Introduction to R

Objects, Functions, and Vectors

Sumit Mishra

Krea University | WSDS002

Contents

Objects in R

Any stored information in R is an object. So, when you type the minimum age for drinking in Maharashtra (like shown below),

```
25
```

```
## [1] 25
```

the number, while it temporarily exists as an unnamed entity, is not stored in R until you define it as an object. You need to assign a name to an object if you want a particular number or a name to be stored in R. The assignment is done via the operator ←.

```
min_age <- 25

min_age
```

```
## [1] 25
```

Classes of Object

In this course, we will be dealing with the following object types:

Numeric Any real number in R is stored as a numeric variable. Example:

```
a_number <- 2
```

Integer Whole numbers and their negative counterparts. R recognizes any number as an integer if you end that number by the letter 'L'. Example:

```
an_integer <- 2L
set_integers <- 1:10
```

Character Non-numeric variables which can be created by sandwiching any given string between quotes. Example:

```
my_name <- "anthony gonsalves"
```

Logical This object-type defines a true or false condition. Example: Suppose you wanted to know if Anthony Gonsalves is *akela* (alone) in *duniya* (world).

```
it_is <- TRUE #tragic
```

Factor Categorical information are stored as factors. At this point, I will proceed without giving you an example in `R`, but it will help to know that variables like color, gender, race, caste, etc. are typically stored as factor objects.

Things to remember

- `R` is case-sensitive.
- Names cannot contain any special character except for underscore or period[1].
- `R` overwrites objects.
- The list of objects can be gleaned by typing `ls()` in the console

Functions in `R`

Any work with data is incomplete without functions. Recall your excel lessons where you learnt different functions to perform operations. Same goes for `R`. To execute any task you need to run a function. The setup of a function is very simple. It is of the following format: you need to call the name of the function and sandwich the operation (formally known as the argument) within parentheses.

`NAME OF THE FUNCTION(YOUR ARGUMENTS GO HERE)`

Let's discuss some basic functions in `R`.

- `print()`: prints any given object (stored or otherwise).

```
print(min_age) #print a number
```

```
## [1] 25
```

```
print(my_name) #print a character
```

```
## [1] "anthony gonsalves"
```

- `mean()`: computes the average of any numerical object.

```
mean(set_integers)
```

```
## [1] 5.5
```

- `round()`: rounds off a number or a set of numbers.

```
round(2.3942)
```

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

- `factorial()`: computes factorial for any given number.

```
factorial(8)
```

```
## [1] 40320
```

- `sample()`: draws a random sample from any given object.

[1]The name of any object cannot start with any of the special character including the underscore (_)

```r
sample(set_integers, 2)
```

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

As discussed above any given function has arguments. These arguments can be classified into required (without this, you will run into problems) and optional. Let's consider an example. You have a number `2.3942`, and you want it to be rounded off to two decimal places. You will need to specify the optional argument `digits =` into `round()` function[2].

```r
round(2.3942, digits = 2)
```

```
## [1] 2.39
```

You can see the detailed layout of a function by using `args()`. Let's see how this works.

```r
args(sample)
```

```
## function (x, size, replace = FALSE, prob = NULL)
## NULL
```

Upon inspecting the output, you can see that the function `sample()` in `R` has four arguments- `x`, `size`, `replace`,`prob = NULL`. To know more about each of these arguments, you can type `help("sample")` in the console.

Inside an object

When you create an object in `R`, you should try to know more about the object. One way is to look at the `Global Environment` window of `R Studio` which contains the following information: `Name, Type, Length, Size, Value`.

The other way to do the same (recommended) is to use a set of functions in `R`.

- `class()`: returns the class of an object.

```r
class(my_name)
```

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

```r
class(min_age)
```

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

- `str()`: returns the class of an object, its size, and gives you a quick snapshot of different components of the object. This is especially useful when you are dealing with a large dataset.

```r
str(my_name)
```

```
##  chr "anthony gonsalves"
```

```r
str(min_age)
```

```
##  num 25
```

- `typeof()`: returns the specific type of object under a given class.

```r
typeof(my_name)
```

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

```r
typeof(min_age)
```

```
## [1] "double"
```

So, you can see that while the class of the object `min_age` is numeric, the type is `double`.

[2]by default `round()` will use `digits = 0` as you saw in the previous example.

