한양대학교
HANYANG UNIVERSITY

Introduction to the R language (1)

Seoncheol Park

2.1 First steps

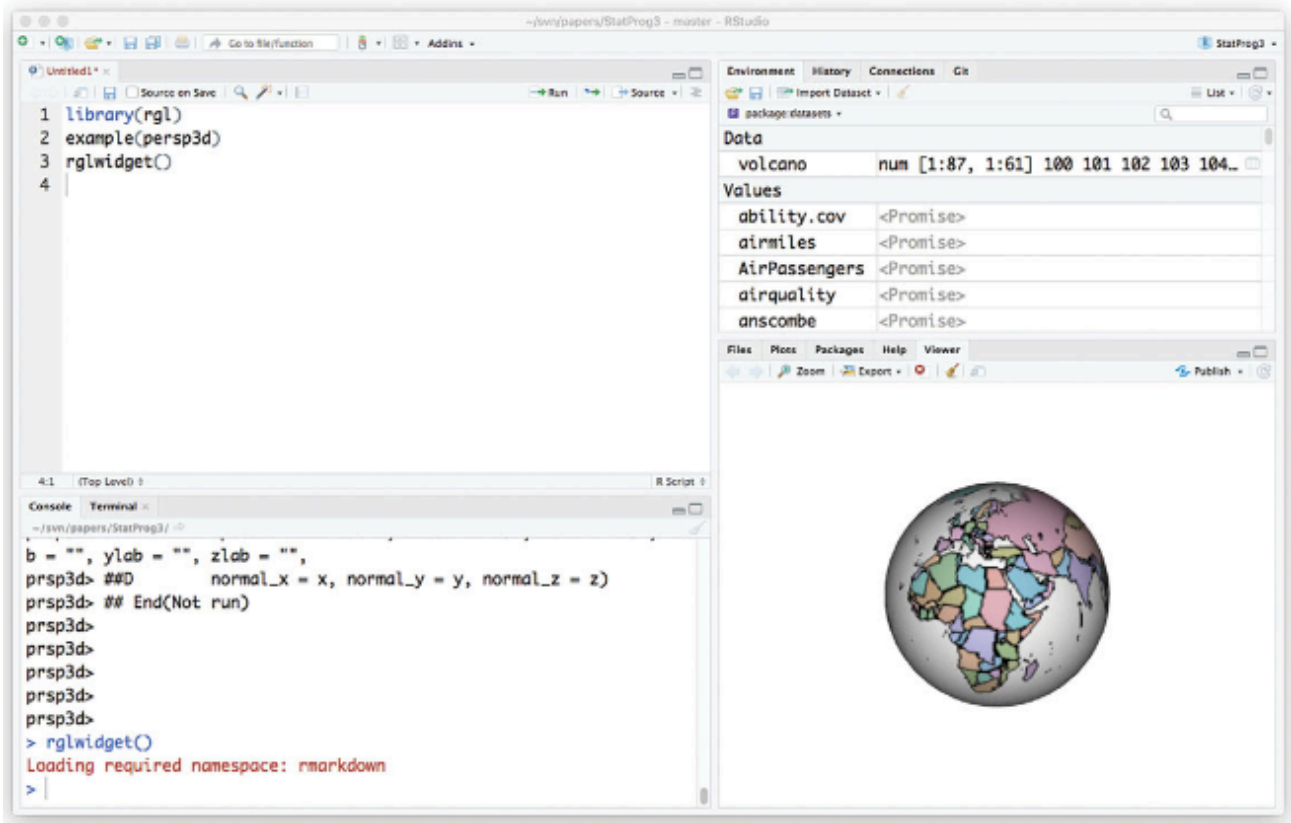

Figure 1: A typical RStudio display

- Having opened R or RStudio, you may begin entering and executing commands.
  - Normally, you will use the **Source Pane** to type in your commands,
  - but you may occasionally use the **Console Pane** directly. The greater-than sign (>) is the prompt symbol which appears in the Console Pane.

2.1.1 R as a calculator

- Anything that can be computed on the calculator app on your smartphone can be computed at the R prompt.

