Statistical programming using R

Lecture 1 Introduction, data types and structures

Introduction

R is a programming language and statistical software. It was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, in 1995 as a replacement of a commercial proprietary programming language S.

Advantages of R:

- open-source and free
- runs on Windows, Linux, Mac OS
- contains advanced statistical routines not yet available in other packages
- has state-of-the-art graphics capabilities
- allows for ready-to-print quality outputs
- vector-based language
- easy to start with and as such it is used in all areas of research - statistics, econometric, actuarial sciences, sociology, finance, marketing, health, epidemiology, etc.
- free!

Disadvantages of R:

- performance
- somewhat messy syntax
- there are always many different ways to do the same job

Programming style

Most commonly used assignment symbol is `<-`

```
x <- 2
```

There is a quick way to get a symbol `<-`, you need to press `Alt` + `-` (on Windows or Linux) or `Option` + `-` (on Mac OS).

There are two possible alternatives that do the same job of assignment a value to a variable

```
x = 2
2 -> x
```

Every new command typically starts on a new line

```r
x <- 2
y <- x + 3
```

However multiple commands can be put on a single line by using delimiter `;`.

```r
x <- 2; y <- x + 3
```

Don't forget to take care about readability of your code.

To improve readability of the code for others and for yourself when you get to the same code later, you should add comments using character `#`.

```r
# this is a comment

x <- 2    # this is a comment too
```

More about readability - don't use or at least try to limit use of blind variable names, like `x` or `y`. Always try to give your variables meaningful names, e.g.

```r
weight.in.kg <- 47       # dot is a legitimate symbol - a part of the name
weightInKg <- 47         # this is called "camelCase" or "camelBack" naming approach
weight_in_kg <- 47       # similar story to using dot in a variable name
```

Underscore is less popular in R and it should not be used as the first character of the name.

Remember, R is case sensitive, hence `x` and `X` are two different objects.

Data types

Data types in R are mostly the same as in Python and other programming languages:

- character: `"a"`, "`test`"
- numeric: `2`, `47.5`
- integer: `2L` (`L` is a special instruction to treat number `2` as an integer)
- logical: `TRUE`, `FALSE`
- complex: `1+2i`

To examine data you can use function `class()` for high-level and `typeof()` for low-level type of data

```r
a <- "abc"
class(a)
```

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

```r
typeof(a)
```

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

```r
b <- 2
class(b)
```

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

```r
typeof(b)
```

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

```r
d <- 2L
class(d)
```

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

```r
typeof(d)
```

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

In fact, *integer* is *numeric* too. There are multiple classes within *numeric* with *integer* and *double* being two most popular. *Integer*s are 32-bit numbers while *numeric*s are 64-bit doubles.

Try following examples

```r
x <- 2L
is.integer(x)
```

```
## [1] TRUE
```

```r
is.numeric(x)
```

```
## [1] TRUE
```

```r
y <- 2
is.integer(y)
```

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

```r
is.numeric(y)
```

```
## [1] TRUE
```

Also, you can check object size

```r
x <- 1:1000
is.integer(x)
```

```
## [1] TRUE
```

```r
object.size(x)
```

```
## 4048 bytes
```

```r
y <- as.numeric(1:1000)
is.integer(y)
```

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

```r
object.size(y)
```

```
## 8048 bytes
```

Data structures

There are many **data structures** in R with the most common:

- vector
- list
- matrix
- data frame
- factors

Vector

A vector is the most common and the most important data structure in R. As you remember, R is a vector-based language. All objects you've seen so far were vectors. For example, if you check the length of the object it might surprise you

```r
a <- "abcde"
length(a)
```

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

Length of `a` is one as `a` is a vector with one element, regardless of the length of that element itself.

In general, a vector is a collection of elements of the same type, that is a character or logical or integer or numeric but never a mix of different types.

Number of elements in a vector starts from zero (empty vector) and goes to whatever allowed by the memory limits in the current version of R and your computer operating system. Try to run `help("Memory-limits")` for more information.

How to create a vector:

```r
x <- vector() # an empty logical vector (default one)
x
```

```
## logical(0)
```

```r
length(x)
```

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

```r
y <- vector("character", length=5)
y
```

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

Now the vector is not empty, it has 5 elements, so its length is 5, and every element is a character.

Alternative ways to create vectors with specific data types:

```r
numeric(5) # a numeric vector with 5 elements
```

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

```r
logical(5) # a logical vector with 5 elements
```

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

```r
character(5) # a character vector with 5 elements
```

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

```r
integer(5) # an integer vector with 5 elements
```

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

You can create a vector by specifying its content using function `c()` which combines all arguments to form a vector.

```r
x <- c(1,2,3,4,5)
class(x)
```

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

```r
length(x)
```

```
## [1] 5
```

Please re-call a definition of a vector - it is a homogeneous collection of elements, elements of the same type. Hence, if you try to combine different data types they will be automatically converted.

```r
x <- c(1,2,3,4,"a")
x
```

```
## [1] "1" "2" "3" "4" "a"
```

```
class(x)
```

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

```
x <- c(1L,2L,3L,4L,5L)
x
```