Packages in R

Operations in R are usually dependent upon packages in R. How do you know which package to install? Well, google/stackoverflow is the answer! Let's try to install a package.To install a new package into R, you need to use: the following syntax- `install.packages("PACKAGE NAME HERE", OPTIONAL ARGUMENT)`. Once you install a package, you need to load it to use it for your work. The syntax is `library(PACKAGE NAME HERE)`. Please note that your installed packages should be called via `library` command without wrapping the package name into quotes. `tl;dr` version- you need to deploy two commands `install.packages()` and `library()` to use a package. We want to cut down our time on this work. Therefore, I recommend that you install a package called `pacman` which will do the two tasks of installing and loading packages in one go as shown below. Once you have installed and loaded the package `pacman`, you are ready to use the function called `p_load()` which lets you install and load as many packages as you want[3]! You should run `install.packages("pacman", dependencies = TRUE)` before you compile the following chunk.

```
library(pacman) #loads an existing package

pacman::p_load(dplyr, plyr,
               tidyr, tidyverse) #installs + loads many packages together
```

Folders and Directories in R

Our datasets are stored in directories or folders of our machines. We would want R to know about the directory structure, and work from a particular directory known as the working directory. We should be able to set the working directory, call the working directory, list files in the folder, and change the working directory.

- Setting working directory: `setwd(FOLDER PATH HERE)`
- Print the working directory: `getwd()`
- Change the working directory to machine's default folder: `setwd(~)`
- Move the working directory up one folder: `setwd("..")`
- List all the files in the working directory: `list.files()` or `dir()`
- List files of certain type (example: xlsx): `list.files(pattern = "xlsx")`

Vectors in R

Vectors are one dimensional objects that represent some information. Examples include age, GDP, profits, etc. You can create:

Numeric vectors

```
nv <- c(10,15,20)
print(nv)
```

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

The vector `nv` contains three objects, each indexed by the order in which it appears. Any object within a vector can be called by using the following syntax: `VECTOR[INDEX]` where `INDEX` is a whole number[4].

```
nv[1] #prints the first object of nv
```

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

```
nv[2] #prints the second object of nv
```

```
## [1] 15
```

[3] If you wish to see the list of installed packages, you can type `installed.packages()` on the console.
[4] indexing will work for character vectors as well.

Using rep() and seq() functions

- `rep()`: repeat a number or a set of numbers. You will need to specify two arguments: `x` (the number or the set), `n` (the number of repetitions)
- `seq()`: generates a sequence of numbers. You will need to specify three arguments `from =` (starting point), `to =` (end point), and `by =` (common difference).

In the examples below, I have repeated the vector `nv` twice, and generated an arithmetic progression for numbers between 2 and 30 with a common difference of 4.

```
nv.rep <- rep(nv, 2)
print(nv.rep)
```

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

```
ap <- seq(from = 2, to = 30, by = 4)
print(ap)
```

```
## [1]  2  6 10 14 18 22 26 30
```

Character vectors

```
cv <- c("please please me",
        "with the beatles",
        "a hard day's night")
```

Logical vectors

```
lv <- c(TRUE, TRUE, FALSE)
```

Working with logical operators

When I introduced logical vectors in class, most of you were nonplussed. Logical operators are, however, extremely usefuly because of the fact all of conditional logic rests on the usage of logical operators. For example, you have a large dataset containing information on customers belonging to all geographies, and you want to analyse data for a particular city. Before we learn how to subset and modify vectors, it is important to learn some of the operators. As we saw with the Anthony Gonsalves example, logical operators in `R` check for condition.

- ==: the 'equal' operator

Let's see how this works.

```
stu_age <- 23 #someone's age
stu_age == min_age #check if her age is above legal drinking in Maharashtra
```

```
## [1] FALSE
```

- ! =: the 'not equal' operator

An example:

```
stu_age <- 23 #someone's age
stu_age != min_age #check if her age is above legal drinking in Maharashtra
```

```
## [1] TRUE
```

- & : the 'and' operator

Let's say that you want to eat out in a restaurant which restricts access to alcohol if you are a man (beside state regulation).

```r
is_female <- T # you happen to be a woman
stu_age <- 23 #your age
is_female == F & stu_age >= 25 #check if you're allowed to drink
```

```
## [1] FALSE
```

- `|`: the 'or' operator (type `Shift + backslash` on your keyboard)

In a university, a professor is eligible for promotion if she publishes more than 4 papers over a period of three years or an average rating of 4 on teaching evaluations.

```r
num_papers <- 3 #number of papers
teach_rating <- 4.2 #average rating
num_papers >= 4 | teach_rating >= 4 #check if eligible for promotion
```

```
## [1] TRUE
```

- `%in%`: the 'contained in' operator.

