Getting Started with R

Marko Šamara

Reference:

- Interview with prof. John Chambers (the interviewer is prof. Trevor Hastie of Stanford University): https://www.r-bloggers.com/john-chambers-recounts-the-history-of-s-and-r/
- Ross Ihaka, *R: Lessons Learned, Directions for the Future*; JSM Proceedings, 2010 https://www.stat.auckland.ac.nz/~ihaka/downloads/JSM-2010.pdf
- R Core Team: https://www.r-project.org/contributors.html
- Info about assignment operators can be found at this page
- Blog post by Colin Fay about the assignment operator `<-`: https://colinfay.me/r-assignment/

A Brief History of R

R was created as a software whose purpose eventually became implemention of programming language S for free. The S language and statistical environment was created by a statistician, prof. John Chambers in 1976, (in collaboration with Richard Becker and Allan Wilks at earlier stage), while working at Bell Labs. After couple of modifications and improvements, and with fairly low license rates to universities and research labs, S became popular in academic community.

Inspired by S statistical system, the R software was created by professors Ross Ihaka and Robert Gentleman at University of Auckland, New Zeeland. The two professors later became known as *R and R of Statistics*. Their initial intention was to create a free, open source statistical system. After mimicking some solutions from S, they realized that due to already existing popularity of S they would quickly attract more people to use their software, if they create an improved clone of S, or as Ross Ihaka puts it (according to prof. Chambers), "*a language that is not unlike S*". Chambers was supportive of this idea. The development started in 1995-6 and soon an initial version was released. In the process of creating R, S language was altered and improved, but most of the later version of S code can be understood by R. The first stable version of R was released in 2000. In 1997, Ihaka and Gentleman transfered the control of development and standardization of R to a team of programmers, called R Development Core Team, of which they are members from the beginning. Also, Chambers later became a member of the Core Team, too.

Hello, World!

It is in tradition of programming (can be seen in books on various programming languages) to have the following as the very first command in the new programming language you want to learn.

```
print("Hello, World!")
```

```
[1] "Hello, World!"
```

Note: ignore the symbol `[1]` that appears in the outputs until we get to the section about creating vectors - it will be justified there.

Assignment Operators = and <-

If we want to store, say, number 3.14 in computer using R, we can create an object/variable `x` and put the value 3.14 into it. This is done either by typing

```
x <- 3.14
```

or

```
x = 3.14
```

in the console. We say that `<-` and `=` are assignment operators. The aforementioned two ways of assigning 3.14 to the variable `x` are equivalent and have the same effect, although the assignment operators `<-` and `=` are not quite the same. You can see both `<-` and `=` being used in books about R, blogs and forums where you can get help on R. Before we move on to further R syntax, it is worth comparing `<-` and `=`, since the existence of two assignment operators often confuses programmers coming to R from other languages that have only one way to assign a value to a variable.

The assignment operator `<-` reminds us of an arrow symbol, as if it says "put 3.14 into the object `x`". This is exactly the reason why this symbol is used.

Historically, the operator `<-` was used in S language as the assignment operator. It was inspired by APL language, which had `<-` symbol on APL keyboard, and which was used for this assignment purpose. At that time (1970's) keyboards were not uniform as nowadays and other languages developed on other keyboards used other symbols for assignment operator. Even more importantly, most of them did NOT use `=` for assingment. Instead, `=` was often used for testing equality of two variables (in most languages used today, including R, testing equality is done by using `==`).

R inherited `<-` from S, and `=` was introduced as an alternative. However, there is a subtle difference between the two operators, although in practice, it doesn't come into play very often. We explore the difference here.

```
x = 1:4
print(x)
```

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

```
sum(x)
```

```
[1] 10
```

As we can see, the command `x = 1:4` created a vector of first four natural numbers, and the command `sum(x)` summed up its entries. Let us now remove/delete the `x` variable from the working environment.

```
rm(x)
```

Now, let's try to print `x` and see what happens. We can use `print(x)`, but it actually suffices just to type `x`:

```
x
```

```
Error in eval(expr, envir, enclos): object 'x' not found
```

So, we were notified about an error, because the printing of `x` could not be executed. As expected, since `x` was removed by the previous command, it doesn't exist anymore. Consider now the following.

```
sum(x = 1:4)
```

```
[1] 10
```

As we can see, the command `sum(x = 1:4)` gives us the value 10 ($= 1 + 2 + 3 + 4$), but did we also create vector `x`?

```
x
```

```
Error in eval(expr, envir, enclos): object 'x' not found
```

So, `x` was NOT created. Another way to check this would be by listing all the variables defined in the global environment, using `ls()` command:

```
ls()
```

```
character(0)
```

This means the environment has no variable defined. Therefore, in the command `sum(x = 1:4)` symbol `x` was just used to set up a vector $[1, 2, 3, 4]$, whose sum (i.e. the sum of its components) was then computed.

Let's alter the command `sum(x = 1:4)` to `sum(x <- 1:4)` and see what happens:

```
sum(x <- 1:4)
```

```
[1] 10
```

```
ls()
```

```
[1] "x"
```

```
x
```

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

As we can see, now vector `x` does exist. Therefore, unlike `sum(x = 1:4)`, the command `sum(x <- 1:4)` not only computes the sum of the first four natural numbers, but it also creates variable `x`, and assigns it a vector of first 4 natural numbers. This is the main difference between assignment operators `<-` and `=`.