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

```
class(x)
```

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

```
y <- c(x, 2.2)
y
```

```
## [1] 1.0 2.0 3.0 4.0 5.0 2.2
```

```
class(y)
```

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

All elements will be converted into whatever is *"easy"* to convert without losing information. For example, if you mix integers and doubles, then everything will be converted to doubles as converting doubles to integers might lead to a loss of information after decimal point. If you mix doubles or integers with characters, then everything will be converted to characters as it might be technically impossible to convert character to numeric.

```
c(1,2,3,4,"5")
```

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

You can change data type if you want and if you believe it is possible

```
x <- c(1,2,3,4,"5")
x
```

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

```
y <- as.numeric(x)
y
```

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

at the same time

```
x <- c(1,2,3,4,"abc")
x
```

```
## [1] "1"   "2"   "3"   "4"   "abc"
```

```
y <- as.numeric(x)
```

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

```
y
```

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

All but one elements were successfully converted and one element failed to convert to numeric.

As you could see above `c()` can be used to combine multiple objects. As every object in R is a vector, `c()` combines multiple vectors into a one vector. E.g.

```
x <- c(1,2)      # combining two numeric vectors of one element each
x                # result is a numeric vector of length two
```

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

```
y <- c(x,x,x)    # combining three vectors together
y                # and a result is a vector of 6 elements
```

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

```
y <- c(y,3)      # adding extra element to an existing vector
y                # and here is a result
```

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

Another way to create a vector is to use sequences

```
x <- 1:10      # a sequence of integers from 1 to 10 inclusive
x
```

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

Similar result you can get by `seq()`

```
x <- seq(10)  # again a sequence of integers starting from 1,
x             # which is a default behaviour of seq()
```

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

You can be more flexible with `seq()` and get any kind of numerical sequences.

```r
seq(from = 5, to = 10, by = 0.2)
```

```
##  [1]  5.0  5.2  5.4  5.6  5.8  6.0  6.2  6.4  6.6  6.8  7.0  7.2  7.4  7.6  7.8
## [16]  8.0  8.2  8.4  8.6  8.8  9.0  9.2  9.4  9.6  9.8 10.0
```

Missing data / Special values

Vectors always have the same type of data with the only exception - they can have missing data represented as `NA` (Not Available), which is a special value indicating missing data.

```r
x <- c(1,2,3,NA,5)
class(x)
```

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

```r
y <- c("a", "b", "c", NA, "e")
class(y)
```

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

```r
z <- c(TRUE, TRUE, FALSE, NA)
class(z)
```

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

The function `is.na()` check every element of the vector to be `NA`. Result will be a logical vector with `TRUE` for elements with missing data and `FALSE` otherwise

```r
x <- c(1,2,3,NA,5)
is.na(x)
```

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

There are other types of special values in R:

- `Inf` is infinity. It can be positive and negative

```r
1/0
```

```
## [1] Inf
```

```r
log(0)
```

```
## [1] -Inf
```

- `NaN` is Not a Number

```r
0/0
```

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

```r
log(-1)
```

```
## Warning in log(-1): NaNs produced
```

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

- `NULL` - is a null object. You can delete information by assigning `NULL` to a variable.

Object attributes

Every object can have attributes. Attributes are object specific and they are part of the object. Attributes include:

- `length()` - length of the object, that is, the number of elements
- `names()` - names of each element if any
- `dim()` - number of dimensions of the object
- `class()` - data type
- `nchar()` - number of characters in the every element of the object
- `attributes()` - provides a list of available attributes

```r
x <- c("a" = 1, "b" = 2, "c" = 3)
x
```

```
## a b c
## 1 2 3
```

This is a named vector. It is the same numeric vector as before `c(1,2,3)` but every element has its own name which can be used to extract the value of the element. More details will come later.

```r
class(x)
```

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

```r
length(x)
```

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

```r
names(x)
```

```
## [1] "a" "b" "c"
```

```r
attributes(x)
```

```
## $names
## [1] "a" "b" "c"
```

```r
y <- c("abc", "defgh")
nchar(y)
```

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

Every function can be used not just to read object attributes but to change them too.

```r
x <- c("a" = 1, "b" = 2, "c" = 3)
x
```

```
## a b c
## 1 2 3
```

```r
names(x) <- c("d", "e", "f")
x
```

```
## d e f
## 1 2 3
```

To access any element of the vector you can use an index or a name (for a named vectors) of the element

```r
x <- c("a" = 11, "b" = 12, "c" = 13)
x[2]        # get a value by an index
```

```
##  b
## 12
```

```r
x["b"]      # get a value by a name
```

```
##  b
## 12
```

You can assign new values to the elements of the vector using indexes or names

```r
x <- c("a" = 11, "b" = 12, "c" = 13)
x
```

```
##  a  b  c
## 11 12 13
```

```r
x["b"] <- 99   # assign value 99 to an element with name "b" of the vector "x"
x
```

```
##  a  b  c
## 11 99 13
```

Matrix

A matrix is a two-dimensional structure very similar to a vector. There is a number of rows and columns and all elements should be of the same type - numeric or character or logical, etc.

```r
m <- matrix(nrow = 2, ncol = 3) # empty matrix with 2 columns and 3 rows
m
```

```
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
```

```r
class(m)
```

```
## [1] "matrix" "array"
```

```r
typeof(m)   # by default empty matrix is logical the same way as empty vector
```

