Department of Economics — Applied Econometrics and Data Analysis
Yale University
Prof. Edward Vytlacil — August 2020

# Working with Data in R

## 1 Data in R

R operates using a variety of data objects including vectors, matrices, arrays, data frames and lists, and it distinguishes different types of data entries such as numeric or character. While some of the distinctions may appear somewhat puzzling or unnecessary at first, misunderstandings about the structure of data objects are among the most common problems for beginners. Therefor, it is helpful to have a basic understanding of R's infrastructure before starting to use it for data analysis purposes.

## 2 Data Types and Objects

R can work with data of different types, called "modes" (or classes). The available modes are called *numeric* (real numbers), *integer* (integers), *logical* (true/false), *complex* (complex numbers) and *character* (strings).

R stores data in objects. Fundamentally, there are two different types of objects: "atomic" objects in which all elements must have the same mode, and "recursive" objects which may contain elements of different modes. The "base" object in R is a vector.

### 2.1 Vectors

A vector in R is an indexed set of values that are all of the same type ("atomic"). The type of the entries determines the mode/class of the vector. A vector is created using the assignment operator `<-`. For example,

```
> x <- 1
> x
[1] 1
```

assigns the value 1 to the vector `x`. To create a vector of more than one element we use the `c()` (concatenate) command.

```
> x <- c(1,2,3,5)
> x
[1] 1 2 3 5
```

R is case sensitive. It also overwrites existing objects. For example, note that by defining the vector x a second time, we have overwritten the initial vector. Elements in vectors and similar data types are indexed using square brackets. For example, if we want to access the fourth component of the vector x, we type

```
> x[4]
[1] 5
```

### 2.1.1 Numeric

The default vector class is numeric. To check the data type of a vector we use the `class()` function. For example,

```
> x <- 5.5
> x
[1] 5.5
> class(x)
[1] "numeric"
```

### 2.1.2 Logical

Often times while working with data it is important to be able to make comparisons between variables or check whether certain conditions hold true. R can check logical expressions and output the logical values `TRUE` or `FALSE`. For example, we can check whether our x variable is greater than 6 by,

```
> x>6
[1] FALSE
```

Similarly, we can ask R to tell us which elements of the vector $(1, 2, 3)$ are equal to one.

```
> x <- c(1,2,3)
> x == 1
[1]  TRUE FALSE FALSE
```

Note that, as with most functions and operators, R performs the operation on each component of the object. We can also ask R to check whether an object is of a certain class using the `is.classname()` command. For example,

```
> is.numeric(x)
[1] TRUE
```

Such logical operations are extremely useful to extract certain entries from a dataset. The logical operators are: $<$, $<=$, $>$, $>=$, $==$, and !=.

Since R considers the entry TRUE as one and FALSE as zero, it is possible to run arithmetic functions on vectors of logical data. For example,

```
> x <- c(TRUE, TRUE, FALSE)
> x
[1]  TRUE  TRUE FALSE
> sum(x)
[1] 2
> mean(x)
[1] 0.6666667
```

### 2.1.3 Character

A character object is used to represent string values in R. In order to tell R that an entry should be considered as string, we have to put it in quotes. For example,

```
> x <- c("Hello", "World")
> x
[1] "Hello" "World"
```

We can also convert objects into character values using the as.character() function:

```
> x <- c(1,2,3)
> x
[1] 1 2 3
> x <- as.character(x)
> x
[1] "1" "2" "3"
```

### 2.1.4 Missing Values

In R, missing values are represented by the symbol NA (not available), irrespective of whether data is numeric or character. For example,

```
> x <- c(1,2,3,NA)
> x
[1]  1  2  3 NA
```

Often times running arithmetic functions on objects including missing values yield missing values. However, most functions in R offer options for dealing with missing values. For example,

```
> x
[1]  1  2  3 NA
> mean(x)
[1] NA
> mean(x,na.rm=TRUE)
[1] 2
```

### 2.1.5 Vector Arithmetic

Numeric vectors can be used in arithmetic expressions, in which case the operations are performed element by element to produce another vector. For example,

```
> x <- c(1,2,3)
> y <- c(4,5,6)
> x+1
[1] 2 3 4
> x+y
[1] 5 7 9
> x*y
[1]  4 10 18
```

The elementary arithmetic operations are $+, -, *, /,$ ^. See also `log()`, `exp()`, `sqrt()` etc., which are again applied to each entry.

