

# Introduction to R

Data Carpentry contributors

## Contents

Creating objects in R . . . . .	1
Vectors and data types . . . . .	4
Subsetting vectors . . . . .	6
Missing data . . . . .	8
Summary of functions used in this section . . . . .	9

---

### Learning Objectives

- Define the following terms as they relate to R: object, assign, call, function, arguments, options.
  - Assign values to objects in R.
  - Learn how to *name* objects
  - Use comments to inform script.
  - Solve simple arithmetic operations in R.
  - Call functions and use arguments to change their default options.
  - Inspect the content of vectors and manipulate their content.
  - Subset and extract values from vectors.
  - Analyze vectors with missing data.
- 

## Creating objects in R

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

```
3 + 5
12 / 7
```

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

```
weight_kg <- 55
```

`<-` 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` or 3 “is stored as” `x`.

By running the code above, you have created the object `weight_kg`. You should now see it in the “Environment” tab in RStudio (top right, by default).

For historical reasons, you can also use `=` for assignments, but not in every context. Because of the slight differences in syntax, it is good practice to always use `<-` for assignments.

In RStudio, typing `Alt + -` (push `Alt` at the same time as the `-` key) will write `<-` in a single keystroke in a PC, while typing `Option + -` (push `Option` at the same time as the `-` key) does the same in a Mac. Try it out.

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., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they are 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`). If in doubt, check the help to see if the name is already in use. It's also best to avoid dots (`.`) within names. Many function names in R itself have them and dots also have a special meaning (methods) in R and other programming languages. To avoid confusion, don't include dots in names. It is also recommended to use nouns for object names, and verbs for function names.

To make your code be easy to read, it's important to be consistent in the styling of your code (where you put spaces, how you name objects, etc.). Using a consistent coding style makes your code clearer to read for your future self and your collaborators. **For now, try to follow the style used in the workshop code.**

Advanced tip: In the future, as you get comfortable with R and want to improve your style you can consult a "style guide". Three popular guides are Google's, Jean Fan's and the tidyverse's. The tidyverse's is very comprehensive and may seem overwhelming at first. You can install the `lintr` package to automatically check for issues in the styling of your code.

## Objects vs. variables

What are known as **objects** in R are known as **variables** in many other programming languages. Depending on the context, **object** and **variable** can have drastically different meanings. However, in this lesson, the two words are used synonymously. For more information see: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Objects>

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

```
weight_kg <- 55      # doesn't print anything
(weight_kg <- 55)    # but putting parenthesis around the call prints the value of `weight_kg`
weight_kg           # and so does typing the name of the object
```

Now that R has `weight_kg` in memory, we can do arithmetic with it. For instance, we may want to convert this weight into pounds (weight in pounds is 2.2 times the weight in kg):

```
2.2 * weight_kg
```

We can also change an object's value by assigning it a new one:

```
weight_kg <- 57.5
2.2 * weight_kg
```

This means that assigning a value to one object does not change the values of other objects. For example, let's store the animal's weight in pounds in a new object, `weight_lb`:

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`? 126.5 or 220? Print the value of the object in R to check your answer.

## Comments

The comment character in R is `#`, anything to the right of a `#` in a script will be ignored by R. It is useful to leave notes and explanations in your scripts.

Advanced tip: RStudio makes it easy to comment or uncomment a paragraph: after selecting the lines you want to comment, press at the same time on your keyboard `Ctrl + Shift + C`. If you only want to comment

out one line, you can put the cursor at any location of that line (i.e. no need to select the whole line), then press Ctrl + Shift + C.

## Challenge

1.1 What are the values of the objects after each statement in the following?

```
mass <- 47.5           # mass?
age  <- 122            # age?
mass <- mass * 2.0     # mass?
age  <- age - 20       # age?
mass_index <- mass/age # mass_index?
```

## Functions and their arguments

Functions are a wrapper around a set of commands that make it easy to run the commands all together. You can think of them as “canned scripts” or “mini scripts” that automate more complicated sets of commands, such as operations assignments, calculations, etc. Many functions are predefined, or can be made available by importing R *packages* (more on that later).