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

```r
dim(m)
```

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

There are two dimensions and the first one (rows) has two elements, while the second one (columns) has three elements. In R rows are always a first dimension.

```r
m <- matrix(c(1:6), 2, 3)
m
```

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

```r
class(m)
```

```
## [1] "matrix" "array"
```

```r
typeof(m)
```

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

By default, matrices in R are filled column-wise. However, you can change that behavior with the optional parameter `byrow` if you need it.

```r
m <- matrix(letters, ncol = 2, byrow = TRUE)       # filling row-wise
m
```

```
##       [,1] [,2]
##  [1,] "a"  "b" 
##  [2,] "c"  "d" 
##  [3,] "e"  "f" 
##  [4,] "g"  "h" 
##  [5,] "i"  "j" 
##  [6,] "k"  "l" 
##  [7,] "m"  "n" 
##  [8,] "o"  "p" 
##  [9,] "q"  "r" 
## [10,] "s"  "t" 
## [11,] "u"  "v" 
## [12,] "w"  "x" 
## [13,] "y"  "z"
```

```
class(m)
```

```
## [1] "matrix" "array"
```

```
typeof(m)
```

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

There are other ways to create a matrix. The first one is to start with a vector and then transform it into a matrix by changing its dimension attribute.

```
x <- 1:12
dim(x)
```

```
## NULL
```

```
dim(x) <- c(3, 4)
x
```

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

The second option is to combine two (or more) vectors by binding them together with functions `rbind()` (row bind) and `cbind()` (column bind). These functions work not just for matrices but for any objects that can create a two-dimensional structure.

```
x <- 1:4
y <- 5:8
cbind(x,y) # take vectors "x" and "y" as columns
```

```
##      x y
## [1,] 1 5
## [2,] 2 6
## [3,] 3 7
## [4,] 4 8
```

```r
rbind(x,y) # take the same vectors as rows
```

```
##   [,1] [,2] [,3] [,4]
## x    1    2    3    4
## y    5    6    7    8
```

Indexing of values in a matrix requires two indexes - first for a row and second for a column.

```r
m <- matrix(1:12, 3, 4)
m
```

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

```r
m[2,4]    # value on the row 2 and on the column 4
```

```
## [1] 11
```

Array

Array is a matrix with more than two dimensions. There might be as many dimensions as you want but all elements should be of the same data type.

```r
x <- 1:12
dim(x) <- c(2, 3, 2) # make a 3-dimensional array
x
```

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

```r
class(x)
```

```
## [1] "array"
```

```r
typeof(x)
```

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

To access values in an array, you need an index for each dimension

```r
x[2,3,2]    # an element in row 2, column 3 of the slice 2
```

```
## [1] 12
```

List

A list is a special type of a vector, it is collection of objects which don't need to be of the same type. Unlike a vector, every object in the list could be absolutely anything. A list can be a combination of vectors, matrices, arrays, other lists, etc. A list is a universal container for any type of data.

```r
x <- list(1:5, 2.5, "abcdef", TRUE)
x
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] 2.5
##
## [[3]]
## [1] "abcdef"
##
## [[4]]
## [1] TRUE
```

Similar to a vector, a list can be named. That is, every element of the list can have a name that can be used to retrieve data.

```r
x <- list(a = 1:5, b = 2.5, c = "abcdef", d = TRUE)
x
```

```
## $a
## [1] 1 2 3 4 5
##
## $b
## [1] 2.5
##
## $c
## [1] "abcdef"
##
## $d
## [1] TRUE
```

To address elements of the list you can use indexes in double square brackets or elements names if the list is named. Also, you can use a so-called dollar-sign notation (for named list only).

```r
x[[3]]    # list element number 3
```

```
## [1] "abcdef"
```

```r
x[["c"]]  # list element with name "c"
```

```
## [1] "abcdef"
```

```r
x$c       # the same as above - element with name "c"
```

```
## [1] "abcdef"
```

As mentioned before, a list is a special type of a vector. Hence, you can use an index (or a name) with a single square brackets but the result will be a list a single element. So, a list is a collection of one-element lists containing object of any data types. Here are examples:

```r
x <- list(a = 1:5, b = 2.5, c = "abcdef", d = TRUE)
x
```

```
## $a
## [1] 1 2 3 4 5
##
## $b
## [1] 2.5
##
## $c
## [1] "abcdef"
##
## $d
## [1] TRUE
```

```r
x[1]
```

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

```r
class(x[1])
```

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

```r
length(x[1])
```

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

```r
x[[1]]
```

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

```r
class(x[[1]])
```

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

```
length(x[[1]])
```

```
## [1] 5
```

Data frame

A data frame is the most important data structure for statistics and data analysis. A data frame is a *de facto* standard for tabular data used in statistics.

A data frame has a rectangular shape with rows and columns. Speaking statistically, every column is a variable (or attribute or category) and every row is an observation (or case or patient or respondent).