Basic operations

- The basic operations are
  - `+` (add),
  - `-` (subtract),
  - `*` (multiply), and
  - `/` (divide).
- For example, try

```
5504982/131071
```

- Upon pressing the Enter key (or `CTRL-Enter`, or `CMD-Enter`, depending on your system), the result of the above division operation, `42`, appears in the Console Pane, preceded by the command you executed, and prefixed by the number 1 in square brackets:

```
5504982/131071
```

```
[1] 42
```

- The `[1]` indicates that this is the first (and in this case only) result from the command.

Multiple commands

- Many commands return multiple values, and each line of results will be labeled to aid the user in deciphering the output.

```
#the sequence of integers from 17 to 58
17:58
```

```
 [1] 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
[26] 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
```

The first line starts with the first return value, so is labeled `[1]`; the second line starts with the 23rd, so is labeled `[23]`.

Parentheses (`(`, `)`)

- Everything that you type after a `#` sign is assumed to be a comment and is ignored by R.

```
5:(2*3 + 10) # the result is the same as 5:16
```

```
 [1]  5  6  7  8  9 10 11 12 13 14 15 16
```

```
(7:10) + pi # pi is a stored constant
```

```
[1] 10.14159 11.14159 12.14159 13.14159
```

- Note that parentheses (`(`, `)`) are used to ensure that the operations (in this caes, `:`, `*`, and `+`) are carried out in the order that we desire.
- In the first case, parentheses were necessary to obtain the result we wanted to see. The following shows what happens when the parentheses are omitted:

```
5:2*3 + 10
```

```
[1] 25 22 19 16
```

```
#comparison:
5:(2*3) + 10
```

```
[1] 15 16
```

- The parentheses were not required in `(7:10) + pi`. We used them anyway, for two reasons.
  - (1) They can help others read and understand the code more quickly.
  - (2) Although R follows strict and consistent rules regarding order of operations, we believe it is too easy for a user to forget one or more of these rules. Therefore, we recommend using parentheses whenever you are unsure.

Other operators

- R can also be used to compute powers with the `^` operator. For example,

```
3^4
```

```
[1] 81
```

- Modular arithmetic is also available. For example, you can compute the remainder after division of 31 by 7, i.e. 31 (mod 7):

```
31 %% 7
```

```
[1] 3
```

and the integer part of a fraction as

```
31 %/% 7
```

```
[1] 4
```

- We can confirm that 31 is the sum of its remainder plus seven times the integer part of the fraction:

```
7*4 + 3
```

```
[1] 31
```

2.1.2 Named storage

- R has a workspace known as the **global environment** that can be used to store the results of calculations, and many other types of objects.
- Suppose we would like to store the result of the calculation `1.0025^30` for future use. We will assign this value to an object called `interest.30`. To do this, we type

```
interest.30 <- 1.0025^30
```

- We tell R to make the **assignment** using an arrow that points to the left, created with the less-than sign (`<`) and the hyphen (`-`).
- R also supports using the equals sign (`=`) in place of the arrow in most circumstances. But the authors recommend using the arrow, as it makes clear that we are requesting an action (i.e. an assignment), rather than stating a relation (i.e. that `interest.30` is equal to `1.0025^30`), or making a permanent definition.
- Note that when you run this statement, **no output appears**: R has done what we asked, and is waiting for us to ask for something else.

- You can see the results of this assignment by **just** typing the name of our new object at the prompt:

```
interest.30
```

```
[1] 1.077783
```

- We can also use `interest.30` in further calculations if you wish. For example, you can calculate the bank balance after 30 years at 0.25% annual interest, if you start with an initial balance of $3000:

```
initialBalance <- 3000
finalBalance <- initialBalance * interest.30
finalBalance
```

```
[1] 3233.35
```

Example 2.1

An individual wishes to take out a loan, today, of P at a monthly interest rate i. The loan is to be paid back in n monthly installments of size R, beginning one month from now.

Goal: calculate R.

Equating the present value P to the future (discounted) value of the n monthly payments R, we have

$$P = R(1+i)^{-1} + R(1+i)^{-2} + \cdots + R(1+i)^{-n}$$

or

$$P = R\sum_{j=1}^{n}(1+i)^{-j}.$$

Summing this geometric series and simplifying, we obtain

$$P = R\Big(\frac{1-(1+i)^{-n}}{i}\Big).$$

This is the formula for the *present value of an annuity*. We can find R, given P, n and i as

$$R = P\frac{i}{1-(1+i)^{-n}}.$$

- In R, we define variables as follows:
  - `principal`: P
  - `intRate`: i, and
  - `n`: n
  - We will assign the resulting payment value to an object called `payment`.

