Description of data types

Roberto Molowny-Horas

April 24-25, 2023

We will begin by describing variable types and basic mathematical operations in R. This way, you will start becoming acquainted with the at-times peculiar R syntax.

Variable types

In R a number can be stored as an integer, double or complex. If nothing is said, R assumes that the number is a double (and, thus, it needs 64 bits to store it). Character variables can also be stored as strings. To print the content of the variable on screen we will use the **print** function.

```
a <- 2                                  # A double (aka "real") number.
a <- 3L                                 # An integer number.
a <- complex(real = 2,imaginary = 3)    # A complex number. Probably of little use to
us.
a <- TRUE                               # A logical (TRUE or FALSE).
a <- "Hello world!"                     # A string.
print(a)
```

As you may have seen, to assign a value to a variable in R we use an arrow $\leftarrow$, but we may also use the equal sign $=$ as in many other computer languages. It is a question of “style”, but it really does not matter. Use $=$ or $\leftarrow$ indistinctly (but use $\leftarrow$ if you want to stick to R-style rules).

```
b = 3
b <- 3
b = "Hello world!"
b <- "Hello world!"
```

Vectors, matrices, arrays…

Arrays are data structures that can have any dimensions. In the example above we have assigned a single number or string to a variable, but in R (and in many other computer languages) you can also assign a 1-D (a vector), 2-D (a matrix) or generic N-D arrays. To create a vector in one step we can use the concatenate **c** operator.

```
a <- c(3, 4)
a <- c(3L, 4L)
a <- c("Hello", "as")
a <- c("Hello", 3)
```

Notice that vector, matrices or arrays always store data of the same type. In the latter example, however, we wanted to concatenate one string and one number and assign the result (a 1D array, i.e. a vector) to a variable called a. Since all elements should have the same type, R internally changed the number 3 to the string “3” (the opposite, in fact, would not make sense) and created a string vector containing the string “Hello” and “3”. Therefore, these data structures are not well suited to store data of different types. The solution would be to use data.frames or lists, as we shall see later.

Matrices (i.e. 2D arrays) are very simple to construct.

```
a <- matrix(runif(12, min = -1.5, max = 9.2), 3, 4)
print(dim(a))
print(a[2,4])
```

Finally, arrays of any dimension can be created with function **array**.

```
a <- array(runif(24, min = -1.5, max = 9.2), dim = c(3, 4, 2))
print(dim(a))
```

Above we have used one of the many statistical built-in function, **runif**, which can be used to create an uniform distribution of points (24 in this case) between -1.5 and 9.2. Check the help pages for the set of statistical distribution available in R.

```
# ?Distributions
```

NB! Help pages for R functions can be accesses by using the question mark, e.g. ***?mean****. Using a double question mark will start a more general (and far slower) search about the term and related issues. Thus,* ***?sd*** *will show up the help page for the* ***sd*** *(i.e. standard deviation) built-in function, whereas* ***??sd*** *will show many different help pages related to functions and packages that contain the word “sd”. On the other hand,* ***?regression*** *will probably turn out nothing on your RStudio session, whereas* ***??regression*** *will likely throw many results.*

Data structures with pre-set values often come in handy.

```
a <- 1:10
a <- seq(1, 10, by = 1)            # "seq" is a very useful function.
a <- seq(-3.5, 7.1, by = 1)
a <- seq(-3.5, 7.1, length = 11)

a <- numeric(4)
a <- integer(5)
a <- character(6)
a <- factor(numeric(2))
a <- matrix(NA, 2, 3)              # NA's too!
a <- rep(3, times = 4)
```

Factors

Factors are categorical variables that can only take on a given number of values. They are usually employed for e.g. classification or regression purposes. Factor variables are caracterized by values, levels and labels.

```
fa <- factor(c("Blue", 3),levels = c("Blue", "Red", 3, 4, 5), labels = LETTERS[1:5])
#
print(fa)

# Add one element
fa[3] <- "B"
print(fa)

# Try!
fa[4] <- "K"
```

Sometimes we may want to convert factors to numbers. If that is the case, try:

```
a <- factor(runif(3))            # Create a factor with random numbers (but elements i
n "a" are not numeric!).
a <- as.numeric(levels(a)[a])    # "as.numeric(as.character(a))" will also work.
```

Strings

There are many functions in R that are aimed at handling strings.

```
# A character variable.
ch <- "CREAF is a research center"
print(length(ch))
print(nchar(ch))

# Extracting and finding substrings
print(substr(ch, 10, 16))
print(gregexpr("ese", ch))

# Add characters to a string.
print(paste(ch, "outside Barcelona"))
print(paste0(ch, "outside Barcelona"))    # Notice the difference!
print(paste0("A-", 1:4))
```

Data frames.