This diference is summarized in the documentation for assignment operators (which can be obtained by typing `?assignOps` in the console), where it says: "*The operators <- and = assign into the environment in which they are evaluated. The operator <- can be used anywhere, whereas the operator = is only allowed at the top level (e.g., in the complete expression typed at the command prompt) or as one of the subexpressions in a braced list of expressions.*"

Nevertheless, the style of the code `sum(x <- 1:4)` is not the best one (at least in my personal opinion). It seems it hides the creation of variable `x`. Instead, we can write

```
x <- 1:4
sum(x)
```

```
[1] 10
```

This seems to be a cleaner code. Using the operator `=` and this cleaner style, we could also write

```
x = 1:4
sum(x)
```

```
[1] 10
```

and the above two code chunks give us the same result. So, using this cleaner programming style, the difference between operators `<-` and `=` becomes almost invisible, and with this style of coding the two operators are practically equivalent, interchangeable.

Remark: You should have in mind that any coding style is a matter of opinion. **In this course, we will use `=` as the assignment operator** (which also seems to be the choice of R creators, Ihaka and Gentleman[1]). Only occasionally, when I am to create a complex object, such as function, will I use `<-`. For my justification of using `=` and more on assignment operators, see the appendix at the end of this file.

[1] Ihaka and Gentleman used `<-` earlier, though; On Ihaka's webpage papers written before 2003ish used `<-`, while those from 2003 and later use `=`.

Built-in Functions

Functions are code chunks (sometimes very long) that can be used as a part of your code. You can build your own functions, but there are also functions already existing in R, i.e. already built in. In fact, we already used some built-in functions: `sum()`, `rm()`, `ls()`. To invoke a function, you use its name, followed by paretheses. Inside parentheses, you usually specify values of parameters/arguments of the function. For example, in the command

```
sum(x = 1:4)
```

the input parameter/argument is the vector `x`. Many functions have multiple input parameters, and some parameters have default values.

Example. We will use function `mean()` to compute arithmetic mean of the following two vectors `x` and `y`:

```
x = 1:4
x
```

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

```
y = c(1,2,3,4,NA)
y
```

```
[1]  1  2  3  4 NA
```

In R, symbol `NA` is a reserved word that represents a missing value (**N**ot **A**vailable).

We now apply the function `mean()` to compute means of these vectors.

```
mean(x)
```

```
[1] 2.5
```

```
mean(y)
```

```
[1] NA
```

The mean of `x` is as expected. However, the mean of `y` is `NA` (i.e. missing value), because R didn't know how to interpret `NA` in $\frac{1+2+3+4+\texttt{NA}}{5}$ =??? In fact, apart from vector as an input parameter/argument, the function `mean()` has another argument, called `na.rm` (**rem**ove `NA`'s), which by default is set to `FALSE`. So, by default, if you don't specify the value of the argument `rm.na` and if your vector contains missing data, the missing data will not be removed from computing the mean. If we change the default value of `rm.na` to `TRUE` value, missing data will be ignored when computing the mean of the vector `y`:

```
y
```

```
[1]  1  2  3  4 NA
```

```
mean(y, na.rm=TRUE)
```

```
[1] 2.5
```

How do you know what parameters/arguments certain function have, whether they have default values and what the default values are? You should NOT know this by heart, of course. Instead, you use R documentation, as discussed in the next section.

Getting Help

You can look for documentation of a desired function within the console by typing `?` in front of the name of the function in the console. For example, if you type

```
?mean
```

you see documentation about `mean()` on the right side in the RStudio window. There, you can see that apart from argument `x`, function `mean` also has parameters/arguments `trim` (set equal to 0 by default) and `na.rm` (set equal to `FALSE` by default.). The ellipsis `...` (also read as "dot-dot-dot") is another argument, which represents arbitrary list of potentially more arguments, and which will be discussed later.

Instead of `?mean`, we could also get help by typing `help(mean)`, which has the same effect as `?mean`.

The problem with this type of help is that you need to know the exact name of the function, which is not always the case. However, if you know at least first couple of letters, the auto complete feature of RStudio IDE will give you suggestions of functions starting with letters you typed in.

Another way to get help is to search online for answer to a question you have. For example, you can google "how to find mean of a vector in R". You will see a lot of suggestions. Google Chrome and other browsers are very good in giving you links with good suggestions where to look at the answer (if you formulate question in a right way), as they use various machine learning techniques to improve recommended correct or at least close answers. In general, the more you google questions/answers about R, the better suggestions you get. Your questions for googling should not include some specific values. They should have more generic nouns instead. For example, if you need to find a mean of a vector `x = 1:4`, don't type `how to find mean of 1:4`. Replace the concrete vector `1:4` by the noun vector. Also, always include "R", or "[R]" or "in R", in your query (usually at the and of the query), so that your browser "knows" which language you need the answer for.