Suppose that the loan amount is $1500, the interest rate is 1% and the number of payments is 10. The resulting code is

```
intRate <- 0.01
n <- 10
principal <- 1500
payment <- principal * intRate / (1 - (1 + intRate)^(-n))
payment
```

```
[1] 158.3731
```

2.1.3 Quitting R

- To quit your R session, run

```
q()
```

or choose `Quit Seccion...` from the `File` menu.

- You will then be asked whether to save an image of the current workspace, or not, or to cancel.
- **Note**: What happens if your omit the parentheses `()` when attempting to quit:

```
q
```

```
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<bytecode: 0x10c1c62e8>
<environment: namespace:base>
```

- This has happened because `q` is a **function** that is used to tell R to quit.
  - Typing `q` by itself tells R to show us the contents of the function `q`.
  - By typing `q()`, we are telling R to call the function `q`. The action of this function is to quit R.

2.2.1 Functions in R

- Most of the work in R is done through **functions**.

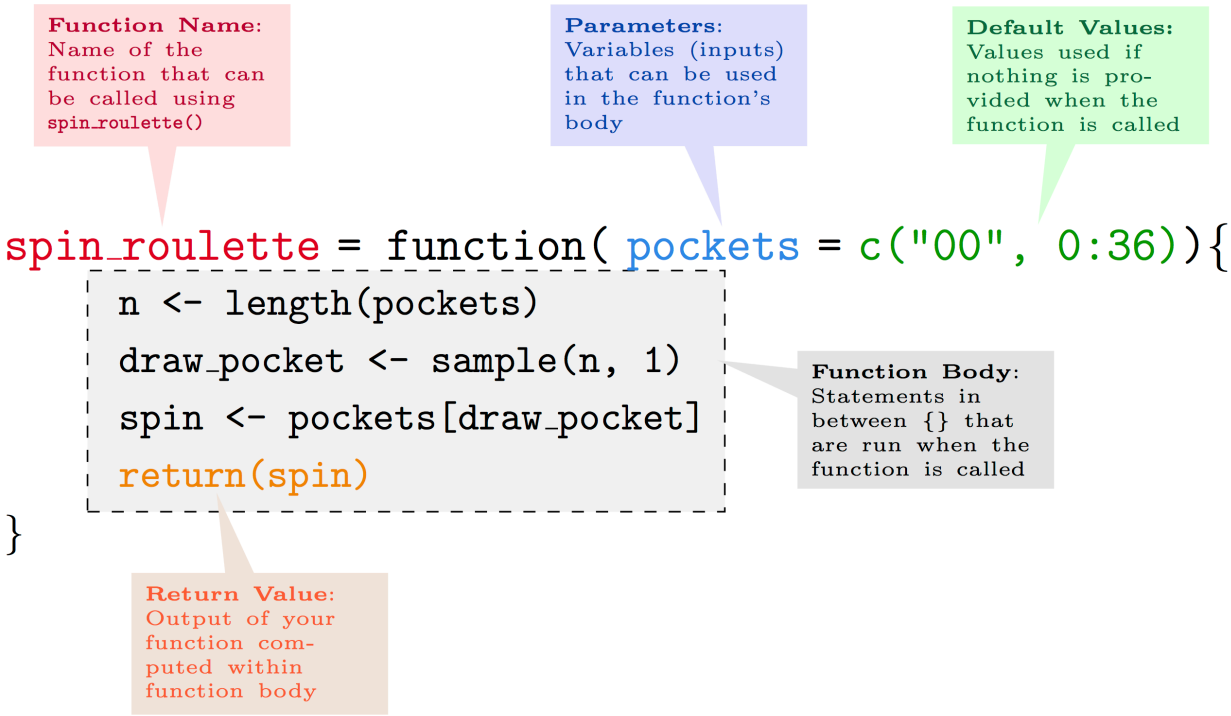

Figure 2: A typical function structure in R from https://smac-group.github.io/ds/functions.html.

