements to your vector simply by using the `c()` function, which stands for combine:

```
one_to_five <- c(1, 2, 3, 4, 5)
one_to_five <- 1:5
one_to_five
```

```
## [1] 1 2 3 4 5
```

A vector can also contain characters:

```
primary_colors <- c("red", "yellow", "blue")
primary_colors
```

```
## [1] "red"    "yellow" "blue"
```

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```
length(one_to_five)
```

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

```
length(primary_colors)
```

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

You can also do math with whole vectors. For instance if we wanted to multiply all the values in our vector by a scalar, we can do

```
5 * one_to_five
```

```
## [1]  5 10 15 20 25
```

or we can add the data in the two vectors together

```
two_to_ten <- one_to_five + one_to_five
two_to_ten
```

```
## [1]  2  4  6  8 10
```

This is very useful if we have data in different vectors that we want to combine or work with.

There are few ways to figure out what's going on in a vector.

`class()` indicates the class (the type of element) of an object:

```
class(one_to_five)
```

```
## [1] "integer"
```

```
class(primary_colors)
```

```
## [1] "character"
```

```
new_digits <- c(one_to_five, 90) # adding at the end
new_digits <- c(30, new_digits) # adding at the beginning
new_digits
```

```
## [1] 30  1  2  3  4  5 90
```

What happens here is that we take the original vector `one_to_five`, and we are adding another item first to the end of the other ones, and then another item at the beginning. We can do this over and over again to build a vector or a dataset. As we program, this may be useful to autoupdate results that we are collecting or calculating.

We just saw 2 of the **data types** that R uses: `"character"` and `"integer"`. The others you will likely encounter during data analysis are:

- `"logical"` for `TRUE` and `FALSE` (the boolean data type)
- `"numeric"` for floating point decimal numbers

- `"factor"` for categorical data. Similar to `"character"` data, but factors have levels

Importantly, a vector can only contain **one** data type. If you combine multiple data types in a vector with the `c()` command, R will try to coerce all the values to the same data type. If it cannot, it will throw an error.

For example, what data type is our `one_to_five` vector if we divide it by 2?

```
divided_integers <- one_to_five/2
divided_integers
```

```
## [1] 0.5 1.0 1.5 2.0 2.5
```

```
class(divided_integers)
```

```
## [1] "numeric"
```

Vectors are indexed sets, which means that every value can be referred to by its order in the vector. R indexes start at 1. Programming languages like Fortran, MATLAB, and R start counting at 1, because that's what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do.

We can index a vector in many different ways. We can specify a position of a single value, a range of values, or a vector of values. We can even specify which values to remove by their indices.

```
one_to_five[3]
```

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

```
one_to_five[1:3]
```

```
## [1] 1 2 3
```

```
one_to_five[c(1, 3, 5)]
```

```
## [1] 1 3 5
```

```
one_to_five[-2]
```

```
## [1] 1 3 4 5
```

0.9 Other data structures

Vectors are one of the many **data structures** that R uses. Other important ones are lists (`list`), matrices (`matrix`), and data frames (`data.frame`)