Note that R even performs arithmetic operations on vectors that do not have the same length (This is why a command like `x+1` produces output). In doing so, it uses a ”recycling rule”, that is, it repeats the values of the shorter object. For example,

```
> x <- c(1,2)
> y <- c(1,2,3,4)
> x+y
[1] 2 4 4 6
```

This default can lead to serious mistakes. As a consequence, you should always be aware of the length of the vectors that you are working with in R. This can be done using the `length()` command.

```
> length(x)
[1] 2
> length(y)
[1] 4
```

## 2.2 Matrices

Matrices in R are data objects in which the elements are arranged in a two-dimensional rectangular layout. Just as vectors they are ”atomic” in the sense that they can only contain entries of the same class. Most of the time matrices with numeric entries are used. To create a Matrix, we use the `matrix()` function. For example,

```
> M <- matrix( c(1,2,3,4,5,6), nrow = 2, ncol = 3, byrow = TRUE)
> M
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

As you can see from the example, the basic syntax for creating a matrix in R is `matrix(data, nrow, ncol, byrow)`, where `data` is the input vector, `nrow` is the number of rows of the matrix, `ncol` the number of columns, and the logical clue `byrow` indicates how the matrix is filled. If it is set to `TRUE`, the matrix is filled by row, if it is set to `FALSE` or left blank, the matrix is filled by column.

Alternatively, a matrix can be created by combining vectors of the same type using the `rbind()` and `cbind()` commands. `rbind()` treats the input vectors `x` and `y` as row vectors, while `cbind()` treats them as column vectors. For example,

```
> x <- c(1,2,3)
> y <- c(4,5,6)
> M <- rbind(x,y)
> M
  [,1] [,2] [,3]
x    1    2    3
y    4    5    6
```

and

```
> M <- cbind(x,y)
> M
     x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

Just like vectors, elements of matricies are indexed using square brackets. For example, if we want to access the element in the second column of the third row, we type

```
> M[3,2]
y
6
```

Besides the corresponding element of the matrix, the output also tells us that the element is coming from the vector `y`.

We can check the row and column dimensions of a matrix using the `dim()` command. For example,

```
> dim(M)
[1] 3 2
```

tells us that the matrix `M` has three rows and two columns. Using matrices of the approriate dimensions, R can also perform matrix addition, multiplication and division.

## 2.3 Arrays

While matrices are confined to two only dimensions, arrays in R are data objects which can store data in more than two dimensions. Just like matrices, arrays can only contain data of the same mode. We create arrays using the `array()` function. The following example creates an array of two matrices each with 2 rows and 2 columns.

```
> x <- c(1,2,3,4)
> y <- c(5,6,7,8)
> a <- array(c(x,y),dim=c(2,2,2))
> a
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

Accessing an element of an array requires specifying all dimensions. In our three dimensional example, we could acess the "last" component of our array by

```
> a[2,2,2]
[1] 8
```

If we leave out one dimension, R outputs all entries in that dimension. For example,

```
> a[1,1,]
[1] 1 5
```

outputs all entries of the third dimension of the array are in the first position in the first two dimensions.

## 2.4 Dataframes

A dataframe in R is a tabular data object. More precisely, it is a list of vectors of equal length. Most econometric data will be in the form of a dataframe because of the objects ability of storing data of different types. Unlike vectors and arrays, dataframes are "recursive" objects in the sense that each column can contain data of different modes. For example, the first column can contain numeric values, while the second column contain characters etc. To create a dataframe using a set of vectors, we use the `data.frame()` command. For example,

```
> age <- c(19, 21, 30)
> name <- c("A", "B", "C")
> student <- c(TRUE, TRUE, FALSE)
> df <- data.frame(age, name, student)
```

creates the matrix-like structure

```
> df
  age name student
1  19    A    TRUE
2  21    B    TRUE
3  30    C   FALSE
```

We can access single columns of a dataframe using the `$` operator. For example,

```
> df$age
[1] 19 21 30
```

outputs the age vector. Individual entries can be accessed using square brackets, just like in the matrix case. A dataframe can be expanded by a column using the `cbind()` command. For example, if we want to add a fourth column `gender` to the data.frame, we type

```
> gender <- c("M","F","M")
> df <- cbind(df,gender)
> df
  age name student gender
1  19    A    TRUE      M
2  21    B    TRUE      F
3  30    C   FALSE      M
```

We can also remove columns. For example, with gender now as the fourth column of the data.frame, we can remove gender from the data.frame as follows:

```
> df <- df[,-4]
> df
  age name student
1  19    A    TRUE
2  21    B    TRUE
3  30    C   FALSE
```

## 2.5 Lists

A list in R is an object which can contain many different types of elements, like vectors, matrices, dataframes, functions and even another list inside it. A list is created using the `list()` function. For example,

```
> l <- list("String", TRUE, df)
> l
[[1]]
[1] "String"

[[2]]
[1] TRUE

[[3]]
  age name student gender
1  19    A    TRUE      M
2  21    B    TRUE      F
3  30    C   FALSE      M
```