- For example, we saw that to quit R we can type `q()`. This tells R to call the function named `q`.

- The parentheses surround the **argument list**, which in this case contains nothing: we just want R to quit, and do not need to tell it how.
- We also saw that `q` is defined as

```
q
```

```
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<bytecode: 0x10c1c62e8>
<environment: namespace:base>
```

- This shows that `q` is a function that has three arguments:
  - `save`,
  - `status`, and
  - `runLast`.
- Each of those has a default value:
  - `"default"`,
  - `0`, and
  - `TRUE`, respectively.
- What happens when we execute `q()` is that R calls the q function with the arguments set to their default values.

- If we want to change the default values, we specify them when we call the function. Arguments are identified in the call (1) by their position, or by (2) specifying the name explicitly. Both

```
q("no")
q(save = "no")
```

tell R to call `q` with the first argument set to `"no"`, i.e. to quit without saving the workspace. If we had given two arguments without names, they would apply to `save` and `status`.

- If we want to accept the defaults of the early parameters but change later ones, we give the name when calling the function, e.g.

```
q(runLast = FALSE)
```

or use commas to mark the missing arguments, e.g.

```
q( , , FALSE)
```

- Note that we must use `=` to set arguments.
  - If we had written `q(runLast <- FALSE)` it would be interpreted quite differently from `q(runLast = FALSE)`.
  - The arrow says to put the value `FALSE` into a variable named `runLast`.
  - We then pass the result of that action (which is the value `FALSE`) as the first argument of `q()`.
  - Since save is the first argument, it will act like `q(save = FALSE)`, which is probably not what we wanted.

2.2.2 R is case-sensitive

- Consider this:

```
x <- 1:10
MEAN(x)
```

```
Error in MEAN(x): could not find function "MEAN"
```

- Now try

```
MEAN <- mean
MEAN(x)
```

```
[1] 5.5
```

- The function `mean()` is built in to R. R considers `MEAN` to be a different function, because it is case-sensitive: `m` is different from `M`.

2.2.3 Listing the objects in the workspace

- A list of all objects in the current workspace can be printed to the screen using the `objects()` function:

```
objects()
```

```
[1] "finalBalance"    "has_annotations" "initialBalance"  "interest.30"
[5] "intRate"         "MEAN"            "n"               "payment"
[9] "principal"       "x"
```

- A synonym for `objects()` is `ls()`. In RStudio the Environment Pane shows both the names and abbreviated displays of the objects' values.
- If we quit our R session without saving the workspace image, then these objects will disappear.
- If we save the workspace image, then the workspace will be restored at our next R session.

2.3.1 Numeric vectors

- A numeric vector is a list of numbers. The `c()` function is used to collect things together into a vector.

```
c(0, 7, 8)
```

```
[1] 0 7 8
```

- Again, we can assign this to a named object:

```
x <- c(0, 7, 8) # now x is a 3-element vector
```

- To see the contents of `x`, simply type

```
x
```

```
[1] 0 7 8
```

- The `:` symbol can be used to create **sequences** of increasing (or decreasing) values.

```
numbers5to20 <- 5:20
numbers5to20
```

```
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

- Vectors can be joined together (i.e. *concatenated*) with the `c` function.

```
c(numbers5to20, x)
```

```
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20  0  7  8
```

- Here is another example of the use of the `c()` function.

```
some.numbers <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
   43, 47, 59, 67, 71, 73, 79, 83, 89, 97, 103, 107, 109, 113, 119)
some.numbers
```

```
 [1]   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  59  67  71  73
[20]  79  83  89  97 103 107 109 113 119
```

- **Note**: If you type this in the R console (not in the RStudio Source Pane), R will prompt you with a `+` sign for the second line of input. That means the code is **incomplete**:

```
#try this
c(numbers5to20, x
```

- We can append `numbers5to20` to the end of `some.numbers`, and then append the decreasing sequence from 4 to 1:

```
a.mess <- c(some.numbers, numbers5to20, 4:1)
a.mess
```

```
 [1]   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  59  67  71  73
[20]  79  83  89  97 103 107 109 113 119   5   6   7   8   9  10  11  12  13  14
[39]  15  16  17  18  19  20   4   3   2   1
```

2.3.2 Extracting elements from vectors

Q. How can we extract the 22nd element of `a.mess`?

- A way to display the 22nd element of `a.mess` is to use square brackets to extract just that element:

```
a.mess[22]
```

```
[1] 89
```

- We can extract more than one element at a time.