You will often be suggested links from Stack Overflow https://stackoverflow.com, which is nowadays the most popular site for programming questions (as well as other disciplines). You can post a question there by yourself, but there are so many questions already posted and answers, that it is highly likely that whatever question you have, someone else already posted it and some programmer was kind enough to answer it. You just need to practice formulating questions in a good way, so that your browser recommends good links for you. Apart from stackoverflow.com, there are other web sites, blogs and forums, some specialized just for R, which you will come accross while searching for answers. Other two most notable are RStudio Community https://community.rstudio.com and R-help mailing list https://stat.ethz.ch/mailman/listinfo/r-help.

Comments

You probably know about comments in programming. They are parts of a code that programming language (in this case R) completely ignores - does not execute. Their most common purpose is to help programmer clarify what certain parts of a code do. For example:

```
x = 1:4

# defining vector y, which consists of squares of entries of the vector x
y = x^2

y  # printing vector y
```

```
[1]  1  4  9 16
```

```
# deleting (i.e. removing) vector x from the global environment - don't need it anymore
rm(x)
```

In R, as you can see from this example, comments start with `#`. Everything after `#` in the corresponding line will be ignored and will not be executed. Although only one `#` is sufficient, for clearity reasons, many R programmers use `##` for comments.

It takes some experience for a programmer to get a good feeling about when to comment and how much. In general, the more comments, the better. Writing comments should be talking to yourself from future, explaining what you had in mind when programing a particular code chunk or snippet. While writing a code, you are all into it, and everything seems obvious. However, as soon as you get to writing some other code, or other part of the same program, you quickly forget what was going on in previous code chunks. When you

get back to your earlier code (for example, to check that part of the code, or to copy an idea), you may be surprized and confused, not understanding why you wrote the code in a certain way. That is why writing a comment, explaining the details, are very useful.

Comments are also useful when testing a code chunk. It is often convenient to turn off/on certain lines of code to test various parts of the code. This turning off/on is done by commenting (i.e. typing `#` in front of the code line) and uncommenting (i.e. removing `#` in fornt of the code line).

Notifications (Messages, Warnings and Errors)

When you run an R code, as a part of output you may get some notification. There are three types of notifications in R:

- message
- warning
- error

Message is a type of notification whose purpose is to give you some information about the execution of a code or of a function or a package used in the code. This information can often be ignored as not so essential or useful.

Warning notification is printed if R thinks there might potentially be a problem or you might need to be cautious about some part of the code. However, even if R thinks there might be a problem, that doesn't necessarily mean you should be worried about. Warnings are not so rarely about some benign things that really are not an issue, but sometimes can be very useful.

Error notification is printed when the code stops (gets halted or execution doesn't even start) because R doesn't understand something due to wrong syntax, or a certain variable is needed but is not defined, or a certain package is to be used but is not loaded or installed. Unlike messages and warnings which you can not so rarely ignore, the error notification CANNOT be ignored, since the code cannot be (entirely) executed. It takes some practice and experience to understand certain error notifications and with this info to fix the code. Common types of errors include texts like "... is not defined", or "... does not exist", "... is not found". This means the object/variable/package whose name appears right before those words was not defined, or loaded. These types of error notifications appear frequently, but there are many other types. Sometimes the notifications are not so clear, especially to an unexperienced user.

Basic Types of Objects (Atomic Objects)

Four basic types of objects/classes (called "atomic" objects):

- character variable (string),
- real variable (real number, floating number, double precision),
- integer variable,
- logical (bolean; TRUE/FALSE)

While in mathematics integers are real numbers, in computer science, integer variables are NOT a type of real (double precision) variables. There are some R functions that can be applied to integers, but not to doubles (i.e. real variables). Also, both integers and doubles are called numeric objects.

Let's see some examples of these 4 atomic object types.

```
x1 = 'This is a string variable'
x1
```

```
[1] "This is a string variable"
```

```
typeof(x1)  ## show the type of the variable x1
```

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

Remark: Character variable is always surrounded by quotation marks. You can use either a pair of single quote marks or a pair of double quote marks, it doesn't matter.

```
x2 = 26.4  ## real, double precision variable
x2
```

```
[1] 26.4
```

```
typeof(x2)
```

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

Let's try to define 26 as an integer.

```
x3 = 26
typeof(x3)
```

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

We see, by default, `26` was treated as a real number (double), not as an integer variable. If you want to enforce `26` to be an integer, you can either type `L` after the number

```
x3 = 26L
typeof(x3)
```

```
[1] "integer"
```

or else, use command `as.integer()` for conversion into integer type:

```
x3 = as.integer(26)
typeof(x3)
```

```
[1] "integer"
```

Logical variables have either value `TRUE` or `FALSE`. The `TRUE` value is also denoted as `T`, but `True`, `true` or `t` are **not** valid logical values. Similarly, `F` is the same as `FALSE`, but `False`, `false` and `f` are not valid logical values. Example:

```
x4 = F
x4
```

```
[1] FALSE
```

```
x4 == FALSE  ## comparing variable x4 with value FALSE
```

```
[1] TRUE
```

```
x4 == false  ## attempt to compare x4 with value false
```

```
Error in eval(expr, envir, enclos): object 'false' not found
```