A function usually takes one or more inputs called *arguments* or *parameters*. 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 (i.e. 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:

```
weight_kg_sqrt <- sqrt(weight_kg)
```

Here, the value of `weight_kg` is given to the `sqrt()` function. The `sqrt()` function calculates the square root and returns the value, which is then assigned to the object `weight_kg_sqrt`. This function is very simple, because it takes just one argument.

The return ‘value’ of a function need not be numeric (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 dataset. We’ll see that when we read data files into 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). Some functions take arguments which may either be specified by the user, or, if left out, take on a *default* value: these are called *options*. Options are typically used to alter the way the function operates, such as whether it ignores ‘bad values’, or what symbol to use in a plot. However, if you want something specific, you can specify a value of your choice which will be used instead of the default.

Let’s try a function that can take multiple arguments: `round()`.

```
round(3.14159)
```

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

Here, we’ve called `round()` with just one argument, `3.14159`, and it has returned the value `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)` to find what arguments it takes, 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
```

And if you do name the arguments, you can switch their order:

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

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

It's good practice to put the non-optional arguments (like the number you're rounding) first in your function call, and to then specify the names of all optional arguments. If you don't, someone reading your code might have to look up the definition of a function with unfamiliar arguments to understand what you're doing.

Tip: in RStudio type "round()" then put your cursor in between the parentheses. RStudio will show you the arguments for the function. Hover over each argument or use the up/down arrow keys to look at each one.

### Challenge

- 1.2 What is the log in base 10 of the population of Germany (83 million people)?
  - a. Create a variable with an appropriate name and assign it the value 83000000
  - b. Use the log() function to do the calculation. How can you specify the base?
- 1.3 The function `toupper()` changes characters to uppercase (capitals). Create this object with the value of a cat name:

```
my_cat <- "Spot"
```

Now run the following two commands:

```
toupper(my_cat)
```

```
toupper("my_cat")
```

Why did these commands give you different results?

Print out the value of the variable `my_cat` again. Has it changed?

What happens if you run this command: `toupper(my_bird)`

Time for a git commit

## Vectors and data types

A vector is the most common and basic "data type" in R, and is pretty much the workhorse of R. A vector is composed by a series of values, which can be either numbers or characters. We can assign a series of values to a vector using the `c()` function. For example we can create a vector of animal weights and assign it to a new object `weights_g`:

```
weights_g <- c(50, 60, 65, 82)
```

```
weights_g
```

A vector can also contain characters:

```
animals <- c("mouse", "rat", "dog")
```

```
animals
```

The quotes around "mouse", "rat", etc. are essential here. Without the quotes R will assume objects have been created called `mouse`, `rat` and `dog`. As these objects don't exist in R's memory, there will be an error message.

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(weights_g)
length(animals)
```

An important feature of a vector, is that all of the elements are the same type of data. The function `class()` indicates the class (the type of element) of an object:

```
class(weights_g)
class(animals)
```

The function `str()` provides an overview of the structure of an object and its elements. It is a useful function when working with large and complex objects:

```
str(weights_g)
str(animals)
```

You can use the `c()` function to add other elements to your vector:

```
weights_g <- c(weights_g, 90) # add to the end of the vector
weights_g <- c(30, weights_g) # add to the beginning of the vector
weights_g
```

In the first line, we take the original vector `weights_g`, add the value 90 to the end of it, and save the result back into `weights_g`. Then we add the value 30 to the beginning, again saving the result back into `weights_g`.

