Fundamentals of R

Jose Toledo Luna

2024-07-21

Table of contents

Variables

A variable provides us with named objects that our programs can manipulate. A valid variable name consists of letters, numbers and the dot or underline characters. It is important to note variable names are case sensitive. That is, `var1` and `Var1` are different variables. Below are appropriate variable names in R

Valid Variable Name	Reason
variable_name	Contains letters and underscore
long.variable_name	Contains letters, dot, and underscore
var	Contains letters
var1	Contains letters and numbers
long.variable_name2	Contains letters, numbers, dot and underscore
var1_name.1	Contains letters, numbers, dot and underscore
.var_name	Can start with period, contains letters and underscore

This is a good starting point for valid variable names. Next we demonstrate a few examples where variable names are invalid.

Invalid Variable Names	Reason
2var	Starts with a number
_varname	Starts with underscore
.2var_name	While starting with a (.) dot is valid, it can not be followed by a number

Now that we have an idea of how to name variables, lets discuss variable assignments. Variables can be assigned values using leftward (`<-`), rightward (`->`) and equal (`=`) operators. However, we will only stick with the leftward and equal assignment operators.

```
var_name1 <- 10
var2 = 20
var.name3 <- 30
var_name_4 = 40
```

Vectors

The easiest method to create any type of vector in R is using `c()` (as in concatenate). We primarily focus on two types of vectors; *numeric* and *character*

Numeric vectors

```
num_vec <- c(0,1,2,3,4)
typeof(num_vec)
```

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

```
class(num_vec)
```

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

Using `c()` is not the only way to generate a vector, we can also generate the above vector using `seq()` as follows

```
seq(from=0,to=4)
```

```
[1] 0 1 2 3 4
```

Another approach to generate the same sequence can be done using `0:4`

```
0:4
```

```
[1] 0 1 2 3 4
```

or more generally `a:b`, where `a` is the starting number and `b` is the last number in the sequence

We can apply arithmetic operations to our numerical vector `num_vec`, such as addition, subtraction, multiplication, division, and exponentiation. These operations will be applied to each element in the vector *(element-wise)*.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division)
%/%	Integer Division

Arithmetic operations applied to numeric vectors follow PEMDAS order of operations, demonstrated in the following example

Subtract 1 from each element

```
(num_vec-1)
```

```
[1] -1  0  1  2  3
```

Subtract 1 from each element, then square them

```
(num_vec-1)^2
```

```
[1] 1 0 1 4 9
```

Subtract 1 from each element, square them, then double each element

```r
 2*(num_vec - 1)^2
```

```
[1]  2  0  2  8 18
```

Subtract 1 from each element, square them, double them, then add 1 to each element

```r
2*(num_vec - 1)^2 + 1
```

```
[1]  3  1  3  9 19
```

```r
pemdas_vec <- 2*(num_vec - 1)^2 + 1
pemdas_vec
```

```
[1]  3  1  3  9 19
```

Generating an odd sequence from 1 to 9, we can use `c()` or `seq()`

```r
c(1,3,5,7,9)
```

```
[1] 1 3 5 7 9
```

```r
seq(from =1,to=10,by=2)
```

```
[1] 1 3 5 7 9
```

Note if you know the ordering of the arguments of a function it is not necessary to specify them. For example, it is optional to write `from` and `to` arguments in the `seq()` function

```r
seq(from = 1,to = 10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq(1,10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Character vectors

```
chr_vec <- c('A','B',"C")
typeof(chr_vec)
```

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

```
class(chr_vec)
```

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

Manipulating vectors

There are multiple ways to access or replace values in vectors. The most common approach is through "indexing". It is important to know in `R` starts with index 1.

```
big_vec <- 1:100
big_vec[1]
```

```
[1] 1
```

```
big_vec[10] # extract the 10th element in your vector
```

```
[1] 10
```

For accessing elements in a vector we can think `vector[indices you want to extract]` the way we extract certain elements can be through some condition, that is `vector[condtion]`

```
big_vec[ c(1,5,10) ]
```

```
[1]  1  5 10
```

```
big_vec[ 1:10 ] # what are the first 10 elements ?
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Using `c()` we can concatenate elements from one vector into another vector. For example, we can add the elements from `pemdas_vec` into the existing vector `num_vec`

```
c(num_vec,pemdas_vec)
```

```
 [1]  0  1  2  3  4  3  1  3  9 19
```

Alternatively, we can add the elements from `num_vec` into the existing vector `pemdas_vec`

```
c(pemdas_vec,num_vec)
```

```
 [1]  3  1  3  9 19  0  1  2  3  4
```

You will notice the order in which we concatenate the elements from the vectors does matter

```
chr_vec
```

```
[1] "A" "B" "C"
```

```
chr_vec[1] <- 'a'
chr_vec
```

```
[1] "a" "B" "C"
```

```
num_vec
```

```
[1] 0 1 2 3 4
```

```
num_vec[3] <- 10
num_vec
```

```
[1]  0  1 10  3  4
```

```
num_vec[ c(1,3) ] <- c(100,200)
num_vec
```

```
[1] 100   1 200   3   4
```

```
num_vec[c(1,2,3)] <- 0
num_vec
```

```
[1] 0 0 0 3 4
```

Installing packages

While base R contains a wide collection of useful functions and datasets, it might be necessary to install additional R packages to increase the power of R by improving existing base R functionalities, or by adding new ones.

In general, you can use this template to install a package in R:

```
install.packages('package_name')
```

For example, in this lab we will need functions/datasets from the following package: `maps`. To install we simply type in our console

```
install.packages('maps')
```

After running the above command you should get something similar to the output below. The messages appeared will depend on what operating system you are using, the dependencies, and if the package was successfully installed.

```
trying URL 'https://cran.rstudio.com/bin/macosx/contrib/4.2/maps_3.4.0.tgz'
Content type 'application/x-gzip' length 3105764 bytes (3.0 MB)
==================================================
downloaded 3.0 MB



The downloaded binary packages are in
    /var/folders/mc/rznpg9ks30sd6wdh7rchs4v40000gn/T//RtmpLUHvkq/downloaded_packages
```

Once the package was installed successfully we now have access to all of its functionalities/datasets. To access them we load the package into memory using the command `library()`

```
library(maps)
```

However, if we only need to access say a specific function/dataset a few times we can do so using the notation `packagename::functionname()`. For example, if we only need to access the Canada cities data set in the `maps` package we run the following command

```
maps::canada.cities
```

```
           name country.etc    pop   lat    long capital
1 Abbotsford BC          BC 157795 49.06 -122.30       0
2      Acton ON          ON   8308 43.63  -80.03       0
3 Acton Vale QC          QC   5153 45.63  -72.57       0
4    Airdrie AB          AB  25863 51.30 -114.02       0
5    Aklavik NT          NT    643 68.22 -135.00       0
```

Alternatively, if you loaded the entire package using `library(maps)` we can access the Canada cities data set using the following command

```
canada.cities
```

```
           name country.etc    pop   lat    long capital
1 Abbotsford BC          BC 157795 49.06 -122.30       0
2      Acton ON          ON   8308 43.63  -80.03       0
3 Acton Vale QC          QC   5153 45.63  -72.57       0
4    Airdrie AB          AB  25863 51.30 -114.02       0
5    Aklavik NT          NT    643 68.22 -135.00       0
```