Vectors

A vector is an array of atomic objects of the same type. Here are some examples.

```
x = c(2, 3, -5, 7, -2)  #'c' stands for 'concatenate', or 'combine'
x
```

```
[1]  2  3 -5  7 -2
```

We use `x[4]` to refer to the 4th entry of the vector `x`:

```
x[4]
```

```
[1] 7
```

```
x[1:3] ## taking first three entries from vector x
```

```
[1]  2  3 -5
```

```
x[c(1,3)] ## taking 1st and 3rd entry from x
```

```
[1]  2 -5
```

```
x[-3] ## taking out the 3rd entry from x
```

```
[1]  2  3  7 -2
```

```
x[-c(1,3)]  ## taking out 1st and 3rd entry
```

```
[1]  3  7 -2
```

```
x[x>0] ## taking all positive values of x
```

```
[1] 2 3 7
```

```
x>0
```

```
[1]  TRUE  TRUE FALSE  TRUE FALSE
```

```
table(x>0) ## summary, how many positive values, how many negatives
```

```
FALSE  TRUE
    2     3
```

```
3 %in% x  ## logical statement that 3 belongs to x (true)
```

```
[1] TRUE
```

```
4 %in% x ## logical statement that 4 belongs to x (not true)
```

```
[1] FALSE
```

```
x
```

```
[1]  2  3 -5  7 -2
```

```
x^3 ## (raise to a power, component-wise)
```

```
[1]    8   27 -125  343   -8
```

```
y = c(1, 2, 0, 5, -1)
x + y
```

```
[1]  3  5 -5 12 -3
```

```
x * y
```

```
[1]  2  6  0 35  2
```

```
x %*% y ## matrix multiplication (in this case dot product)
```

```
     [,1]
[1,]   45
```

- another commonly used object is a matrix (think of it as a table of real/integer numbers)

```
M = matrix(c(1,2,3,4,5,6),2,3)
M
```

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

Creating Vectors of Various Types

Important: While you can create vectors of various types, each vector has to have all components/entries of the same type.

```
x1 = seq(from=1,to=7,by=2)
x1
```

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

```
x2 = seq(from=2, to=109, by=2)
x2
```

```
 [1]   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38
[20]  40  42  44  46  48  50  52  54  56  58  60  62  64  66  68  70  72  74  76
[39]  78  80  82  84  86  88  90  92  94  96  98 100 102 104 106 108
```

We now finally see what the symbol `[1]` means in all outputs we had earlier. It marks the index of the component of an output vector that is right next to the symbol `[1]`. In the case of this vector `x2`, the 1st entry of `x2` is 2. In the same fashion, we see that the 18th entry of the vector `x2` is 36, and the 35th entry is 70.

Also, note that the vector `x2` stopped with 108 as its last entry, since we don't get 109 when starting from 2 and jumping by 2.

```
x3 = c(2.3, -1.) ## numeric (double precision)
x3
```

```
[1]  2.3 -1.0
```

```
x4 = c(TRUE, FALSE) ##logical
x4
```

```
[1]  TRUE FALSE
```

```
x5 = c(T,F) ##logical, the same thing as in previous line
x5
```

```
[1]  TRUE FALSE
```

```
x6 = c("you", "me", "everybody")
x6
```

```
[1] "you"       "me"        "everybody"
```

```
str(x6)
```

```
 chr [1:3] "you" "me" "everybody"
```

```
x7 = paste(x6, collapse=", ")
x7
```

```
[1] "you, me, everybody"
```

Initiating vector (pre-allocating memory)

```
foo = vector('integer',length=10)
foo
```

```
 [1] 0 0 0 0 0 0 0 0 0 0
```

```
bar = vector('numeric',length=10)
bar
```

```
 [1] 0 0 0 0 0 0 0 0 0 0
```

```
vector('logical',length=10)
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
vector('character',length=10)
```

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

Some objects may have attributes/features. For example, we can talk about length of `x`.

```
M = matrix(c(1,2,3,4,5,6),nrow=2,byrow=T) #compare this with the above matrix M
M
```

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

```
attributes(M)
```

```
$dim
[1] 2 3
```

```
class(M)
```

```
[1] "matrix"
```

```
typeof(M)
```

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

```
x = 3:6
x
```

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

```
attributes(x)
```

```
NULL
```

```
class(x)
```

```
[1] "integer"
```

```
typeof(x)
```

```
[1] "integer"
```

We see R doesn't have an attribute for a vector. However, `x` does have a length (although R doesn't consider it as an attribute).

```
length(x)
```

```
[1] 4
```

```
dim(x)
```

```
NULL
```

Creating Matrices

To use these two function (`rbind` and `cbind`), we first create some vectors.

```
x = 1:3; y = 4:6  ##we can have multiple commands in a single row
x
```

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

```
y
```

```
[1] 4 5 6
```

```
M1 = rbind(x,y) ##binding vectors into a matrix (row-wise)
M1 ##note that the matrix has row names ('x' and 'y')
```

```
  [,1] [,2] [,3]
x    1    2    3
y    4    5    6
```

```
class(M1)
```

```
[1] "matrix"
```

```
M2 = cbind(x,y) ##binding vectors into a matrix (column-wise)
M2  ##note that the matrix has column names ('x' and 'y')
```

```
     x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