A data frame is a special type of the list where every element is a vector of the same length. Element of each column are of the same type as this is a vector. However, different columns can be of different type as a list can hold multiple type of data.

```
df <- data.frame(id = letters[1:10], x = 1:10, y = 11:20)
df
```

```
##    id  x  y
## 1   a  1 11
## 2   b  2 12
## 3   c  3 13
## 4   d  4 14
## 5   e  5 15
## 6   f  6 16
## 7   g  7 17
## 8   h  8 18
## 9   i  9 19
## 10  j 10 20
```

A data frame always have column names. If you try to create a data frame without providing column names, which is possible, column names will be created for you automatically. Rows can have names too but this is optional.

Now let's check some attributes of a data frame

```
names(df)
```

```
## [1] "id" "x"  "y"
```

```
class(df)
```

```
## [1] "data.frame"
```

```
typeof(df)
```

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

```r
dim(df)
```

```
## [1] 10  3
```

A data frame is a special type of list, which is a special type of a vector. Also, a data frame has two-dimensional structure as a matrix, which is a special type of a vector too. Hence, all types of indexing or accessing values you have seen before can be used with data frame.

```r
df["x"]    # single brackets with a name. Result is a data frame with one column.
```

```
##     x
## 1   1
## 2   2
## 3   3
## 4   4
## 5   5
## 6   6
## 7   7
## 8   8
## 9   9
## 10 10
```

```r
df[2]      # single brackets with an index. Result is the same as above - a data frame.
```

```
##     x
## 1   1
## 2   2
## 3   3
## 4   4
## 5   5
## 6   6
## 7   7
## 8   8
## 9   9
## 10 10
```

```r
df$id      # dollar-sign notation. Result is a vector of all values in the column.
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```r
df[["x"]] # double square brackets with a name. Result is as above.
```

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

```r
df[[2]]    # double square brackets with an index number. Result is as above.
```

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

```r
df[3, 2]  # individual indexes for row and column. Result is a vector of length 1.
```

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

```r
df[3,"x"] # the same as above but with the name instead of index.
```

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

When you access values in a vector, matrix, array, list or data frame names or indexes should not be single values. Moreover, you remember that there are no single values in R - there are always vectors. Hence, you can access or change multiple values at the same time.

```r
x <- c("a", "b", "c", "d", "e", "f")
x
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```r
x[c(1,3,6)]   # an index is a vector with three elements
```

```
## [1] "a" "c" "f"
```

```r
m <- matrix(1:12, nrow=3, ncol=4)
m
```

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

```r
m[c(1,2), c(1,3)]  # indexes for rows and columns are vectors
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
```

```r
df <- data.frame(id = letters[1:10], x = 1:10, y = 11:20)
df
```

```
##    id  x  y
## 1   a  1 11
## 2   b  2 12
## 3   c  3 13
## 4   d  4 14
## 5   e  5 15
## 6   f  6 16
## 7   g  7 17
## 8   h  8 18
## 9   i  9 19
## 10  j 10 20
```

```r
df[c(1,2), c(1,3)] # the same story as with a matrix above
```

```
##   id  y
## 1  a 11
## 2  b 12
```

```r
df[c(1,2), c("id", "y")] # now there is a vector of names for columns
```

```
##   id  y
## 1  a 11
## 2  b 12
```

When you deal with two-dimensional (or higher) data structures and you don't want to make a selection along any particular dimension then you can just drop that index.

```r
m <- matrix(1:12, nrow=3, ncol=4)
m
```

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

```r
m[ , c(1,3)]  # to take all rows for columns 1 and 3
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
```

```r
df <- data.frame(id = letters[1:10], x = 1:10, y = 11:20)
df
```

```
##    id  x  y
## 1   a  1 11
## 2   b  2 12
## 3   c  3 13
## 4   d  4 14
## 5   e  5 15
## 6   f  6 16
## 7   g  7 17
## 8   h  8 18
## 9   i  9 19
## 10  j 10 20
```

```r
df[ , c("id", "y")] # all rows for columns "id" and "y"
```

```
##    id  y
## 1   a 11
## 2   b 12
## 3   c 13
## 4   d 14
## 5   e 15
## 6   f 16
## 7   g 17
## 8   h 18
## 9   i 19
## 10  j 20
```

```
df[c(1,3), ]  # to take rows 1 and 3 for all columns in the data frame
```

```
##   id x  y
## 1  a 1 11
## 3  c 3 13
```

There are many data sets embedded in R that can be used as examples. Let's have a look on one of them

```
mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

… and try some functions for dealing with data frames

```
head(mtcars)   # first 6 rows of the data frame. You can change the number of rows.
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
tail(mtcars)   # last 6 rows of the data frame
```

```
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

```
ncol(mtcars)   # number of columns
```

```
## [1] 11
```

```
nrow(mtcars)   # number of rows
```

```
## [1] 32
```

```
dim(mtcars)    # dimensions, that is number of rows and columns
```

```
## [1] 32 11
```

```
names(mtcars)  # column names of the data frame
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
colnames(mtcars) # again, column names
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
rownames(mtcars) # row names of the data frame
```

```
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
```

```
str(mtcars)     # structure - name, type and preview of every column
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Factors

The last data structure to mention is a factor. It is used to efficiently store and process categorical data.

