Data Science for Beginners, University of Essex

Day 3: Basic Data Structures

Dr. Howard Liu

12-01-2022

Learning Objectives Today

1. Vectors
2. Matrices
3. Lists
4. The apply functions (why sometimes it makes things faster than loops)

1. Creating a Vector

So far we have worked with single numbers. For example, we have seen how to multiply a single number 3 by another single number 23. (i.e., 3 * 23).

In statistical analysis, we will be working with *sequences* and *tables* of numbers rather than single numbers. They are called vectors and matrices.

A vector is nothing but a collection of numbers. Vectors do not have to be sequential or ordered.

The easiest way to create a vector is to use the `c` function. (short for concatenate or combine).

```
c(2, -1, 0, 9)
```

```
## [1]  2 -1  0  9
```

The above is a vector that binds together four numbers, 2, -1, 0, 9. We can also store this vector into an object.

```
vec.1 <- c(2, -1, 0, 9)
```

Another way to create a vector is to use the seq function (short for sequence). As the name implies, this function creates a sequence from one number to another. It takes at least two arguments, from and to

```
seq(from = 0, to = 10)
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

```
seq(from = 5, to = 1)
```

```
## [1] 5 4 3 2 1
```

There are some additional arguments you may provide, “by” or “length”. The “by” arguments specifies the increments.

For example, if we specify “by = 0.5”, R will create a sequence with an increment of 0.5.

```
seq(from = 0, to = 5, by = 0.5)
```

```
##  [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

The “length” arguments determines the number of elements of the resulting sequence. For example, if we specify “length = 10”, R will create a sequence that has 10 elements (numbers).

```
seq(from = 0, to = 2, length = 10)
```

```
##  [1] 0.0000000 0.2222222 0.4444444 0.6666667 0.8888889 1.1111111 1.3333333
##  [8] 1.5555556 1.7777778 2.0000000
```

```
# For obvious reasons, we cannot use "by" and "length" simultaneously.
# seq(from = 0, to = 5, by = 0.5, length = 10)
```

1-2: Vector operations

Let’s create several vectors and perform some operations on them.

```
x.vec <- seq(from = 2, to = 6)
y.vec <- c(3, 0, 1, -2, 4.3)

x.vec
```

```
## [1] 2 3 4 5 6
```

```
y.vec
```

```
## [1]  3.0  0.0  1.0 -2.0  4.3
```

We can do arithmetic operations with vectors. For example

```
x.vec + 3 # adds three to every number in the vector.
```

```
## [1] 5 6 7 8 9
```

Since the number of elements is the same for x.vec and y.vec, we can do arithmetic operations with them.

What would happen if we try to combine two vectors of different length? It turns out that R will still work, but the output may not be what you'd expect. Moreover, R gives you a warning.

2. Creating a Matrix

Well…It would be cool to generate the virtual reality in the movie **Matrix**. But fellas, we are not there yet as programming beginners.

So what does matrices mean in R? It means that when we have a collection of numbers that are arranged in two dimensions rather than one, we say we have a matrix. We can create one using the matrix function.

```
mat.1 <- matrix(data = seq(from = 1, to = 12), nrow = 3, ncol = 4)
mat.1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

Let's see what the code above does. The first argument "data = …" specifies the contents (numbers) that are stored in the matrix called mat.1. In this particular case, we tell R to create a sequence of numbers from 1 to 12 (1,2,3,4,…,11,12).

The second argument "nrow = …" specifies the number of rows. The third argument "ncol = …" specifies the number of columns.

When using functions that take multiple arguments like this, it is advisable that you provide arguments in separate lines, as follows

```
mat.1 <- matrix(data = seq(from = 1, to = 12),
                nrow = 3,
                ncol = 4)
mat.1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

This is to improve readability of your code. Readers can easily see that you are providing nrow and ncol as arguments for the matrix function, not for the seq function.

Notice that a matrix is nothing but a collection of vectors (and a vector is nothing but a collection of numbers). In other words, we can break down any matrices into vectors.

There are two ways to break down this matrix. First, we can think of this matrix as being made up with three **row vectors**. The first row vector is `1 4 7 10`, the second is `2 5 8 11`, and the third is `3 6 9 12`.

To extract (row or column) vectors, we use **square brackets []**. For example, to extract the second row vector, we write

```
mat.1[2, ]
```

```
## [1]  2  5  8 11
```

Or the third column:

```
mat.1[, 3]
```

```
## [1] 7 8 9
```

Or a specific cell entry:

```
mat.1[2 , 3]
```

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

We can transpose a matrix

```
t(mat.1)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
```

We can also create a matrix by combining multiple vectors. Recall that we have created two vectors, x.vec and y.vec above. We can create new matrices by binding these two together. The `rbind()` function binds multiple vectors, treating them as row vectors.

```
rbind(x.vec, y.vec)
```

```
##       [,1] [,2] [,3] [,4] [,5]
## x.vec    2    3    4    5  6.0
## y.vec    3    0    1   -2  4.3
```

Similarly, the `cbind()` function binds multiple vectors treating them as column vectors.

```
cbind(x.vec, y.vec)
```

```
##      x.vec y.vec
## [1,]     2   3.0
## [2,]     3   0.0
## [3,]     4   1.0
## [4,]     5  -2.0
## [5,]     6   4.3
```

3. Creating a List

Lists are the R objects which contain elements of different types like – numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. List is created using `list()` function.