As we saw in the example above, arrays (of any dimension) can only contain elements of the same type, be it integer, double or strings. However, it is often the case that we are in the need of storing data sets of varied type, like e.g. numbers and strings or factors. Data frames allow us to do just that. They are like tables (think of an Excel worksheet) where each column corresponds to a different variable of (maybe) different type.

```
df <- data.frame(Name=c("Peter","Jordi","Carlos"),Age=c(25L,31L,34L),Height=c(1.74,1.
80,1.81))
print(df)
print(df$Name)
print(df["Name"])
print(df[["Name"]])
```

By default, a character element in a data frame is assumed not to be a factor. This can be changed by setting the logical argument **stringsAsFactors**=TRUE (before R 4.0.0, it was the opposite).

Data frames can be access by number of row or column or, alternative, by name of row or column. Be aware that, depending on the way you access the elements of a data.frame, the type of the output variable may vary.

```
df <- data.frame(Name=c("Peter", "Jordi", "Carlos"),
                 Age = c(25L, 31L, 34L),
                 Height = c(1.74, 1.80, 1.81),
                 row.names = c("First", "Second", "Third"))

# Accessing rows.
print(df["First",])
print(df[1,])

# Accessing columns. Notice the differences!
print(df["Name"])        # This yields another data.frame
print(df$Name)           # This yields a vector.
print(df[, 1])           # And this.
print(df[["Name"]])
```

Exercise 1. Create by hand a vector of numbers, but stored as strings (hint: use the concatenate function **c()***), i.e. with numbers as text, and then convert it to a numeric vector with function* **as.double()** or **as.numeric()**. *Vector should have 5 elements.*

*Exercise 2. Add by hand a new column “Weight” to the data.frame “df” created above containing the weight, in Kg, that you have assigned to each person. Then, calculate a new column named BMI that yields the Body Mass Index of each individual, defined as BMI=Weight/Height^2.

Lists

Lists are similar to data frames, but they are even more flexible when storing data. They can accept data in any format (like data frames). Unlike data frames, however, list elements do not necessarily have the same length.

```
l <- list(a = c(1, 3, 4, 5, 3),
          v = c("T", "h", "i", "s", " ", "i", "s", " ", "h", "e", "a", "v", "e", "n",
"!"))
print(l)
print(length(l))
print(dim(l))
cat(l[[1]], "\n")
cat(l[[2]], "\n")
cat(l[[2]], "\n", sep = "")
print(l$v)
print(l[["v"]])
print(l["v"])         # Notice the difference.
```

What makes lists so interesting is that they can be used to store almost anything in any format. For example, we can make a single list containing several of the data structures created above. Also, each element in the list can be given a name, which comes in handy to access those elements.

```
a <- array(runif(24, min = -1.5, max = 9.2), dim = c(3, 4, 2))
fa <- factor(c("Blue", 3),levels = c("Blue", "Red", 3, 4, 5), labels = LETTERS[1:5])
l <- list(a = c(1, 3, 4, 5, 3),
          v = c("T", "h", "i", "s", " ", "i", "s", " ", "h", "e", "a", "v", "e", "n",
"!"))
df <- data.frame(Name = c("Peter", "Jordi", "Carlos"),
                 Age = c(25L,31L,34L),
                 Height = c(1.74, 1.80, 1.81),
                 row.names = c("First", "Second", "Third"))
big_list <- list(a, fa, l, df)
names(big_list) <- c("My array", "My factor", "My simple list", "My data frame")
print(big_list[["My simple list"]])
print(big_list[[3]])
```

Simple maths and vectorization

We can use many built-in statistical functions that are available in R. For an exhaustive list of functions, check **library(help = "stats")** or look it up on the internet. Those functions (or most of them) are vectorized and can be used on variables containing single numbers or N-D arrays. A vectorized function acts on all elements of a variable without the user having to specify one element at the time.

```
a <- c(1, 3, 4, 5, 3, 34, 6)
print(c('Mean' = mean(a), 'SD' = sd(a)))
a <- a^2
print(a)
b <- c(3, 5, 7, 9, 9, 9, 8)
b <- b/5
ab <- a+b-5
ab <- log(a+b, base = (a+b)/5)
print(ab)
```

In arithmetics we can use the typical operators +, -, *, /, ^ to add, substract, multiply, divide compute power of a number. Matrix multiplication is %*%. Logical operators are:

- **<** strictly smaller than
- **<=** smaller than, or equal to
- **==** equal to
- **>=** larger than, or equal to
- **>** strictly larger than
- **!=** not equal

All these operators can also be vectorized.

```
a <- 1:10
print(a^2)
print(a > 4)
print(letters[1:10] == "g")
print(LETTERS[1:10] != "G")
```