```
#3, 6, 7 elts of `a.mess`
a.mess[c(3, 6, 7)]
```

```
[1]  5 13 17
```

- To get the third through seventh element of `numbers5to20`, type

```
numbers5to20[3:7]
```

```
[1]  7  8  9 10 11
```

- Negative indices can be used to avoid certain elements.

```
#select all but the second and tenth elts of `numbers5to20`
numbers5to20[-c(2,10)]
```

```
 [1]  5  7  8  9 10 11 12 13 15 16 17 18 19 20
```

- The third through eleventh elements of `numbers5to20` can be avoided as follows:

```
numbers5to20[-(3:11)]
```

```
[1]  5  6 16 17 18 19 20
```

- Using a zero index returns nothing. This is not something that one would usually type, but it may be useful in more complicated expressions. For example, recall that `x` contains the vector `(0,7,8)` so that

```
numbers5to20[x]
```

```
[1] 11 12
```

- **Note 1**: Do **not** mix positive and negative indices. To see what happens, observe

```
x[c(-2, 3)]
```

```
Error in x[c(-2, 3)]: only 0's may be mixed with negative subscripts
```

- The problem is that it is not clear what is to be extracted: do we want the third element of x before or after removing the second one?
- **Note 2**: Always be careful to make sure that vector indices are integers. When fractional values are used, they will be truncated towards 0. Thus 0.6 becomes 0, as in

```
x[0.6]
```

```
numeric(0)
```

- The output `numeric(0)` indicates a numeric vector of length zero.

2.3.3 Vector arithmetic

- Arithmetic can be done on R vectors. For example, we can multiply all elements of `x` by 3:

```
x*3
```

```
[1]  0 21 24
```

- Note that the computation is performed **elementwise**. Addition, subtraction, and division by a constant have the same kind of effect.

```
y <- x - 5
y
```

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

- For another example, consider taking the 3rd power of the elements of `x`:

```
x^3
```

```
[1]   0 343 512
```

- In general, the binary operators also work element-by-element when applied to pairs of vectors. For example, we can compute $y_i^{x_i}$, for $i = 1, 2, 3$, i.e. $(y_1^{x_1}, y_2^{x_2}, y_3^{x_3})$, as follows:

```
y^x
```

```
[1]   1  128 6561
```

- When the vectors are different lengths, the shorter one is extended by **recycling**: values are repeated, starting at the beginning. For example, to see the pattern of remainders of the numbers 1 to 10 modulo 2 and 3,

```
c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10) %% 2:3
```

```
 [1] 1 1 0 2 1 0 0 1 1 2 0 0 1 1 0 2 1 0 0 1
```

- R will give a warning if the length of the longer vector is not a multiple of the length of the smaller one, because that is often a symptom of an error.

```
c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10) %% 2:4
```

```
 [1] 1 1 2 0 0 3 0 1 1 1 0 2 1 1 0 0 0 1 0 1
```

Q. Do you see the error? -> **No**.

2.3.4 Simple patterned vectors

- Patterned vectors can also be produced using the `seq()` function as well as the `rep()` function. For example, the sequence of odd numbers less than or equal to 21 can be obtained using

```
seq(1, 21, by = 2)
```

```
 [1]  1  3  5  7  9 11 13 15 17 19 21
```

- Notice the use of `by = 2` here. The `seq()` function has several **optional parameters**, including one named `by`. If `by` is not specified, the default value of 1 will be used.

- Repeated patterns are obtained using `rep()`. Consider the following examples:

```
rep(3, 12) # repeat the value 3, 12 times
```

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

```
rep(seq(2, 20, by = 2), 2) # repeat the pattern 2 4 ... 20, twice
```

```
 [1]  2  4  6  8 10 12 14 16 18 20  2  4  6  8 10 12 14 16 18 20
```

```
rep(c(1, 4), c(3, 2)) # repeat 1, 3 times and 4, twice
```

```
[1] 1 1 1 4 4
```

```
rep(c(1, 4), each = 3)  # repeat each value 3 times
```

```
[1] 1 1 1 4 4 4
```

```
rep(1:10, rep(2, 10)) # repeat each value twice
```

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

2.3.5 Vectors with random patterns

- The `sample()` function allows us to simulate things like the results of the repeated tossing of a 6-sided die.