3-1: Creating a List

Let's create a list containing strings, numbers, vectors and a logical values.

```
list_data <- list("Red", "Green", c(21,32,11), TRUE, 51.23, 119.1)
list_data
```

```
## [[1]]
## [1] "Red"
##
## [[2]]
## [1] "Green"
##
## [[3]]
## [1] 21 32 11
##
## [[4]]
## [1] TRUE
##
## [[5]]
## [1] 51.23
##
## [[6]]
## [1] 119.1
```

Create a list containing a vector, a matrix and a list.

```
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
                  list("green",12.3))
list_data
```

```
## [[1]]
## [1] "Jan" "Feb" "Mar"
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    3    5   -2
## [2,]    9    1    8
##
## [[3]]
## [[3]][[1]]
## [1] "green"
##
## [[3]][[2]]
## [1] 12.3
```

Give names to the elements in the list.

```
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
list_data
```

```
## $`1st Quarter`
## [1] "Jan" "Feb" "Mar"
##
## $A_Matrix
##      [,1] [,2] [,3]
## [1,]    3    5   -2
## [2,]    9    1    8
##
## $`A Inner list`
## $`A Inner list`[[1]]
## [1] "green"
##
## $`A Inner list`[[2]]
## [1] 12.3
```

3-2: Accessing List Elements

Elements of the list can be accessed by the index of the element in the list. In case of named lists it can also be accessed using the names.

```
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
                  list("green",12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Access the first element of the list withe the name.
list_data[1]
```

```
## $`1st Quarter`
## [1] "Jan" "Feb" "Mar"
```

```
# Access the first element of the list w/o the name.
list_data[1][[1]]
```

```
## [1] "Jan" "Feb" "Mar"
```

```
# Extract the values from within the first element
list_data[1][[1]][1]
```

```
## [1] "Jan"
```

Access the list element using the name of the element.

```
list_data$A_Matrix
```

```
##      [,1] [,2] [,3]
## [1,]    3    5   -2
## [2,]    9    1    8
```

3-3: Merging Lists

You can merge many lists into one list by placing all the lists inside one `list()` function.

```
list1 <- list(1,2,3)
list2 <- list("Sun","Mon","Tue")

# Merge the two lists by this way:
list(list1,list2)
```

```
## [[1]]
## [[1]][[1]]
## [1] 1
##
## [[1]][[2]]
## [1] 2
##
## [[1]][[3]]
## [1] 3
##
##
## [[2]]
## [[2]][[1]]
## [1] "Sun"
##
## [[2]][[2]]
## [1] "Mon"
##
## [[2]][[3]]
## [1] "Tue"
```

```
# Or this way
merged.list <- c(list1,list2)

merged.list
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] "Sun"
##
## [[5]]
## [1] "Mon"
##
## [[6]]
## [1] "Tue"
```

3-4: Converting List to Vector

A list can be converted to a vector so that the elements of the vector can be used for further manipulation. All the arithmetic operations on vectors can be applied after the list is converted into vectors. To do this conversion, we use the unlist() function. It takes the list as input and produces a vector.

```
list1 <- list(1:5)

# Convert the lists to vectors.
v1 <- unlist(list1)
v1
```

```
## [1] 1 2 3 4 5
```

4. Apply functions

They are powerful when you want to achieve a simple task but don't want to write a loop which will computationally take longer time than you would want to.

4-1: `apply()` function

apply() takes **Data frame** or **matrix** as an input and gives output in **vector**, **list** or **array**. Apply function in R is primarily used to avoid explicit uses of loop constructs. It is the most basic of all collections and can be used over a matrix. The Syntex is:

```
# apply(X, MARGIN, FUN)
```

Example: We want to calculate the mean of all the columns in a matrix.

```
m1 <- matrix(c(1:10), nrow=5, ncol=6)
m1
```

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

```
a_m1 <- apply(m1, 2, sum) # 1: means rows 2: means columns
a_m1 # output as a vector
```

```
## [1] 15 40 15 40 15 40
```

4-2: `lapply()` function

`lapply()` function is useful for performing operations on a **list object** as an input and returns a **list object** as an output of same length of original set. `lapply()` returns a list of the similar length as input list object, each element of which is the result of applying FUN to the corresponding element of list. Lapply in R takes list, vector or data frame as input and gives output in list. The Syntex is:

```
# lapply(X, FUN)
# Arguments:
# -X: A vector or an object
# -FUN: Function applied to each element of x
```

```
movies <- c("SPYDERMAN","BATMAN","VERTIGO","CHINATOWN")
movies
```

```
## [1] "SPYDERMAN" "BATMAN"    "VERTIGO"   "CHINATOWN"
```

```
movies_lower <-lapply(movies, tolower) # FUN --> convert them all to lower cases
movies_lower
```

```
## [[1]]
## [1] "spyderman"
##
## [[2]]
## [1] "batman"
##
## [[3]]
## [1] "vertigo"
##
## [[4]]
## [1] "chinatown"
```

```
unlist(movies_lower) # We can unlist it back to a vector
```

```
## [1] "spyderman" "batman"    "vertigo"   "chinatown"
```

4-3: `sapply()` function

`sapply()` function takes **list, vector or data frame** as input and gives output in **vector or matrix**. It is useful for operations on list objects and returns a list object of same length of original set. `Sapply()` function in R does the same job as `lapply()` function but returns a vector.

```
# sapply(X, FUN)
# Arguments:
# -X: A vector or an object
# -FUN: Function applied to each element of x
```

Example: We load the default `cars` dataset from the base R

```
dt <- cars
head(dt) # only print out a part of this data
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

```
lmn_cars <- lapply(dt, min) # output is a list
lmn_cars
```

```
## $speed
## [1] 4
##
## $dist
## [1] 2
```

```
smn_cars <- sapply(dt, min) # output is a vector
smn_cars
```

```
## speed  dist
##     4     2
```

Great! We've finished the lecture and you can go to day3 exercise to do some additional practices for today's content.