Session 2_2. Introduction to R

Table of contents

Questions 1
Learning Objectives 2
What is object in R? 2
Creating objects in R 2
How to name the object? 2
Comments 4
Functions and their arguments 4
R data 5
R data structures 5
R data types 6
Basic inspection of R objects 6
Subsetting vectors 7
Conditional subsetting 8
Missing data 9

Questions

- What is an object?
- How can values be initially assigned to variables of different data types?
- What data types are available in R?
- What arithmetic and logical operators can be used?
- How can subsets be extracted from vectors?
- How does R treat missing values?
- How can we deal with missing values in R?

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 values from vectors.
- Analyze vectors with missing data.

What is object in R?

In R, an "object" refers to a data structure that holds a specific type of data, along with associated functions (methods) that can operate on that data. The same concept is often called as a 'variable' in other programming languages.

Creating objects in R

To create an object, we need to give it a name followed by the assignment operator `<-`, and the value we want to give it:

```r
area <- 1.0
```

`<-` is the assignment operator. It assigns values on the right to objects on the left. You can also use `=` for assignments, but not in every context.

Best practice: Use `<-` for assignments!

Because of the slight differences in syntax, it is good practice to always use `<-` for assignments. More generally we prefer the `<-` syntax over `=` because it makes it clear what direction the assignment is operating (left assignment), and it increases the read-ability of the code.

How to name the object?

- You want your object names to be explicit and not too long
- Object name can't start with a number (e.g., `2x` is not a valid object name)

- R is case sensitive (e.g., `age` is different from `Age`)
- There are some names that can't be used because they are the names of fundamental objects in R (e.g., `if`, `else`, `for`).
- Best to avoid dots (`.`) within an object name as in `my.dataset`. There are many objects 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.
- The recommended writing style is called *'snake_case'*, which implies using only lower-case letters and numbers and separating each word with underscores (e.g., `animals_weight`, `average_income`).
- It is also recommended to use nouns for object names, and verbs for function names.

Names already taken

You can check the complete list of reserved names here. In general, even if it's allowed, it's best to not use them (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). If in doubt, check the help to see if the name is already in use.

Being consistent

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. Here are three popular style guides in R: Google, Dr. Jean Fan's, and tidyverse's

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 course, the two words are used synonymously. For more information see: HERE

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:

```r
area <- 1.0     # doesn't print anything
(area <- 1.0)   # putting parenthesis around the call prints the value of `area`
```

```
[1] 1
```

```r
area           # and so does typing the name of the object
```

```
[1] 1
```

Comments

Including comments to your code helps you explain your reasoning and forces you to be tidy. A commented code is also a great tool not only to your collaborators, but to your future self. Comments are also the key to a reproducible analysis. In R we use the `#` character for comments.

Functions and their arguments

Function refers to a block of organized, reusable code designed to perform a specific task. Many functions are predefined, or can be made available by importing R ***packages***. A function usually gets one or more inputs called ***arguments***. Functions often (but not always) ***return*** a value. Executing a function ('***run***ning it') is called ***calling*** the function. An example of a function call is:

```
res <- sqrt(4)
res
```

```
[1] 2
```

Exactly what arguments are accepted and what they means differs per function, and must be looked up in the documentation. 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*.

```
round(3.14159)
```

```
[1] 3
```

`round(3.14159)` returns `3`, because the default is to round to the nearest whole number. We can get more information about this function and its argument in its 'Help' page using '?round".

```
?round
```

```
round(3.14159, 2) # inputs without their argument names taken in the order listed
```

```
[1] 3.14
```

```r
round(digits = 2, x = 3.14159) # if you name the arguments, you can switch their order
```

```
[1] 3.14
```

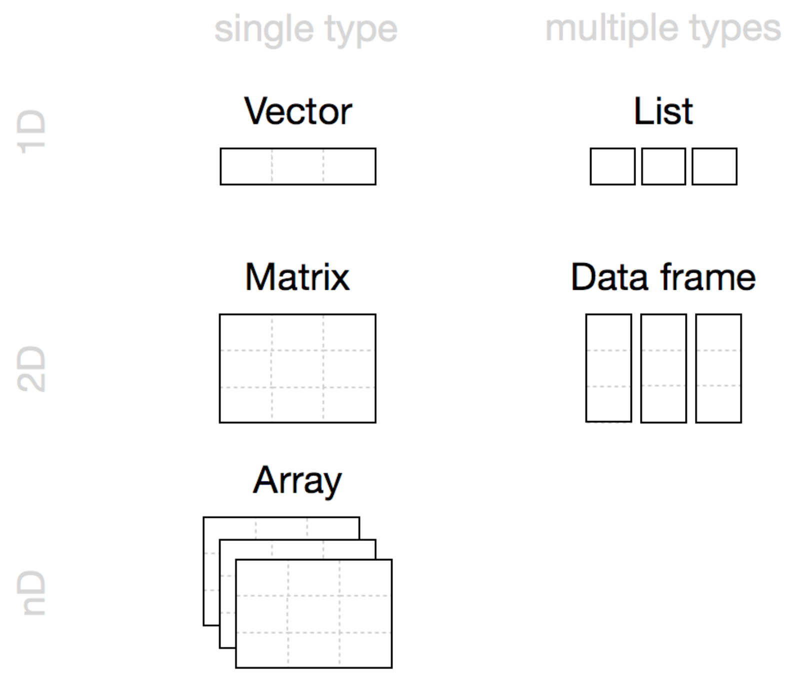
Tip

It's good practice to put the non-optional arguments first in your function call, and to specify the names of all optional arguments.

R data

R data structures

R offers a variety of data structures to store and manipulate data, each with its unique properties and capabilities. Below is the pictorial representation (source) of R's most common data structures.