```
sample(1:6, size = 8, replace = TRUE) # an imaginary die is tossed 8 times
```

```
[1] 1 1 1 1 1 3 1 6
```

2.3.6 Character vectors

- Scalars and vectors can be made up of strings of characters instead of numbers. All elements of a vector must be of the same type. For example,

```
colors <- c("red", "yellow", "blue")
more.colors <- c(colors, "green", "magenta", "cyan") # this appended some new elements
z <- c("red", "green", 1) # an attempt to mix data types in a vector
```

- To see the contents of `more.colors` and `z`, simply type

```
more.colors
```

```
[1] "red"     "yellow"  "blue"    "green"   "magenta" "cyan"
```

```
z # 1 has been converted to the character "1"
```

```
[1] "red"   "green" "1"
```

- There are two basic operations you might want to perform on character vectors. To take substrings, use `substr()`. It takes arguments `substr(x, start, stop)`, where
  - `x` is a vector of character strings, and
  - `start` and `stop` say which characters to keep.

- For example, to print the first two letters of each color use

```
substr(colors, 1, 2)
```

```
[1] "re" "ye" "bl"
```

- The other basic operation is building up strings by concatenation within elements. Use the `paste()` function for this. For example,

```
paste(colors, "flowers")
```

```
[1] "red flowers"    "yellow flowers" "blue flowers"
```

- There are two optional parameters to `paste()`.
  - The `sep` parameter controls what goes between the components being pasted together.
  - The `paste0()` function is a shorthand way to set `sep = ""`:
  - The collapse parameter to `paste()` allows all the components of the resulting vector to be collapsed into a single string:

```
paste("several ", colors, "s", sep = "")
```

```
[1] "several reds"    "several yellows" "several blues"
```

```
paste0("several ", colors, "s")
```

```
[1] "several reds"    "several yellows" "several blues"
```

```
paste("I like", colors, collapse = ", ")
```

```
[1] "I like red, I like yellow, I like blue"
```

2.3.7 Factors

- Factors offer an alternative way to store character data. A factor with four elements and having the two levels, `control` and `treatment` can be created:

```
grp <- c("control", "treatment", "control", "treatment")
grp
```

```
[1] "control"   "treatment" "control"   "treatment"
```

```
grp <- factor(grp)
grp
```

```
[1] control   treatment control   treatment
Levels: control treatment
```

- Factors can be an efficient way of storing character data when there are repeats among the vector elements. This is because the levels of a factor are internally coded as integers. To see what the codes are for our factor, we can type

```
as.integer(grp)
```

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

- The labels for the levels are only stored once each, rather than being repeated. The codes are indices of the vector of levels:

```
levels(grp)
```

```
[1] "control"   "treatment"
```

```
#levels(grp)[as.integer(grp)]
```

- The `levels()` function can be used to change factor labels as well. For example, suppose we wish to change the `"control"` label to `"placebo"`. Since `"control"` is the first level, we change the first element of the `levels(grp)` vector:

```
levels(grp)[1] <- "placebo"
```

- An important use for factors is to list all possible values, even if some are not present. For example,

```
sex <- factor(c("F", "F"), levels = c("F", "M"))
sex
```

```
[1] F F
Levels: F M
```

shows that there are two possible values for `sex`, but only one is present.

2.3.8 More on extracting elements from vectors

- As for numeric vectors, square brackets `[]` are used to index factor and character vector elements. For example, the factor `grp` has four elements, so we can print out the third element by typing

```
grp[3]
```

```
[1] placebo
Levels: placebo treatment
```

- We can access the second through fifth elements of `more.colors` as follows:

```
more.colors[2:5]
```

```
[1] "yellow"  "blue"    "green"   "magenta"
```

2.3.9 Matrices and arrays

- To arrange values into a matrix, we use the `matrix()` function:

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m
```

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

- We can then access elements using two indices. For example, the value in the first row, second column is

```
m[1, 2]
```

```
[1] 3
```