```
colour <- c("red", "blue", "red", "red")
colour     # This is a character vector
```

```
## [1] "red"  "blue" "red"  "red"
```

```
colour_f <- factor(colour)
colour_f   # This is a factor
```

```
## [1] red  blue red  red
## Levels: blue red
```

A factor is an integer vector with all characters coded alphabetically (as it is common in R you can change everything).

```
as.integer(colour_f) # actual data stored as integers
```

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

```
nlevels(colour_f)    # number of levels in the data
```

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

```
levels(colour_f)     # what are the levels
```

```
## [1] "blue" "red"
```

Storing character data as a factor is so efficient computationally that R used to do it by default. Each time you create a data frame or load data from external sources, R automatically converts character columns to factors. Most data analytic or predictive functions in R expect that numerical variables presented as *numeric* and categorical variables presented as *factors*. However, in some situations that behavior is not desirable and you have to say explicitly that you want to keep characters as characters.

```
df <- data.frame(id = letters[1:10], x = 1:10, y = 11:20, stringsAsFactors = FALSE)
df$id
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
class(df$id)
```

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

Categorical variables can be nominal or ordinal. Categorical nominal does not have any order, e.g. “red”, “blue”, “green” or “male” and “female”. None of the values is more important or goes ahead of others. Categorical ordinal has an order, some values are above the others, e.g. “high”, “medium”, “low” or “good”, “bad”, “ugly”.

In a similar fashion, factors can be ordered and non-ordered. For non-ordered factors (default behavior) levels assigned alphabetically. For ordered factors, you have to nominate levels and set parameter `ordered = TRUE`.

```
movies <- c("good", "bad", "ugly", "ugly", "good", "good")
movies_f <- factor(movies)
levels(movies_f)    # levels were assigned alphabetically
```

```
## [1] "bad"  "good" "ugly"
```

```
movies_f <- factor(movies, levels = c("ugly", "bad", "good"))
levels(movies_f)    # levels were manually assigned
```

```
## [1] "ugly" "bad"  "good"
```

```
min(movies_f)       # does not work as our factor is non-ordered, there is no min or max
```

```
Error in Summary.factor(c(3L, 2L, 1L, 1L, 3L, 3L), na.rm = FALSE) : ‘min’ not meaningful for factors
```

```r
# let's try again
movies_f <- factor(movies, levels = c("ugly", "bad", "good"), ordered = TRUE)

levels(movies_f)    # levels were manually assigned and order was set
```

```
## [1] "ugly" "bad"  "good"
```

```r
min(movies_f)       # now it works as "ugly" is the lowest level
```

```
## [1] ugly
## Levels: ugly < bad < good
```

Logical indexing

You already know how to address data by using indexes and names. There is one more way - using logical or Boolean variables.

```r
x <- seq(10)
x      # integer vector
```

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

```r
y <- c(TRUE,TRUE,TRUE,TRUE,TRUE,FALSE,FALSE,FALSE,FALSE,FALSE)
y      # logical vector
```

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

```r
x[y]   # select values from "x" that correspond to TRUE in "y"
```

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

A logical indexing works for all data structures - vectors, lists, matrices, data frames. You have to pay attention that length of the logical vector used for indexing is the same as the length of the object you try to index.

Logical vectors can be created using relational operators: `<`, `>`, `<=`, `>=`, `==`, `!=`, `%in%`.

```r
df <- data.frame(x = c(1,-2,3,-4,5,-6,7,-8,9,-10), y = letters[1:10])
df
```

```
##      x y
## 1    1 a
## 2   -2 b
## 3    3 c
## 4   -4 d
## 5    5 e
## 6   -6 f
## 7    7 g
## 8   -8 h
## 9    9 i
## 10 -10 j
```

```r
df[df$x > 0, ]     # select only rows where value in column "x" is positive
```

```
##   x y
## 1 1 a
## 3 3 c
## 5 5 e
## 7 7 g
## 9 9 i
```

```r
df[df$y %in% c("a", "e", "i"), ] # select row if column "y" value is a vowel
```

```
##   x y
## 1 1 a
## 5 5 e
## 9 9 i
```

Logical indexing as other types of indexing can be used for assignment data as well

```r
x <- c(1,-2,3,-4,5,-6,7,-8,9,-10)
x[x < 0] <- 99  # select all values that less than zero and set them to 99
x
```

```
##  [1]  1 99  3 99  5 99  7 99  9 99
```

Control flow

Control flow in R is very similar of it in Python. There are loops and selections.

Loops

There are three types of loops:

- for
- while
- repeat

IMPORTANT NOTE! Loops are very inefficient in R. Always try to avoid them if possible, instead use vectorisation.

