Introduction to R

STAT213

Let us start with using R as a fancy calculator.

```
# computing the sum of 5 and 8
5 + 8
```

```
## [1] 13
```

Here the + symbol is a function that calculates the sum of the object before and after the +. For many functions in R a help page exists, which can be called with the function `help("+")` or shorter by `?"+"`.
The line starting with the `#` contains a comment and everything in the line after the `#` will not be evaluated in R. It is a good idea to use comments to explain the code to others, but they are also useful for yourself when you revisit your code after a while. Although they are not necessarily needed, adding empty spaces around the + makes the code much easier to read.
We have to be careful about any typos, e.g. R is case-sensitive and distinguishes between capital- and small-case letters. If something goes wrong, most of the time R will show a more or less informative error message. Each command needs to be complete, e.g. `5 +` will still wait for an input that should be added to the number 5. In the terminal, you will see the `+` sign instead of the `>`, which means that the line of code is not complete and can be continued in the next line.

The function + can also be evaluated in a different format.

```
"+"(5, 8)
```

```
## [1] 13
```

Most of the functions in R are specified in the same way, starting with the function name `"+"`, followed by a pair of round brackets. Within the round brackets several arguments can be defined, separated by a comma.

Objects and classes

The numbers 5 and 8, which were summed up in the previous code chunk, are objects of the class numeric. We can find out about the class of an object by another function `class()`.

```
class(8)
```

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

```
class(log)
```

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

```
class("A")
```

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

The class of the number 8 in R is numeric, whereas the object `log` is a function, which computes logarithms of a value. The class of the letter A is a character; character strings are always written within quotation marks, either double `"A"` or single `'A'`.

Some constants like `pi` are directly built into R.

We can define an object with the number 8 as an integer instead of numeric by adding the letter `L`.

```
class(8L)
```

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

Assignment operator

When we call the name of an object, for example the constant π, then we will print the value of this object into the terminal.


```
pi
```



```
## [1] 3.141593
```

This is the same as using the function `print(pi)`. Hence, by calling any function, we create a new object that is printed in the terminal, e.g. computing the logarithm of 5.


```
log(5)
```

```
## [1] 1.609438
```

If we want to use the result of this calculation any further, we should assign the resulting value to a name. Assignments can be performed by the `<-` operator. The name should not contain any special characters or empty spaces, and it should not start with a number.

```
x <- log(5)
print(x)
```

```
## [1] 1.609438
```

Now the object `x` can be used like any other object in R, e.g. adding 3 to the logarithm of 5.

```
x + 3
```

```
## [1] 4.609438
```

When a new object is assigned to the name `x`, the previous content in `x` is overwritten.

```
print(x)
```

```
## [1] 1.609438
```

```
x <- 1
print(x)
```

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

When we add round brackets around an assignment operation, then the object will be automatically printed in the terminal.

```
(x <- 1 + 1)
```

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


Logical and relational operators

Instead of using R only as a calculator, we can also compare values using binary operators. For example, we could ask R if 3 is smaller than 5.


```
(x <- 3 < 5)
```



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

An object of class logical is returned, that can either take the values TRUE or FALSE. We can calculate with objects of class logical, assuming that TRUE is equal to 1 and FALSE equal to 0.

```
TRUE + TRUE + FALSE
```

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

The `!` operator indicates a logical negation.

```
!TRUE
```

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

Additional relational operators are available.

```
3 > 5   # larger
```

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

```
3 <= 5  # smaller or equal
```

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

```
3 == 5  # equal
```

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

```
3 != 5  # not equal
```

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

We have to be a bit careful about checking equality of two objects. Any calculation might introduce some machine error that might cause a difference to what we would expect as a result.

```
sqrt(2)^2
```



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



```
2 == sqrt(2)^2
```



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

Two objects of class logical can be connected by logical operators.

```
TRUE & TRUE   # AND
```

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

```
TRUE & FALSE  # AND
```

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

```
TRUE | FALSE  # OR
```

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

While the single `&` and `|` are element-wise operators the double `&&` and `||` perform left-to-right examination.