```
class(M2)
```

```
[1] "matrix"
```

```
colnames(M2)
```

```
[1] "x" "y"
```

```
M2[x]
```

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

```
M2[y]
```

```
[1] 4 5 6
```

```
M2[x] + M2[y]
```

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

```
M2[x] * M2[y]
```

```
[1]  4 10 18
```

```
M2[x] %*% M2[y] ##matrix multiplication (in this case dot product)
```

```
     [,1]
[1,]   32
```

```
apply(M2, MARGIN=1, FUN=sum)
```

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

```
apply(M2, MARGIN=2, FUN=sum)
```

```
 x  y
 6 15
```

Important Note: R does **NOT** distinguish between row-vectors and column-vectors. That is why the columns of the matrix `M2` were printed in a row (horizontal line).

If you (mistakenly or not) try to make a vector with different class types, R will do a coercion; that is, will (forcefully) convert some of the entries into other types, according to its internal convention (and will not report an error). Examples:

```
x = c('first name', 2.3)
x
```

```
[1] "first name" "2.3"
```

```
class(x)
```

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

```
typeof(x)
```

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

```
y = c(7,FALSE)
y
```

```
[1] 7 0
```

```
c(class(y),typeof(y))
```

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

```
z = c('a',TRUE)
z
```

```
[1] "a"    "TRUE"
```

```
c(class(z),typeof(z))
```

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

Explicit Coercion

Objects can be intentionally coerced to others using `as.*` functions, if applicable:

```
x = 0:5
x
```

```
[1] 0 1 2 3 4 5
```

```
class(x)
```

```
[1] "integer"
```

```
xnum = as.numeric(x)
xnum
```

```
[1] 0 1 2 3 4 5
```

```r
class(xnum)
```

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

```r
xlog = as.logical(x)
xlog  ##note how the values are converted
```

```
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```r
class(xlog)
```

```
[1] "logical"
```

```r
xchar = as.character(x)
xchar
```

```
[1] "0" "1" "2" "3" "4" "5"
```

```r
class(xchar)
```

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

```r
y = as.numeric(c('a','b','c','d')) ##cannot convert characters into numbers
```

```
Warning: NAs introduced by coercion
```

```r
y
```

```
[1] NA NA NA NA
```

```r
class(y)
```

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

Lists

Vector is an ordered array-like object, whose all entries are of the same (atomic) type. Sometimes, you need an ordered array-like object, but whose components are not necessarily of the same type. In R, such an object type is called list. We can use command `list()` to create a list. For example

```r
xli = list(17, 17., c('bla',22,FALSE), c(7,'eleven'), c(12, TRUE, F), 2-3i)
xli  ## printing xli list
```

```
[[1]]
[1] 17

[[2]]
[1] 17

[[3]]
[1] "bla"   "22"    "FALSE"

[[4]]
[1] "7"      "eleven"

[[5]]
[1] 12  1  0

[[6]]
[1] 2-3i
```

Note that indexing elements of lists is done by double brackets

```
typeof(xli)
```

```
[1] "list"
```

```
attributes(xli)
```

```
NULL
```

```
c(xli[[5]],xli[[3]])  ##printing 5th and 3rd element of the list xli
```

```
[1] "12"    "1"     "0"     "bla"   "22"    "FALSE"
```

Note from the above that when the concatenating operator `c()` combines several types of data into a vector, it has to convert some data into other types, so that all entries of the obtained concatenated vector are of the same type.

NA, NaN and Inf

```
x = as.numeric(1/0)
x
```

```
[1] Inf
```

```
is.finite(x)
```

```
[1] FALSE
```

```
y = as.integer(1/0)
```

```
Warning: NAs introduced by coercion to integer range
```

```
y
```

```
[1] NA
```

```
is.finite(y)
```

```
[1] FALSE
```

```
is.infinite(y)
```

```
[1] FALSE
```

```
is.na(y)
```

```
[1] TRUE
```

`NA` = *Not Available* - denotes a missing value

```
z = 0/0
class(z)
```

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

```
z
```

```
[1] NaN
```

```
is.nan(z)
```

```
[1] TRUE
```

```
c(is.finite(z),is.infinite(z),is.na(z),is.nan(z))
```

```
[1] FALSE FALSE  TRUE  TRUE
```

```
x = c(1, 2., NaN, NA, Inf, 6)
x
```

```
[1]   1   2 NaN  NA Inf   6
```

```
typeof(x)
```

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

Function `is.na()` is very useful in R. It checks whether a value is missing. For previous vector `x`,

```
x
```

```
[1]   1   2 NaN  NA Inf   6
```

```
is.na(x)
```

```
[1] FALSE FALSE  TRUE  TRUE FALSE FALSE
```

We also have `is.nan()` function, checking whether a value is `NaN`:

```
is.nan(x)
```

```
[1] FALSE FALSE  TRUE FALSE FALSE FALSE
```

As the above example shows, `NaN` is a special type of `NA` Also, `NA` can be of numeric class, integer class, character class, or some other classes.

Let

```
y <- c(2, NA, 3.3, 4, NA, NA, 6)
```

Since `is.na(y)` is a logical vector, i.e.