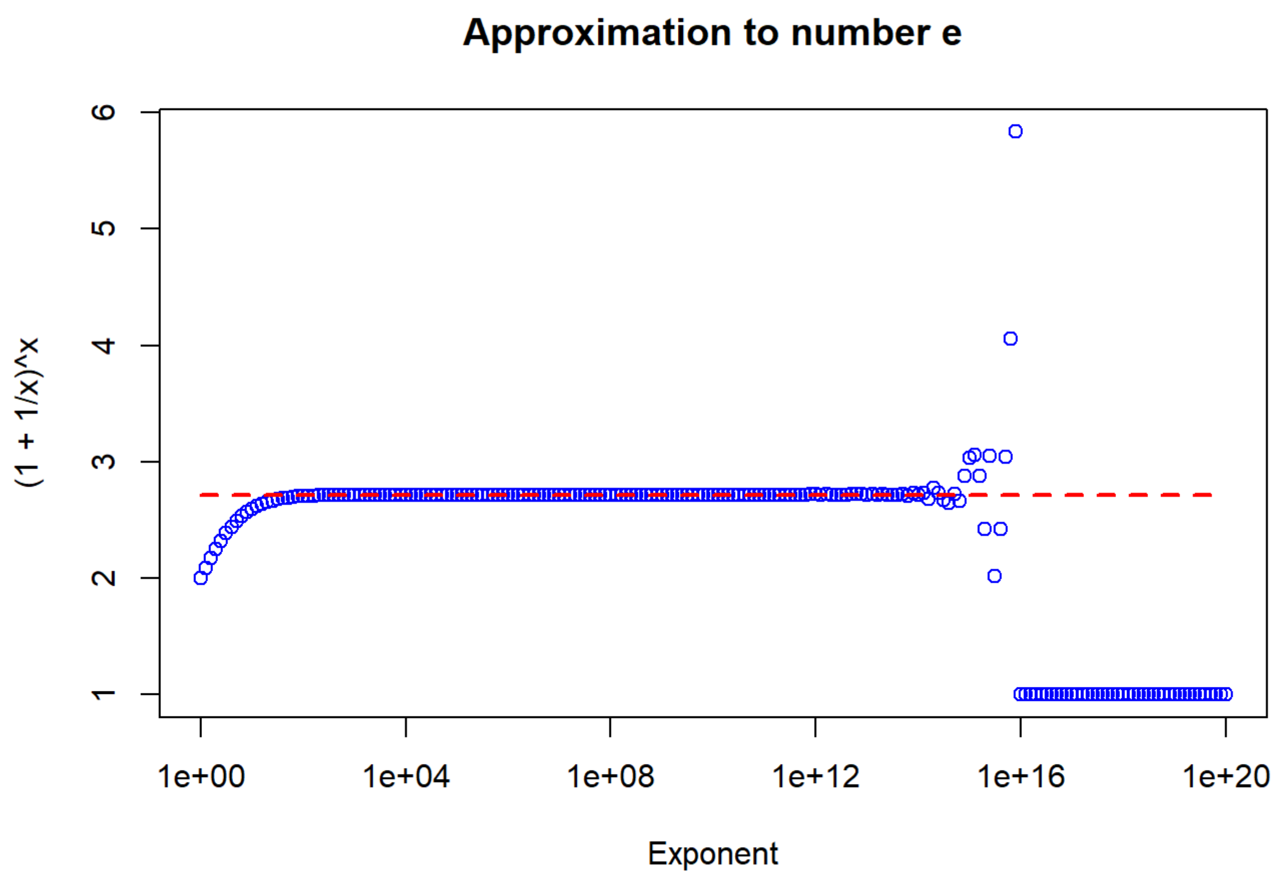
In addition we can use boolean AND (&) and OR(|).

```
print(2>=3 | 3<4)
print("a"=="a" & !(1>2))
```

A note on machine precision and number representation

Real numbers in R in particular, and in digital computers in general, are stored as double-precision numbers. To understand which are the numerical characteristics of your machine, you can check the **.Machine** variable. This number will approximately tell us the limit to the precision with which mathematical operations take place in R. To visualize this limitation in accuracy, we can calculate for example the irrational number **e** (=2.7182818284590451) as a limit.

```
print(.Machine$double.eps)
x <- 10^seq(0, 20, by = .1)
plot(x, (1+1/x)^x, main = "Approximation to number e", xlab = "Exponent",
     log = "x", pch = 1, col = "blue")
points(c(min(x), max(x)), c(exp(1), exp(1)), type = "l", lty = 2, lwd = 2, col = "re
d")
```


Approximation to number e
(1 + 1/x)^x
1e+00
1e+04
1e+08
1e+12
1e+16
1e+20
Exponent