An **atomic vector** is the simplest R **data type** and is a linear (1 dimensional) vector of a single type. Above, we saw 2 of the 6 main **atomic vector** types that R uses: "character" and "numeric" (or "double"). These are the basic building blocks that all R objects are built from. The other 6 **atomic vector** types are:

- "character" for letters
- "double" for numbers that might have decimal values
- "integer" for numbers that never have decimal values (whole numbers) (e.g., 2L, the L indicates to R that it's an integer)
- "logical" for TRUE and FALSE (the boolean data type)
- "complex" to represent complex numbers with real and imaginary parts (e.g., 1 + 4i) and that's all we're going to say about them
- "raw" for bitstreams that we won't discuss further

An integer and a double are both **numeric**.

These are the basic building blocks of data in R.

You can check the type of your vector using the `typeof()` function and inputting your vector as the argument.

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

### Challenge

- 1.4 We've seen that atomic vectors can be of type character, numeric (or double), integer, and logical. But what happens if we try to make a vector with a mix of these types? What will happen in each of these examples? (hint: use `class()` to check the data type of your objects):

```
num_char <- c(1, 2, 3, "a")
num_logical <- c(0, 1, 2, 3, TRUE, FALSE)
char_logical <- c("a", "b", "c", TRUE)
tricky <- c(1, 2, 3, "4")
```

- 1.5 Why do you think it happens?
- 1.6 In the example below, how many values in the vector `combined_logical` are "TRUE" (as a *character*) (reusing the two `..._logical` vectors from above):

```
combined_logical <- c(num_logical, char_logical)
```

- 1.7 You've probably noticed that objects of different types get converted into a single, shared type within a vector. In R, we call converting objects from one class into another class *coercion*. These conversions happen according to a hierarchy, whereby some types get preferentially coerced into other types. Can you draw a diagram that represents the hierarchy of how these data types are coerced?

Time for a git commit

## Subsetting vectors

If we want to extract one or several values from a vector, we must provide one or several indices in **square brackets**. For instance:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[2]
#> [1] "rat"
animals[c(3, 2)]
#> [1] "dog" "rat"
```

We can also repeat the indices to create an object with more elements than the original one:

```
more_animals <- animals[c(1, 2, 3, 2, 1, 4)]
more_animals
#> [1] "mouse" "rat" "dog" "rat" "mouse" "cat"
```

R indices start at 1. Programming languages like Fortran, MATLAB, Julia, 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.

## Conditional subsetting

Another common way of subsetting is by using a logical vector. TRUE will select the element at that position in the vector, while FALSE will not:

```
weights_g <- c(21, 34, 39, 54, 55)
weights_g[c(TRUE, FALSE, FALSE, TRUE, TRUE)]
#> [1] 21 54 55
```

Typically, these logical vectors are not typed by hand, but are the output of other functions or logical tests. For instance, if you wanted to select only the values above 50:

```
weights_g > 50 # will return logicals with TRUE for the indices that meet the condition
#> [1] FALSE FALSE FALSE TRUE TRUE
```

We can use this to select only the values above 50

```
weights_g[weights_g > 50]
#> [1] 54 55
```

You can combine multiple tests using `&` (both conditions are true, AND) or `|` (at least one of the conditions is true, OR):

```
weights_g[weights_g < 30 | weights_g > 50]
#> [1] 21 54 55
weights_g[weights_g >= 30 & weights_g == 21]
#> numeric(0)
```

Here, < stands for “less than”, > for “greater than”, >= for “greater than or equal to”, == for “equal to”, and != for “not equal to”. The double equal sign == is a test for numerical equality between the left and right hand sides, and should not be confused with the single = sign, which performs variable assignment (similar to <-).

The ! symbol flips a logical value. What does the following give you?

