R 101

Nastasiya F. Grinberg

2024-12-10

Vectors, Matrices, Lists and Data Frames

Vectors

There are several ways to create vectors in `R`:

- Using the `c` function, like so `c(1, 10)`.
- Consecutive numbers using the `:` operator, e.g. `1 : 10`, `-10 : 2`.
- Using the functions `rep` and `seq`:
  - `rep(NA, 10)` creates a vector of `NA`s of length 10.
  - `seq(0, 1, by = 0.1)` creates a vector of values from 0 to 1 by step 0.1.
  - `seq(0, 1, len = 20)` creates a vector of evenly spaced values from 0 to 1 of length 20.
- You can combine all of the above, e.g. `c(1 : 4, seq(0, 1, by = 0.1), 2)`.

You can extract elements from vectors using the square brackets `[]`, e.g. `vec[c(1, 3)]` would return the 1st and 3rd of `vec`; `vec[1:10]` would return the first 10 elements of `vec`. To *omit* elements of a vector, use *negative indices*, e.g. `vec[-1]` returns all *but the 1st element* of `vec`. Note that indexing in `R` starts from 1, not 0 as in many other programming languages. This is true not only for vectors bot for all other types of objects.

Combining conditions and the square brackets is an effective way to extract vector elements with particular properties, e.g. `vec[vec > 10]` returns all elements of `vec` greater than 10.

Matrices

Matrices are usually created using the `matrix` function:

- `matrix(1 : 10, nrow = 2)` creates a matrix from values `1 : 10` arranged in a 2×5 grid.
- `matrix(NA, nrow = 2, ncol = 5)` creates a matrix of `NA`s with 2 rows and 5 columns.

Extracting matrix elements is also performed with `[]` and using the same principles as for vectors, the only difference being that matrices are two-dimensional objects. The two dimensions within the square brackets are separted by a comma with the convention `[row_indices, col_indces]`, e.g. `mat[1:10, 1:10]` returns the submatrix consisting of the first ten rows and first ten columns of the original matrix. Leaving one dimension empty would return all entries of that dimension, e.g. `mat[1:10,]` would return the submatrix of the first ten rows and *all* the columns.

Lists

Lists, unlike vectors, can hold data of different types and are thus very useful. You can create a list like so:

```r
my_list <- list(a = 10, b = 'foo', c = list(1, 2, 3), d = matrix(1:20, nrow = 4))
my_list
```

```
## $a
## [1] 10
##
```

```
## $b
## [1] "foo"
##
## $c
## $c[[1]]
## [1] 1
##
## $c[[2]]
## [1] 2
##
## $c[[3]]
## [1] 3
##
##
## $d
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

Indexing of lists is done via double square brackets `[[]]` or via a `$` operator or a `[[]]` + entry name, if the list is named:

```
my_list[[2]] #second entry
```

```
## [1] "foo"
```

```
my_list$a #an entry called "a"
```

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

```
my_list[["b"]] #an etnry called "b"
```

```
## [1] "foo"
```

If you have a list of single values of the same class, sometimes you might want to “unlist” it into a vector:

```
my_list <- list(1, 2, 3)
my_list
```

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

```
unlist(my_list)
```

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

Data Frames

Like matrices, data frames are rectangular arrays of data but, unlike matrices, data frames can hold values of more than just one type. You can create a `data.frame` from scratch:

```r
my_df <- data.frame(col1 = 1:10, col2 = 11:20)
my_df
```

```
##    col1 col2
## 1     1   11
## 2     2   12
## 3     3   13
## 4     4   14
## 5     5   15
## 6     6   16
## 7     7   17
## 8     8   18
## 9     9   19
## 10   10   20
```

or convert a matrix:

```r
mat <- matrix(1:10, nrow = 2)
mat
```

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

```r
as.data.frame(mat)
```

```
##   V1 V2 V3 V4 V5
## 1  1  3  5  7  9
## 2  2  4  6  8 10
```

```r
data.frame(mat)
```

```
##   X1 X2 X3 X4 X5
## 1  1  3  5  7  9
## 2  2  4  6  8 10
```

You can extract subsets of a `data.frame` using the double square bracket approach utilised for matrices but you can also extract columns using their names:

```r
my_df$col1
```

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

```r
my_df[["col2"]]
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

Vectorisation

Many vector- and matrix-wise operations that require looping in other programming languages are vectorised in `R` and can be easily and intuitively deployed directly on the vector and matrix objects. For example,:

- For a matrix `mat`, `mat^2` returns a matrix with every entry of the original matrix squared. Similarly, `sqrt(mat)` returns a matrix with entries being square roots of entries of `mat`.
- For two matrices of the same dimensions `mat1` and `mat2`, `mat1 + mat2` creates a matrix with each entry being a sum of the corresponding entries of the first two matrices.
- For a vector `vec`, `vec^2` returns a vector with every entry of the original vector squared. Similarly, `sqrt(vec)` returns a vector with entries being square roots of entries of `vec`.