```r
x <- 1:10
x * 2    # multiply every value in x by 2
```

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

```r
y <- 11:20
x + y    # sum up corresponding values of "x" and "y"
```

```
##  [1] 12 14 16 18 20 22 24 26 28 30
```

Loop `for`

The same job as above can be done using `for` loop

```
res <- numeric(length = length(x)) # prepare empty vector of the size 10
for(i in seq(1, length(x))){       # iterate through number from 1 to 10
  res[i] <- x[i] + y[i]            # do summation for paired values of x and y
}
res                                # output results
```

```
##  [1] 12 14 16 18 20 22 24 26 28 30
```

Another example

```
x <- 1:10                 # initial data set
res <- c()                # prepare empty vector of length zero
for(i in x){              # iterate through all elements of x
  res <- c(res, i * 2)    # add new value to the end of result vector
}
res                       # output results
```

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

Result is the same as before in `x * 2`, however this approach is the least computationally efficient.

- First, it uses a loop while vectorisation is possible.
- Second, it changes the length of vector `res` on every iteration - that takes a lot of resources. The previous example is much better in this sense. It creates an empty numeric vector of the right length and then changes values inside the vector but not the vector itself.

Loop `while`

A loop `for` will run for some predefined number of times - once for an every element of a sequence `seq(1, length(x))` or a vector `x`. However, sometimes you might not know how many times you plan to run a given set of expressions. A loop `while` can help here. It runs till a given condition remains `TRUE`

```
x <- 0           # initial value of x
while(x < 5){    # repeat while x is less than 5
  print(x)       # print to control the process
  x <- x + 1     # change value of x
}
```

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

A loop `while` can be dangerous. You have to be sure that a condition in the loop eventually becomes `FALSE`. Otherwise, you get an infinite loop that never stops.

Loop `repeat`

A `repeat` loop in R is very similar to a loop `while` and it is even more dangerous than a `while` loop as it has no condition to check. So, `repeat` loop will do precisely that - repeating a given set of expressions until something *breaks* the loop.

```
x <- 0         # optional value to start with
repeat{        # keep repeating till break command
  print(x)     # do something
  x <- x + 1   # these 4 lines create a stopping condition
  if(x > 5){
    break
  }
}
```

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

Operator `break`

An operator `break` can be used in any type of loops to break the loop even if condition in a `while` loop is still satisfied (`TRUE`) or a sequence in a `for` loop is not exhausted yet.

```
x <- 1:10
res <- c()
for(i in x){
  res <- c(res, i * 2)
  if(i > 5){      #
    break
  }
}
res
```

```
## [1]  2  4  6  8 10 12
```

Conditions

Conditions if, if-else, if-else-if

Conditions in R work the same way as in Python: `if(condition to check) {statement}`. Below are some examples.

The simplest version - if the condition is `TRUE` then R execute codes inside `{}`; if the condition is `FALSE` then R skips `{}` and continues with the code after `{}`.

```
x <- 10
if(x < 20){
  print(x)
}
```

```
## [1] 10
```

You can have some statements to execute not only for `TRUE` condition but the `FALSE` case too.

```
x <- 10
if(x < 20){
  print(x)
} else {
  print(x / 10)
}
```

```
## [1] 10
```

You can have a multilevel hierarchical structure of many conditions

```
x <- 10
if(x < 5){
  print("too small")
} else if(x > 20){
  print("too large")
} else {
  print("just right")
}
```

```
## [1] "just right"
```

While you see three different `print()` functions, only one will be executed depending on the value of `x` and as a result of what conditions will be `TRUE` or `FALSE`.

Function `ifelse()`

It is useful to mention here a function `ifelse()`. It is not a control flow statement, it is a function, but it does a similar job.

```
x <- 10
ifelse(x < 20, print(x), print(x / 10))
```

```
## [1] 10
```

```
## [1] 10
```

If condition above is `TRUE` then the first statement `print(x)` will be executed; if `FALSE` the the second one `print(x / 10)`. An important consequence if `ifelse()` being a function is that it returns a result which can be assigned to a variable. And this is an explanation why you see output `10` twice in the example above. The first output is `print()` function execution, the second one is a result of `ifelse()` function.

```
x <- 10
y <- ifelse(x < 20, x * 10, x / 10)
print(y)
```

```
## [1] 100
```

Function `switch()`

A function `switch()` is an alternative to multilevel `if-else-if` statements. You can cycle through many possible options and return the "right" one.

```
x <- "sad"
y <- switch(x,
            happy = "I am glad you are happy",
            afraid = "There is nothing to fear",
            sad = "Cheer up",
            angry = "Calm down now",
            "Nothing to say")     # this is a default output
                                  # if "x" does not match any conditions above
print(y)
```

```
## [1] "Cheer up"
```

Functions

User-defined functions

There is a huge number of built-in functions in R. And the number is really HUGE. In fact, no one knows how many functions in R as people keep adding new functions every day. Still, you might need to create your own functions doing something specific to your project and your individual needs. You know all advantages of custom functions: more manageable program development, simpler and easier to understand code, code re-usability, minimised code duplication within the program, better testing, etc.

Format of user-defined function in R is following:

```
function_name <- function(argument1, argument2, ...){
  statement1
  statement2
  ...
  return(object)
}
```

For example, here is a function to convert US miles to kilometers

```
miles.to.kilometers <- function(miles){
  kilometers <- miles * 1.60934
  return(kilometers)
}
```

Now, each time you need to convert miles to kilometers you just run this function

```
miles.to.kilometers(250)    # mileage range of Tesla Model 3 Standard Range Plus version
```

```
## [1] 402.335
```

```r
miles.to.kilometers(322)    # mileage range of Tesla Model 3 Long Range version
```

```
## [1] 518.2075
```

Function parameters can be filled by name and by position, they can be optional and compulsory. All the same ideas as in Python user-defined functions.