Example: check if the numbers 16 and 18 are contained in the sequence `ap` that we generated earlier.

```r
c(16,18) %in% ap
```

```
## [1] FALSE  TRUE
```

Working with logical functions

Apart from logical operators, it is useful to know some logical functions.

- `any()`: checks if any of the objects meet a condition.

```r
set_age <- c(20, 23, 26, 29)
any(set_age >= 25)
```

```
## [1] TRUE
```

- `all()`: checks if all of the objects meet a particular condition.

```r
all(set_age >= 25)
```

```
## [1] FALSE
```

- `which()`: tells you which of the objects of a vector meet a given condition.

```r
ap %in% c(12,16,18) #this checks whether ap contains those numbers
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
```

```r
which(ap %in% c(12,16,18))
```

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

The output tells you that the fifth object of the vector `ap` is contained in the vector `c(12,16,18)`.

Subsetting a vector

Now that we have learnt logical operators and functions, we are all set to subset a vector (and create a new vector). Consider the following example. To subset, we will use the following syntax: `VECTOR[LOGICAL OPERATION]`.

```r
# Subsetting a numerical vector
ap <- seq(2, 56, 6)

ap[ap < 10] # subset where ap is less than 10
```

```
## [1] 2 8
```

```
ap[ap > 10] # subset where ap is greater than 10
```

```
## [1] 14 20 26 32 38 44 50 56
```

```
ap[ap > 10 & ap < 50] # and condition
```

```
## [1] 14 20 26 32 38 44
```

```
ap[ap ≥ 10 & ap < 40] # 'and' condition
```

```
## [1] 14 20 26 32 38
```

```
ap[ap ≠ 50] # 'not equal to' condition
```

```
## [1]  2  8 14 20 26 32 38 44 56
```

```
ap[ap < 10 | ap > 50] # 'or' condition
```

```
## [1]  2  8 56
```

```
ap[ap > 56] ##in this step, look at the output carefully
```

```
## numeric(0)
```

Modifying a vector

Suppose that you are give a list of Beatles songs, and you notice that someone messed up norwegian wood's year (the actual year is 1965).

```
beatles.songs <- c("please please me",
                   "magical mystery tour",
                   "norwegian wood")
year <- c(1963, 1967, 1963)
names(year) <- beatles.songs # the function names() assigns names to objects in a numeric vector
print(year)
```

```
##     please please me magical mystery tour       norwegian wood
##                 1963                 1967                 1963
```

Let's fix it.

```
year["norwegian wood"] <- 1965 #fixes the year
print(year)
```

```
##     please please me magical mystery tour       norwegian wood
##                 1963                 1967                 1965
```

Summarizing a vector

You can summarize a vector using different functions. Consider, as an example, our old friend ap.

```
#Summarizing Vectors
ap <- seq(2,56,6) # generate a sequence
class(ap)  #check class
```

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

```
length(ap) #check length
```

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

```r
max(ap) #check maximum value inside this vector
```

```
## [1] 56
```

```r
min(ap) #check min
```

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

```r
sum(ap) #computes the sum of the AP
```

```
## [1] 290
```

```r
mean(ap) #average of the sequence
```

```
## [1] 29
```

```r
var(ap) #variance of the sequence
```

```
## [1] 330
```

```r
quantile(ap) #quantiles of the vector (default is quartile)
```

```
##   0%  25%  50%  75% 100% 
##  2.0 15.5 29.0 42.5 56.0
```

```r
quantile(ap, probs = seq(0, 1, 0.1)) #percentiles
```

```
##   0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100% 
##  2.0  7.4 12.8 18.2 23.6 29.0 34.4 39.8 45.2 50.6 56.0
```

```r
summary(ap) #summary statistics
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##     2.0    15.5    29.0    29.0    42.5    56.0
```

Done for the day

```
## Sorry, this silly GIF is only available in the the HTML version of the notes.
```