```
is.na(y)
```

```
[1] FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE
```

to check how many `NA` values vector `y` contains, we can apply `sum()` function on the vector `is.na(y)`. Before applying `sum`, R will coerce the logical values `TRUE` and `FALSE` into 1 and 0, respectively.

```
sum(is.na(y))
```

```
[1] 3
```

We can also use `table()`, which gives counts for all different values that appear in a given vector:

```
table(is.na(y))
```

```
FALSE  TRUE
    4     3
```

Data Frames

Data frames can be thought of as well structured, two-dimensional lists (tables). Each element of this list can be thought of as a column. All columns are of the same length. Also, entries withing each column are of the same type/class, but different columns may be of different classes. Typically, data frames are created when importing tables from files (eg. csv, excel, dat, txt files). This will be discussed soon. We can also create data frame using `data.frame()` function, as illustrated below.

```r
x = runif(4) ##sampling 4 numbers from Uniform distn on [0,1]
x
```

```
[1] 0.8098837 0.9663120 0.9571504 0.5626746
```

```r
x = floor(1000*x)
x
```

```
[1] 809 966 957 562
```

```r
df = data.frame(ID=x, Name=c('Alice','Ben','Courtney','David')) ##creating data frame df
df
```

```
   ID     Name
1 809    Alice
2 966      Ben
3 957 Courtney
4 562    David
```

```r
nrow(df) ##number of rows of df
```

```
[1] 4
```

```r
ncol(df) ##number of columns of df
```

```
[1] 2
```

```r
colnames(df)
```

```
[1] "ID"   "Name"
```

Packages

As an open source, R has countless contributors. They are usually statisticians and/or programmers who solve certain complex (statistical) problem by writing a long, complex code. They are often kind enough to make the code quite universal and flexible, so that it can be used in various situations, and they package the code to make it available to other users. Packages are actually a collection of various functions, compiled code and often include sample data, as well. R packages are one of the main reasons why R is so popular.

When you installed R, a limited basic packages were installed by default. You can then add any package you come accross and want to use. Those packages are stored in R servers (mirrors), and installation of a new package requires internet connection to the CRAN mirror you chose to install R from.

To install a package, you use the command

```r
install.packages("<packagename>")
```

where `<packagename>` should be replaced by the name of the package you want to install. For example, `swirl` is a package for learning R (within R). We discuss `swirl` package later in this file. To install `swirl`, we type

```r
install.packages("swirl")   ## don't forget quotation marks
```

in the console.

Any time you open R (in a new session), if you want to use certain package, you need to load it. This is done by the command `library()`. For example, once we installed `swirl`, in order to use it, we type

```r
library(swirl)   ## note that qoutation marks are not required! (but could be used)
                 ## talk about inconsistency in R syntax!!
```

Remark: Instead of loading a package using `library(<packagename>)`, you can see on internet codes that use command `require(<packagename>)` instead. While it could be used, I do NOT recommend[2].

Reading Data Into R

Most commonly used functions for reading data into `R`:

- `read.table, read.csv` - for reading tabular data (eg. data frames)
- `readLines` - for reading lines of a text file
- `source` - for reading in `R` code files (inverse of `dump`)
- `dget` - for reading in `R` code files (inverse of `dput`)
- `load` - for reading in saved workspaces
- `unserialize` - for reading single `R` objects in binary form

Writing Data Into Files

Most commonly used functions for writing data into `R` (which are analogs of the above reading commands):

- `write.table`
- `writeLines`
- `dump`
- `dput`
- `save`
- `serialize`

Reading Data Files with read.table()

The command `read.table()` is one of the most commonly used commands for reading tabular data (apart from recently developed `readr()`). The `read.table()` function has several important arguments:

- `file` - the name of a file, or a connection
- `header` - logical indicating if the file has a header line
- `sep` - a string indicating how the columns are separated
- `colClasses` - a character vector indicating the class of each column in the dataset
- `nrows` - the number of rows in the dataset. By default `read.table()` reads an entire file.
- `comment.char` - a character string indicating the comment character. The default is "#". If there are no commented lines in your file, it's worth setting this to be the empty string `""`.
- `skip` - the number of lines to skip from the beginning (often, some lines at the beginning are description of data)
- `stringsAsFactors` - should character variables be coded as factors? This defaults to `TRUE` because back in the old days, if you had data that were stored as strings, it was because those strings represented levels of a categorical variable. Now we have lots of data that is text data and they don't always represent categorical variables. So you may want to set this to be `FALSE` in those cases. If you always want this to be `FALSE`, you can set a global option via `options(stringsAsFactors = FALSE)`.

Round-off Error

Comparing the result of adding numbers 1 and 2 with the value of 3, we get

```
1 + 2 == 3
```

[2]Once you become familiar with basic R, see this blog post for the reason of using `library()` over `require()`: https://yihui.org/en/2014/07/library-vs-require/