```r
miles.to.kilometers <- function(miles, n.round=1){
  kilometers <- round(miles * 1.60934, n.round)
  return(kilometers)
}

# rounding to one decimal place only (default parameter)
miles.to.kilometers(322)
```

```
## [1] 518.2
```

```r
# rounding to two decimal places, parameters order is important
miles.to.kilometers(322, 2)
```

```
## [1] 518.21
```

```r
# if using names then parameters order is not important
miles.to.kilometers(n.round=3, miles=322)
```

```
## [1] 518.207
```

Important reminder about a scope of variables: variables created inside the function have a local scope and they are not available outside the function. If you try to call for `kilometers` or `miles`, you get an error message.

Built-in functions

The main "power" of R comes from packages that extends its capabilities. When you just started R, several packages were loaded automatically and all functions you used so far were from these packages. You can see the list of loaded packages:

```r
print(.packages())
```

```
## [1] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "methods"  
## [7] "base"
```

There are way too many functions in R especially considering more than 15,000 packages. It is impossible to remember them all. Let's have a look on some functions only - the most common or the most useful.

Getting help in R

As you can not memorise all functions, you should be able to find a help on any function you need.

```
help.start()    # opens a HTML document with major manuals
                # and reference list of all packages

help(switch)    # opens a help-file for function "switch"
?(switch)       # the same as above

help(package="base")  # opens documentation for chosen package, e.g. "base"

vignette()      # gives a list for all available vignettes (tutorials)
                # for all installed packages

vignette(topic="dplyr", package="dplyr")  # opens selected vignetts from selected package
```

The best way to find help is to Google. All manuals for all functions and packages are available online. Numerous tutorials, examples, FAQ, etc, are available online too. Ask Google, e.g. "how to open csv file in R" and you get a lot of instructions.

Basic functions

Here is a short example of some functions that are loaded automatically and always available in R

```
# generate 100 random numbers from normal distribution with given mean and standard deviation
x <- rnorm(100, mean = 10, sd = 2)

# descriptive statistics
mean(x)
```

```
## [1] 10.1993
```

```
sd(x)
```

```
## [1] 2.070667
```

```
median(x)
```

```
## [1] 10.11881
```

```
IQR(x)
```

```
## [1] 2.48354
```

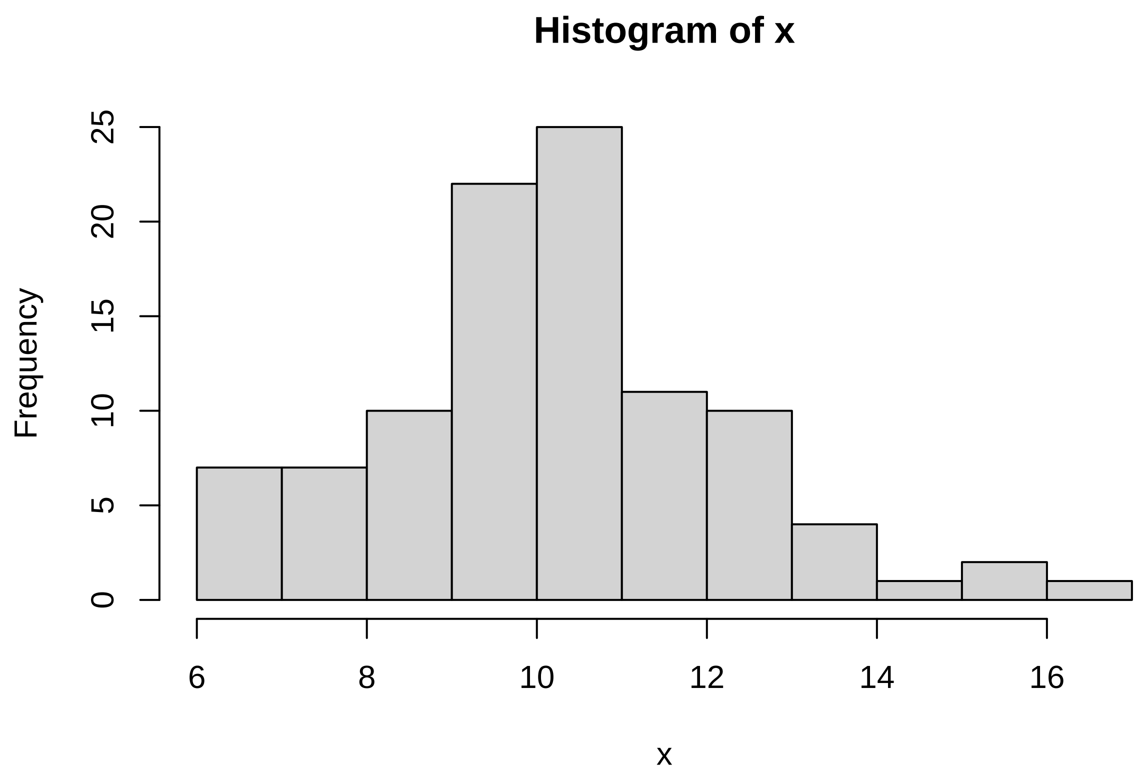
```
quantile(x, c(0, 0.25, 0.5, 0.75, 1))
```