- Somewhat confusingly, R also allows a matrix to be indexed as a vector, using just one value:

```
m[4]
```

```
[1] 4
```

- Here elements are selected in the order in which they are stored internally: down the first column, then down the second, and so on. This is known as **column-major storage order**.
- Some computer languages use **row-major storage order**, where values are stored in order from left to right across the first row, then left to right across the second, and so on.
- Whole rows or columns of matrices may be selected by leaving one index blank:

```
m[1,]
```

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

```
m[,1]
```

```
[1] 1 2
```

- A more general way to store data is in an **array**. Arrays have multiple indices, and are created using the array function:

```
a <- array(1:24, c(3, 4, 2))
a
```

```
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

```
dim(a) #dimensions of a
```

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

Floating point numbers

- When R creates an integer, R uses the native **32-bit integer** format, which uses (1) 31 bits to store the number and (2) one bit to store the sign of the number.
- Within these limitations, the largest positive number R can hold as an integer is $2^31 - 1 = 2147483647$.

```
as.integer(2^31 - 1)
```

```
[1] 2147483647
```

```
as.integer(2^31) ##too large number for the integer type
```

```
[1] NA
```

- Of course, R has no trouble with numbers larger than $2^{31} - 1$. However, it will not use the internal integer storage format to hold these numbers.

```
2^32
```

```
[1] 4294967296
```

```
class(2^32)
```

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

- **Floating point** numbers provide the way to overcome the limitations of binary integers.

```
2^40
```

```
[1] 1.099512e+12
```

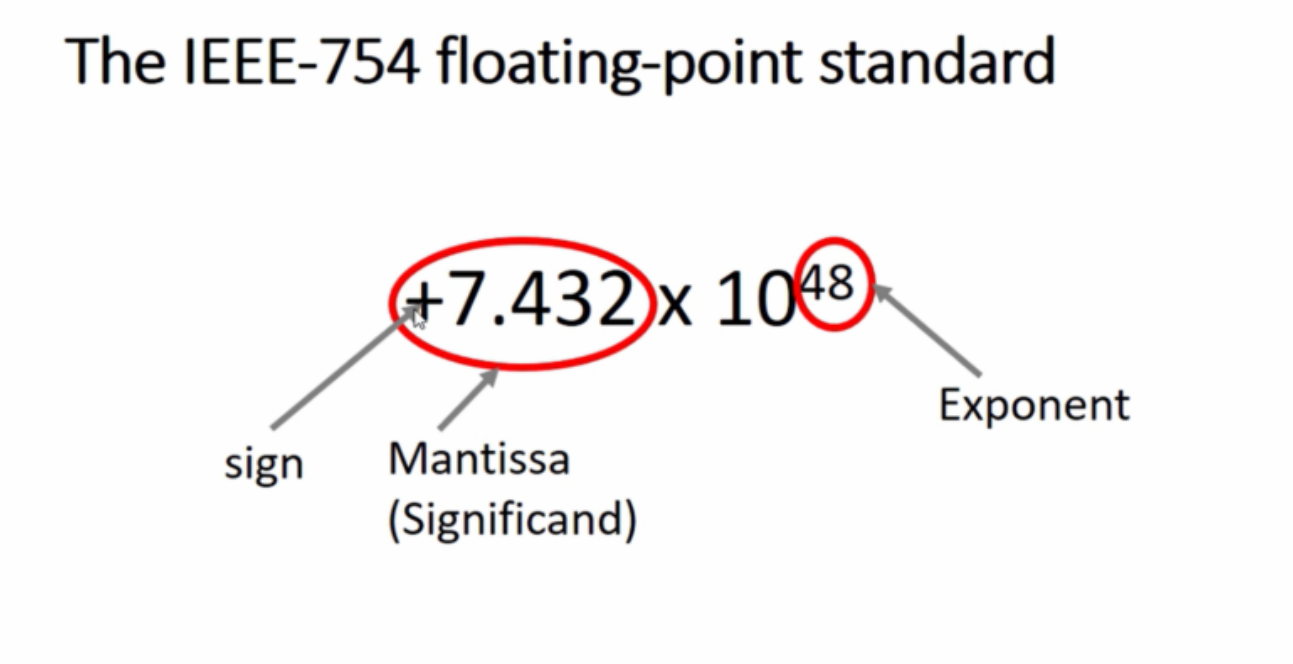

Figure 3: The IEEE-754 floating-point standard from https://fastbitlab.com.

- For more information about the floating point numbers system, please refer this webpage.

- As we have seen, floating point can represent a very broad range of numbers. However, it cannot also represent every number.
- Machine ϵ is defined as the smallest value such that,

$$1 + \epsilon > 1,$$

  within the floating point system.
- Intuitively, we know that 1 plus any number greater than 0 will be greater than 1. But with the number spacing within floating point, that is not necessarily the case. Machine ϵ is the threshold value for representation.