- For two vectors of the same length `vec1` and `vec2`, `vec1 + vec2` creates a vector with each entry being a sum of the corresponding entries of the first two vectors. Note that if vectors are of different lengths, the shorter vector will get *recycled* (try it!).
- Both for vectors and matrices, conditions, e.g. `mat == 0` or `vec == 0`, would return a vector/matrx of the same dimensions with `TRUE` for every entry satisfying a condition, and `FALSE` otherwise. Try, e.g. `mat <- matrix(1 : 10, nrow = 2); mat > 3` or `vec <- rnorm(10); vec > 0`.
- The `sum` function can be effectively combined with conditions for counting, e.g. `sum(vec > 0)` would count a number of positive elements of `vec`.

Common and useful in-built functions

- `colSums` and `rowSums` for summing columns and rows of a matrix or a data frame. Use argument `na.rm = TRUE` if your matrix contains `NA` in order to ignore missing values.
- `colMeans` and `rowMeans` for finding means of columns and rows of a matrix or a data frame. Use argument `na.rm = TRUE` if your matrix contains `NA` in order to ignore missing values.
- `sample` can be used to sample from a vector of values with or without replacement, e.g. `sample(1 : 10, 100, replace = TRUE)` returns 100 values randomly chosen from `1:10` with replacement or `sample(1 : 5)` returns 5 values from `1:5` without replcement (i.e. a random permutation of `1:5`).
- `is.na(x)` returns `TRUE` if `x` is `NA` and `FALSE` otherwise. `is.na` is vectorised, which means that it works on vectors and matrices by applying itself to each of its elements individually.
- `which.min` and `which.max`, when passed a vector, return the index of its minimum and maximum element, respectively. For example, `which.max(c(2, 10, 5))` would return 2 because 10 is the largest element of the vector argument.

- `unique`, given a vector of objects, returns a vector of unique values featuring in the input: try `unique(c(1, 2, 3, 1, 2))` and `unique(iris$Species)`.
- `table`, given a one argument, a vector of values, outputs a table of counts of occurences of each unique value in the vector: try `table(iris$Species)`. Given two or more vectors, `table` counts number of occurences of all unique tuples, e.g. `table(mtcars$cyl, mtcars$gear)`.

Loops

Remember that if you calculte values within a loop that you'd like to save, you need to create an object to store these values. For example, if you want to create a loop to square numbers from 1 to 10 (obviously, a much easier way to do this is `(1 : 10)^2`), you need to create an empty vector to store the squres first:

```r
res <- rep(NA, 10)
for(i in 1 : 10) {
  res[i] <- i^2
}
print(res)
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Similarly, if you would like to create a double loop to calculte i^k for $i = 1, \dots 5$ and $k = 1, 2, 3$, you would need a 5×3 (or a 3×5) matrix for your results:

```r
res_mat <- matrix(NA, nrow = 5, ncol = 3)
for(i in 1 : 5) {
  for(k in 1 : 3) {
    res_mat[i, k] <- i^k
  }
}
print(res_mat)
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    2    4    8
## [3,]    3    9   27
## [4,]    4   16   64
## [5,]    5   25  125
```

`*apply` functions

The family of `*apply` functions (most common `apply`, `sapply`, `lapply`) are an alternative to loops, a very `R`-esque way of applying a function to a matrix, vector or a list. The functions you can use within the `*apply` functions can be arbitrarily complex.

- `apply` can be used to apply a function to each row (1) or column (2) of a matrix. For example, `apply(mat, 1, max)` will find the biggest element in each row (argument `1`) and return them in a vector. You can also supply a more sophisticated function to be applied, e.g. `apply(mat, 2, FUN = function(x) {x[-which.min(x)]})` returns each row *except* the mnimum value.
- `lapply` and `sapply` apply a function to each element of a list/vector and return the output in the form of a list and vector, respectively. Try `lapply(1 : 10, FUN = function(x) {x^2})` and `sapply(1 : 10, FUN = function(x) {x^2})`. The above double-loop can be rewritten using the `sapply` function like so:

```
sapply(1 : 5, FUN = function(i) {
  sapply(1 : 3, FUN = function(k) {
    i^k
  })
})
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    1    4    9   16   25
## [3,]    1    8   27   64  125
```

The advantage of `sapply`/`lapply` over loops is that you do not need to create a dedicated container for the output.

Writing your own functions

In-built functions are not enough to cater for all your needs and more often than not you will have to write your own bespoke functions. To create a function, you follow the following template:

```
fun <- function(x, y) {
  #body of the function
  #return(output)
}
```

Note, however, that `return` is optional as, by default, your function will return the last line. So `fun1 <- function(x) {x^2}` and `fun2 <- function(x) {ans <- x^2; return(ans)}` will return the same answer. Your function can have as many arguments as you like; they might or might not have default values.

Here is an example of a function that calculates sample variance for a vector of values `x` (of course there is an in-built function that does that called `var`):

```
my_var <- function(x) {
  xbar <- mean(x)
  sum((x - xbar)^2)/(length(x) - 1)
}

my_var(rnorm(10))
```

```
## [1] 1.317909
```

Here is an example of a function with two arguments, one of which has a default value:

```
sum_of_powers <- function(x, k = 2) {
  sum(x^k)
}

sum_of_powers(rep(1, 10)) #uses default value of k = 2
```

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

```
sum_of_powers(1 : 4, 3) #k is set to 3
```

```
## [1] 100
```

Links

- General introduction to `R`: "An Introduction to R" (Douglas, Mancini, Couto & Lusseau)
- Introduction to `R` with `tidyverse` and with application to data analysis: "R for Data Science" (Wickham & Grolemund)