```
##        0%       25%       50%       75%      100%
##  6.031015  9.079756 10.118812 11.563296 16.342306
```

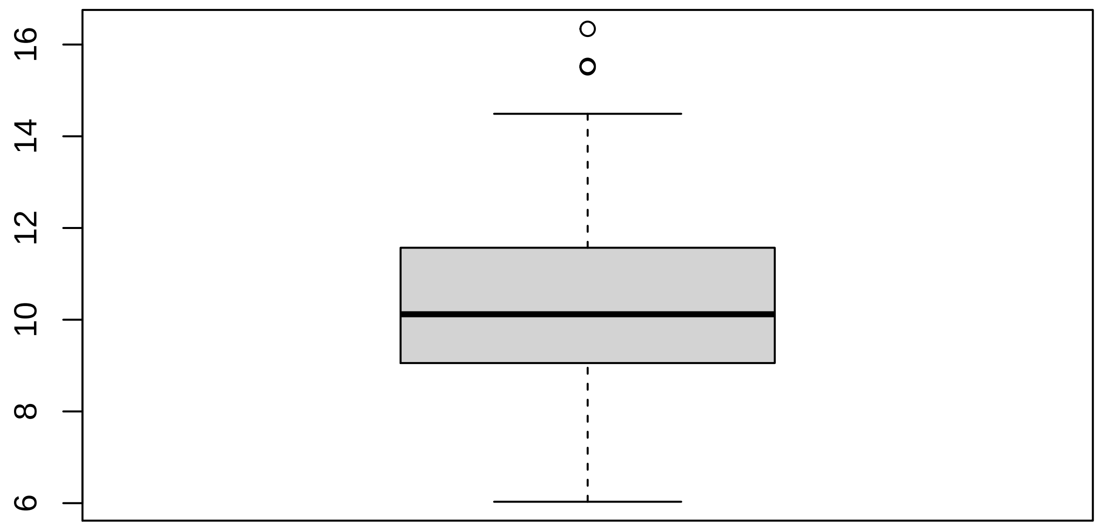
```
# 5-poit descriptive statistics by one function
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   6.031   9.080  10.119  10.199  11.563  16.342
```

```
# plotting - histogram
hist(x)
```



```
# plotting - boxplot
boxplot(x)
```



```
# t-test that mean equal 10
t.test(x, mu=10)
```

```
##
##  One Sample t-test
##
## data:  x
## t = 0.96251, df = 99, p-value = 0.3381
## alternative hypothesis: true mean is not equal to 10
## 95 percent confidence interval:
##   9.788438 10.610168
## sample estimates:
## mean of x
##   10.1993
```

You will see more a bit later. Now you can have a look on some reference guides available online:

- https://cran.r-project.org/doc/contrib/Short-refcard.pdf
- https://rstudio.com/wp-content/uploads/2016/10/r-cheat-sheet-3.pdf

Packages

To use any package you need to load it into the session and before loading you need to install it (if it is not yet available on your computer)

```
# download and install package into the default location
install.packages("dplyr")

# load the package into the session, making all its functions available to use
library(dplyr)
```

As there are so many packages, it is very common to have a conflict between packages when the same function name use in different packages. To avoid problems you should limit a number of packages you load. Load only packages you really need for the code you run.

Also, you can specify what particular function from what package you want to run. In this case you don't need to load the package.

```
dplyr::select()    # function "select" from package "dplyr"
```

Finding the right package for the job is not easy. There might be several packages able to do the same operations. The positive side is an option to choose: if you don't like some package, if you don't find it user-friendly, there is a very good chance that you can find another one with similar (or better) functionality.

To look for packages you can use Google as for anything R-related. Or you can check *CRAN Task Views* https://cran.r-project.org/web/views/ This is a collection of R packages sorted by the topic of research. You can install any individual package from the list or you can install all packages within a View.

Anonymous functions

Sometimes you might want to create a custom function for a single use. You don't plan to use this function again and you don't want to go through a "normal" way of defining a function with the name and `return` statement. Later you will see more example when this "lazy" function might be very useful. Here is a small preview based on use of build-in function `sapply`

```r
# create a data frame
df <- data.frame(Column.A=rnorm(100), Column.B=rnorm(100))
head(df)
```

```
##      Column.A   Column.B
## 1  0.04162114  1.3366736
## 2 -0.40900746 -0.8913365
## 3 -0.62365826  0.9140609
## 4  1.32485597  0.1582932
## 5  2.06373481  1.8249187
## 6  0.81531798 -0.2219813
```

```r
# apply built-in function max() to every column
sapply(df, max)
```

```
## Column.A Column.B
## 2.582803 2.122063
```

```r
# apply built-in function min() to every column
sapply(df, min)
```

```
##  Column.A  Column.B
## -2.037951 -2.687856
```

```r
# apply a custom anonumous function calculating a difference
sapply(df, function(x) max(x) - min(x))
```

```
## Column.A Column.B
## 4.620754 4.809919
```

You could do the job above by creating and applying a “normal” function. Also, you could do the same job manually:

```r
sapply(df, max) - sapply(df, min)
```

```
## Column.A Column.B
## 4.620754 4.809919
```

Results are the same. However, using anonymous function is a more elegant and more powerful way.