```
!TRUE
#> [1] FALSE
!c(TRUE, FALSE)
#> [1] FALSE TRUE
weights_g[ ! weights_g < 30]
#> [1] 34 39 54 55
```

A common task is to search for certain strings in a vector. One could use the “or” operator | to test for equality to multiple values, but this can quickly become tedious. The function %in% allows you to test if any of the elements of a search vector are found:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[animals == "cat" | animals == "rat"] # returns both rat and cat
#> [1] "rat" "cat"
animals %in% c("rat", "cat")
#> [1] FALSE TRUE FALSE TRUE
animals[animals %in% c("rat", "cat")]
#> [1] "rat" "cat"
```

Having elements in your search vector that do not appear in your query vector is not a problem:

```
animals %in% c("rat", "cat", "duck", "goat")
#> [1] FALSE TRUE FALSE TRUE
animals[animals %in% c("rat", "cat", "duck", "goat")]
#> [1] "rat" "cat"
```

## Challenge

Using the weights\_g vector from above: weights\_g <- c(21, 34, 39, 54, 55)

- 1.8 Use a logical vector to get the first and third number from the weights\_g vector
- 1.9 Get the values from the weights\_g vector that are:
  - a) greater than 30
  - b) between 30 to 40
  - c) exactly 21
  - d) any value other than 21
- 1.10 (Optional) Can you figure out why "four" > "five" returns TRUE?

Time for a git commit

## Missing data

As R was designed to analyze datasets, it includes the concept of missing data (which is uncommon in other programming languages). Missing data are represented in vectors as NA.

When doing operations on numbers, most functions will return NA if the data you are working with include missing values. This feature makes it harder to overlook the cases where you are dealing with missing data. You can add the argument `na.rm = TRUE` to calculate the result while ignoring the missing values. Try the commands below:

```
heights <- c(2, 4, 4, NA, 6)
mean(heights)
max(heights)
mean(heights, na.rm = TRUE)
max(heights, na.rm = TRUE)
```

If your data include missing values, you may want to become familiar with the functions `is.na()`, `na.omit()`, and `complete.cases()`. See below for examples.

```
## Extract those elements which are not missing values.
## Tip: if you're confused, try running is.na(heights) on its own
## then run it again with the !
## then run the whole command together
heights[!is.na(heights)]
```

```
## Returns the object with incomplete cases removed. The returned object is an atomic vector of type `n`
na.omit(heights)
```

```
## Extract those elements which are complete cases. The returned object is an atomic vector of type `n`
heights[complete.cases(heights)]
```

Recall that you can use the `typeof()` function to find the type of your atomic vector.

### Challenge

- 1.11 Using this vector of heights in inches, create a new vector, `heights_no_na`, with the NAs removed.

```
heights <- c(63, 69, 60, 65, NA, 68, 61, 70, 61, 59, 64, 69, 63, 63, NA, 72,
65, 64, 70, 63, 65)
```

- 1.12 Use the function `median()` to calculate the median of the `heights` vector.
- 1.13 Use R to figure out how many people in the set are taller than 67 inches.

Answer

```
heights <- c(63, 69, 60, 65, NA, 68, 61, 70, 61, 59, 64, 69, 63, 63, NA, 72, 65, 64, 70, 63, 65)

##
heights_no_na <- heights[!is.na(heights)]
# or
heights_no_na <- na.omit(heights)
# or
heights_no_na <- heights[complete.cases(heights)]

##
```

```
median(heights, na.rm = TRUE)
# or
median(heights_no_na)

##
heights_above_67 <- heights_no_na[heights_no_na > 67]
length(heights_above_67)
```

Now that we have learned how to write commands, and the basics of R's data structures, we are ready to start working with data and learn about data frames.

Time for a git commit

## Summary of functions used in this section

- `sqrt()` # calculate the square root
- `round()` # round a number
- `args()` # find what arguments a function takes
- `length()` # how many elements are in a particular vector
- `class()` # the class (the type of element) of an object
- `str()` # the structure or overview of the object and the elements it contains
- `toupper()` # convert the given character string to uppercase
- `c()` # create vector; add elements to vector
- `[ ]` # extract and subset vector
- `%in%` # to test if a value is found in a vector
- `is.na()` # test if there are missing values
- `na.omit()` # Returns the object with incomplete cases removed
- `complete.cases()` # elements which are complete cases
- `typeof()` # Returns the type of the given object

Page built on: 2023-04-18 13:20:03