- ***vector***: a one-dimensional collection of elements, where all the elements are of the *same* data type. You can create it with `c()` function.
- ***list***: a versatile data structure that can store elements of *different* types and sizes. You can create a list using `list()` function.

- ***matrix***: a two-dimensional data structure that extends vector. It's similar to data frame in that they are two-dimensional, however, entries in matrix have to be all the *same* type.
- ***data frame***: a two-dimensional tabular data, consisting of multiple columns where each column represents a vector. Columns of data frame can have *different* types of data.
- ***array***: While we will not focus directly on the `array` data type, which is a multidimensional data structure that *extends matrices*, it is very similar to matrices, but with a third dimension.
- ***factor***: a special type of vector, normally used to hold a categorical variable (e.g. smoker/nonsmoker, state of residency, zipcode) in many statistical functions.

R data types

- ***character***: strings (e.g. "E", "Good morning", "21.3")
- ***numeric*** (or ***double***): floating point numbers (e.g. 10.5, 77, 1094)
- ***integer***: integer numbers (e.g., `2L`, the `L` indicates to R that it's an integer)
- ***logical***: `TRUE` and `FALSE` (the boolean data type)

Example: numeric vector 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.

```r
hh_members <- c(3, 7, 10, 6) ## households members
hh_members
```

```
[1]  3  7 10  6
```

Example: character vector A vector can also contain characters.

```r
nyc_boroughs <- c("Manhattan", "Brooklyn", "Queens", "Bronx", "Staten Island")
nyc_boroughs
```

```
[1] "Manhattan"     "Brooklyn"      "Queens"        "Bronx"
[5] "Staten Island"
```

Basic inspection of R objects

R provides many functions to examine features of vectors and other objects, for example:

- `class()` - what kind of object is it (high-level)?
- `typeof()` - what is the object's data type (low-level)?
- `length()` - how long is it? What about two dimensional objects?

- `attributes()` - does it have any metadata?
- `str()` - overview the structure of an object and its elements

Exercise

Apply above functions to `hh_members` and `nyc_boroughs` objects you created.

Subsetting vectors

Subsetting (sometimes referred to as extracting or indexing) involves accessing out one or more values based on their numeric placement or "index" within a vector.

```r
nyc_boroughs[2]
```

```
[1] "Brooklyn"
```

```r
nyc_boroughs[c(3, 2)]
```

```
[1] "Queens"   "Brooklyn"
```

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

```r
nyc_boroughs_added <- nyc_boroughs[c(1:5, 2, 3, 1)]
nyc_boroughs_added
```

```
[1] "Manhattan"     "Brooklyn"      "Queens"        "Bronx"
[5] "Staten Island" "Brooklyn"      "Queens"        "Manhattan"
```

R indices start at 1.

The first index

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

Using a logical vector

`TRUE` will select the element with the same index, while `FALSE` will not.

```
hh_members
```

```
[1]  3  7 10  6
```

```
hh_members[c(TRUE, FALSE, TRUE, TRUE)]
```

```
[1]  3 10  6
```

Using operators

Below are some of the widely used R operators.

- `&` both conditions are true, AND
- `|` at least one of the conditions is true, OR
- `<` for "less than"
- `>` for "greater than"
- `>=` for "greater than or equal to"
- `==` for "equal to"
- `!=` for "unequal to"
- `!` for "not"
- `%in%` to identify if an element belongs to a vector

```
hh_members > 5
## [1] FALSE  TRUE  TRUE  TRUE
hh_members[hh_members > 5]
## [1]  7 10  6
hh_members[hh_members < 4 | hh_members > 7]
## [1]  3 10
hh_members[hh_members >= 4 & hh_members <= 7]
## [1] 7 6
```

```
nyc_boroughs[nyc_boroughs == "Queens" | nyc_boroughs == "Bronx"]
## [1] "Queens" "Bronx"
nyc_boroughs %in% "Manhattan"
## [1]  TRUE FALSE FALSE FALSE FALSE
"Manhattan" %in% nyc_boroughs
## [1] TRUE
nyc_boroughs %in% c("Brooklyn", "Bronx", "Montauk", "Manhattan", " Manhattan")
## [1]  TRUE  TRUE FALSE  TRUE FALSE
nyc_boroughs[nyc_boroughs %in% c("Brooklyn", "Bronx", "Montauk", "Manhattan", " Manhattan")]
## [1] "Manhattan" "Brooklyn"  "Bronx"
```

Missing data

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. You can add the argument `na.rm = TRUE` to calculate the result while ignoring the missing values. There are other functions to handle missing data - `is.na()`, `na.omit()`, and `complete.cases()`. See below for examples.

```
rooms <- c(2, 1, 1, NA, 7)
mean(rooms)
## [1] NA
max(rooms)
## [1] NA
mean(rooms, na.rm = TRUE)
## [1] 2.75
max(rooms, na.rm = TRUE)
## [1] 7
```

```
## Extract those elements which are not missing values.
rooms[!is.na(rooms)]
## [1] 2 1 1 7

## Count the number of missing values.
## The output of is.na() is a logical vector (TRUE/FALSE equivalent to 1/0) so the sum() fun
sum(is.na(rooms))
## [1] 1

## Returns the object with incomplete cases removed.
na.omit(rooms)
## [1] 2 1 1 7
## attr(,"na.action")
## [1] 4
## attr(,"class")
## [1] "omit"

## Extract those elements which are complete cases.
rooms[complete.cases(rooms)]
## [1] 2 1 1 7
```

Reference
https://datacarpentry.org/r-socialsci/instructor/01-intro-to-r.html