```
[1] TRUE
```

As expected, this is `TRUE`. However,

```
0.1 + 0.2 == 0.3
```

```
[1] FALSE
```

What!!?? What just happened!!?? How is this possible!!??

To understand why the computer gave us an obviously wrong answer, we need to dive in for a bit into very basics of computer architecture and how numbers are stored.

There are infinitely many real numbers, and only finite memory. This means there are only finitely many numbers that computer can store and write down exactly as they really are (and all of them are rational numbers). Moreover, the numbers are stored using binary system (rather than human preferred decimal system). Conversion from decimal to binary (as well as other way around) can result in yet another round-off error. For example, neither 0.1 nor 0.2 (written this way in decimal notation) can be written in binary system using only finitely many digits of 0's and 1's. To see this, note that 0.1 is written this way in decimal system because

$$0.1 = \frac{1}{10} = \frac{0}{10^0} + \frac{1}{10^1} + \frac{0}{10^2} + \frac{0}{10^3} + \dots$$

It is not difficult to see that rewriting this number as a linear combination of 2^{-i}'s, we have

$$0.1 = \frac{0}{2^0} + \frac{0}{2^1} + \frac{0}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \frac{0}{2^{11}} + \frac{1}{2^{12}} + \frac{1}{2^{14}} + \dots$$

This means the binary notation of this number is $0.0001100110011001100110011\ldots$. The chunk of digits 0011 repeats indefinitely. Therefore, we cannot exactly store it in 64 bits (with each bit being 0 or 1). Similarly,

$$0.2 = \frac{0}{2^0} + \frac{0}{2^1} + \frac{0}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{0}{2^5} + \frac{0}{2^6} + \frac{1}{2^7} + \frac{1}{2^8} + \frac{0}{2^9} + \frac{0}{2^{10}} + \frac{1}{2^{11}} + \frac{1}{2^{12}} + \dots$$

and thus, 0.2 written in binary system looks like $0.0011001100110011\ldots$, with indefinite repetition of 0011. Therefore, the round-off errors are being made when storing numbers written in decimal system as 0.1 and 0.2. Consequently, the result of adding them up is made with an error. Furthermore, it can also be shown that binary representation of 0.3 is $0.011001100110011\ldots$, with indefinite repetition of 0011. Therefore, 0.3 cannot be stored exactly in 64 bits neither. Unfortunately, the error of storing 0.3 does not match the error of storing the result of the operation 0.1 + 0.2, and thus, the representations of 0.1 + 0.2 and 0.3 in binary system using 64 bits are not the same, causing the result of checking the equality of 0.1 + 0.2 and 0.3 to be `FALSE`.

We can learn an important lesson from this example: if a double precision real variable `x` is a result of a computation, and if we are hoping it to be equal to certain number, say, `a`, we should avoid using the command `x == a` for comparing `x` with `a`. Instead, it is better to check whether the difference between `x` and `a`, i.e. `|x-a|`, is very small (practically zero). For example, the test of equality can be whether `|x - a|` $< 10^{-10}$, which in R is written as

```
abs(x-a) < 1e-10
```

There is another thing we should also be aware of. Let us define variable `x` to be the result of the addigion `0.1 + 0.3`. We then compare this variable with `0.3` using `==`.

```
x = 0.1 + 0.2
x == 0.3  ## logical statement which claims that x is equal to 0.3
```

```
[1] FALSE
```

This is as expected, based on the previous discussion. In other words, variable `x`, i.e. the result of adding `0.1+ 0.2` is **NOT** stored as `0.3`. However, printing out variable `x` to see what it actually is (in decimal system), we get:

```
x
```

```
[1] 0.3
```

At first glance it seems that `x` **IS** equal to `0.3`. Nevertheless, a closer look at its 17 (decimal) digits by using the command `format()`, we can see why `x == 0.3` gave us `FALSE` as an output.

```
format(x, digits=17)
```

```
[1] "0.30000000000000004"
```

What happened is that by default R prints out real numbers with 6 decimal places, unless formated by user in a different way. So, R wanted to print out `0.300000`, which is then simplified to the notation `0.3`.

This issue with round-off errors when storing variables, happens only with real variables. Integer variables, however, are stored exactly. As already noted:

```
1 + 2 == 3
```

```
[1] TRUE
```

Remark: We had that the round-off errors of storing `0.1` and `0.2` prior to computing `0.1 + 0.2` and the round off error of storing `0.3` resulted in `FALSE` outcome of the comparison `0.1 + 0.2 == 0.3`. However, sometimes two wrongs are equally wrong:

```
0.1 + 0.1 == 0.2
```

```
[1] TRUE
```

Plots

We will discuss plots soon, but if you are anxious to see them, you can check out these websites to see the capability of `R` graphics and menagerie of its plots

- https://www.r-graph-gallery.com
- http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html
- https://exts.ggplot2.tidyverse.org/gallery/
- https://www.ling.upenn.edu/~joseff/rstudy/week4.html

SWIRL

Swirl is an interactive way to learn R (within R). It was developed by Nick Carchedi, a student from Johns Hopkins University (two other guys joined him later). Swirl stands for Statistics With Interactive R Learning. To use Swirl, do the following:

1. Install `swirl`

Since `swirl` is an `R` package, you can easily install it by entering a single command from the R console:

```
install.packages("swirl")
```

2. Load `swirl`

Every time you want to use `swirl`, you need to first load the package (this is true for any R package). From the R console, type:

```
library(swirl)
```

(**Side note:** There is a command `require()` often used instead of `library()`, and it often appears in forums and literature. However, do **NOT** use it just for loading a package (for clue why not, type `require` in the console (without parentheses) to see what the source code looks like; it tries to use `library()`!!?? Here is a blog page for more detail, but don't read it now, as you first need to learn basics of R coding)

3. Install the R Programming course

Swirl offers a variety of interactive courses, but for our purposes, you want the one called "R Programming". Type the following from the R prompt to install this course:

```
install_from_swirl("R Programming")
```

4. Start `swirl`

```
swirl()
```

Then, follow the menus and select the R Programming Course when given the option. For the first part of this course you should complete the following lessons:

1: Basic Building Blocks

2: Workspace and Files

3: Sequences of Numbers

4: Vectors

5: Missing Values

6: Subsetting Vectors

7: Matrices and Data Frames

8: Logic

9: Functions

10: lapply and sapply

11: vapply and tapply

12: Looking at Data

13: Simulation

14: Dates and Times

15: Base Graphics

Appendix: More on Assignment Operators

The Tidyverse Style Guide (https://style.tidyverse.org/), and a current version of Google's R Style Guide (https://google.github.io/styleguide/Rguide.html) which was derived from the Tidyverse Style[3], suggest the use of `<-` rather than `=`.

However, any style guide is opinionated. To support my choice of `=`, I believe it is no problem for this syntax to be widely understood, especially by programmers who come to R from some other languages (C++, Java, Matlab, Python, JavaScript) which use `=`. Moreover, anyone who has extensive experience in these languages prior to starting with R would probably find more natural to continue with `=`, than to switch to `<-`.

RStudio has a key shortcut for `<-`, which is `Alt + '-'`. They would have convinced me to use `<-`, had they chosen `Alt + '='` instead.

Although the R community seemingly coalesced to `<-`, it is still quite common to see `=`.

[3]The first Google's R Style Guide was not related to Tidyverse style, but the current version was derived from Tidyverse Style

Fun fact: One of the R Core Team members is prof. Duncan Temple Lang. In the book "*Data Science in R; A case Studies Approach to Computational Reasoning and Problem Solving*", which he wrote with prof. Deborah Nolan, they use `=`. Furthermore, Hadley Wickham, a great R contributor and an advocate of Tidyverse Style (which also means the opponent of using `=`), is also one of the contributors to the book. He wrote chapter 9 (1st edition) and the code in this chapter uses the assignment operator `=`. It is interesting to see Hadley's code using `=`. I thought it was due to consistency in the book. However, in chapter 5 of this book, written by the contributor Michael Kane, the operator `<-` is used.

There are three more assignment operators: `->`, `<<-` and `->>`. Using analogy with `<-`, one can write

```
3.14 -> x
```

which has the same effect as `x <- 3.14` or `x = 3.14`.

The operator `->` is archaic (but still exists and works!), and is a reminder of the olden days when there was no graphical user interface nor mouse. Just after writing a long expression, if the programmer realizes he/she forgot to assign the expression to a variable, it is not a problem today just to use a mouse, click at the beginning of the expression and assign it to a variable. However, in the old days, you would need to walk the cursor by keeping the arrow key pressed all the way from the end to the beginning of the expression.

Here is a weird, dirty-style code that assigns value 3.14 to both variables `x` and `y`:

```
x <- 3.14 -> y
x   ## printing x
```

```
[1] 3.14
```

```
y   ## printing y
```

```
[1] 3.14
```

This is equivalent with

```
x = 3.14; y = x
```

By the way, both commands `x <- 3.14 -> y` and `x = 3.14; y = x` actually assign two names, `x` and `y`, to the same memory address, i.e. to the same object. To see this, we can use `obj_addr()` from library `lobstr`, which you first need to install by typing the command `install.packages("lobstr")` in the console.

```
library(lobstr)

x = 3.14
y = x

obj_addr(x)
```

```
[1] "0x195c6c80"
```

```
obj_addr(y)
```

```
[1] "0x195c6c80"
```

You will get a different memory address than what you see here, and moreover, each time you run the code a new address is assigned. But you will always see that both names `x` and `y` refer to the same object. (**Note:** After modifying `x` or `y`, these two names will then refer to diffent addresses, i.e. different objects; this is so called copy-on-modify property. We will see later that Python behaves differently in such situation.)

But wait, there is more! Yet, there are two more assignment operators: `<<-` and `->>`. They are at quite advanced level of R programming and most likely you will not need them for a while. So, we will not discuss them here, other then saying the following. The operators `<<-` and `->>` are typically used within functions, for accessing variables at different, higher levels/scopes. For example, sometimes there is a need for creating a function that can create functions (the former is called parent function, the latter are child functions). While

working in the scope of a child function you might need to access and manipulate variable in the scope of the parent function. This cannot be done by `<-`, but by `<<-` (or `->>`, which is the rightward version of `<<-`). Anyways, in this course you only need `<-` or `=`.