```
.Machine$double.eps
```

```
[1] 2.220446e-16
```

```
1 + .Machine$double.eps > 1
```

```
[1] TRUE
```

```
1 + (.Machine$double.eps/2) > 1
```

```
[1] FALSE
```

```
print(.Machine$double.eps, digits=20)
```

```
[1] 2.2204460492503130808e-16
```

```
print(.Machine$double.eps/2, digits=20)
```

```
[1] 1.1102230246251565404e-16
```

```
print(1 + .Machine$double.eps, digits = 20)
```

```
[1] 1.0000000000000222
```

```
print(1 + .Machine$double.eps / 2, digits = 20)
```

```
[1] 1
```

- R provides a second value to measure the precision of the floating point implementation called `.Machine$double.neg.eps.`. This is the smallest value ϵ such that

$$1 - \epsilon < 1,$$

within the floating point system.

```
.Machine$double.neg.eps
```

```
[1] 1.110223e-16
```

- The most important effect of this is that certain numbers cannot be represented precisely within a floating point system. This is a machine-induced error called **round-off error**.
- Round-off error can be brought to prominence with a simple subtraction exercise (**loss of significance**):

```
20.55 - 19.2 - 1.35
```

```
[1] 1.332268e-15
```

```
20.55 - 1.35 - 19.2
```

```
[1] 0
```

```
print(20.55 - 19.2, digits=20)
```

```
[1] 1.3500000000000014211
```

```
print(20.55 - 1.35, digits=20)
```

```
[1] 19.199999999999999289
```

2.4.3 Dates and times

- The standard calendar is very complicated: months of different lengths, leap years every four years (with exceptions for whole centuries) and so on.
- Times are also messy, because there is often an unstated time zone (which may change for some dates due to daylight savings time).
- There have been several attempts to deal with this in R.
  - The base package has the function `strptime()` to convert from strings (e.g. `"2020-12-25"`, or `"12/25/20"`) to an internal numerical representation, and `format()` to convert back for printing.
  - Functions in the `xts` and `lubridate` are also useful.

2.4.4 Missing values and other special values

- The missing value symbol is `NA`. Missing values often arise in real data, but they can also arise because of the way calculations are performed.

```
some.evens <- NULL # creates a vector with no elements
some.evens[seq(2, 20, 2)] <- seq(2, 20, 2)
some.evens
```

```
 [1] NA  2 NA  4 NA  6 NA  8 NA 10 NA 12 NA 14 NA 16 NA 18 NA 20
```

- What happened here is that we assigned values to elements $2, 4, \cdots, 20$ but never assigned anything to elements $1, 3, \cdots, 19$, so R uses `NA` to signal that the value is unknown.
- Consider the following:

```
x <- c(0, 1, 2)
x/x
```

```
[1] NaN   1   1
```

- The `NaN` symbol denotes a value which is **not a number** which arises as a result of attempting to compute the indeterminate 0/0. This symbol is sometimes used when a calculation does not make sense. In other cases, special values may be shown, or you may get an error or warning message.

```
1/x
```

```
[1] Inf 1.0 0.5
```

Here R has tried to evaluate 1/0 and reports the infinite result as `Inf`.

- When there may be missing values, the `is.na()` function should be used to detect them. For instance,

```
is.na(some.evens)
```

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

The result is a **logical vector**. The `!` symbol means **not,** so we can locate the non-missing values in `some.evens` as follows:

```
!is.na(some.evens)
```

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

- We can then display the even numbers only:

```
some.evens[!is.na(some.evens)]
```

```
 [1]  2  4  6  8 10 12 14 16 18 20
```

Here we have used `logical indexing`.