Vectors, arrays, lists, and data.frames

Vectors

We can combine multiple objects in a vector, using the function `c()`.

```
(x <- c(3, 7, -1, 6.3))
```

```
## [1]  3.0  7.0 -1.0  6.3
```

Any arithmetic operation (`+`, `-`, `*`, `/`, `^`) will be performed element-wise.

```
x + 2
```

```
## [1] 5.0 9.0 1.0 8.3
```

```
y <- c(-2, -6, 2, -5.3)
x + y
```



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


We have to be a bit careful, when the dimensions of two objects do not match. In this case, R will cycle through the shorter object, starting again with the first element.

```
c(0, 0, 0, 0, 0, 0, 0) + c(1, 2, 3)
```



```
## Warning in c(0, 0, 0, 0, 0, 0, 0) + c(1, 2, 3): longer object length is not a
## multiple of shorter object length
```

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

Instead of element-wise multiplication, the inner product or dot product of two vectors (or matrices) is computed with the `%*%` operator.

```
x %*% y
```

```
##        [,1]
## [1,] -83.39
```

We can combine objects with different classes, but all elements in the resulting vector will be converted to the same class.

```
# numeric and logical class converted to character
(x <- c(2, "A", TRUE))
```

```
## [1] "2"    "A"    "TRUE"
```

```
class(x)
```

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

```
# logical and integer converted to numeric
(y <- c(2, TRUE, 7.3))
```

```
## [1] 2.0 1.0 7.3
```

```
class(y)
```

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

Character vectors

Instead of numeric or logical objects, a vector can also contain characters.


```
(x <- c("A", "B", "C"))
```

```
## [1] "A" "B" "C"
```


There are several functions available to modify character objects.

```
# paste character objects together, separated by :
paste("A", "B", "C", sep=":")
```

```
## [1] "A:B:C"
```

```
(y <- paste(x, collapse=":"))
```

```
## [1] "A:B:C"
```

```
# substitute character
(z <- sub("B", "D", y))
```

```
## [1] "A:D:C"
```

```
# separating the character object again
strsplit(z, split=":")
```

```
## [[1]]
## [1] "A" "D" "C"
```

Factors

A special class for a vector is the factor class. A factor groups similar elements in to categories, which are the levels of the factor.

```
(x <- factor(c("A", "A", "B", "B", "A"), levels=c("A", "B")))
```

```
## [1] A A B B A
## Levels: A B
```

Sequences

There are some convenience functions available, automising the construction of vectors. A sequence of values can be constructed with the function `seq()`.

```
# sequence of integers from 1 to 10 (increment of 1)
seq(1, 10, by=1)
```



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



```
# short alternative form
1:10
```



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

```
# sequence of 11 elements from 5 to 1
seq(5, 1, length=11)
```

```
##  [1] 5.0 4.6 4.2 3.8 3.4 3.0 2.6 2.2 1.8 1.4 1.0
```

A vector can be repeated several times with the function `rep()`.

```
# repeating c(1,2,3) two times
rep(1:3, times=2)
```

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

```
# repeating each element in c(1, 2, 3) two times
rep(1:3, each=2)
```

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

Matrices, Arrays

There are several ways to construct a matrix in R.

```
# using function matrix,
# filling the matrix column-wise with integers from 1 to 6
# only the number of rows or the number of columns need to be given,
# the remaining number of columns or rows respectively
# are based on the number of elements (1:6)
(A <- matrix(1:6, nrow=3))
```

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

```
# combination of column vectors
A <- cbind(1:3, 4:6)
# combination of row vectors
A <- rbind(1:2, 3:4, 5:6)
```


The function `array()` works in the same way as a matrix, but allows for higher dimensions. Thus, a matrix is a two-dimensional array.

Similar to a vector, all arithmetic operations are performed element-wise. The `%*%` can be used to compute the dot product of two matrices.


```
A + 1
```

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

```
(B <- t(A) %*% A) # t() computes the transpose of a matrix
```

```
##      [,1] [,2]
## [1,]   35   44
## [2,]   44   56
```

Further matrix operations are available.

```
dim(B)   # dimensions
```

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

```
diag(B)  # diagonal elements
```

```
## [1] 35 56
```

```
det(B)   # determinant
```

```
## [1] 24
```

```
solve(B) # inverse
```

```
##           [,1]      [,2]
## [1,]  2.333333 -1.833333
## [2,] -1.833333  1.458333
```

The `diag()` fuction can also be used to construct a diagonal matrix based on the number of rows/columns or on a vector containing the elements on the diagonal.

```
diag(4)
```

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



```
diag(1:4)
```

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

data.frames

A matrix can only have elements of class numeric or logical; instead we might be interested in an object, that combines several types of variables in its columns. Multiple vectors with different classes can be combined to an object of class data.frame.

```
dfr <- data.frame(x = c(5, 2, 7, 4),
                  y = c(TRUE, FALSE, FALSE, TRUE),
                  z = factor(c("A", "B", "A", "B")))
print(dfr)
```

```
##   x     y z
## 1 5  TRUE A
## 2 2 FALSE B
## 3 7 FALSE A
## 4 4  TRUE B
```

Lists

When we want to collect any multiple objects possibly of different dimensions together, we can combine them as an object of class list. The list object works similar to a shopping list, where every item is stored in a different slot.

```
listobj <- list(x = c(3, 7, 2),
                A = diag(c(9, 2, 1)))
print(listobj)
```

```
## $x
## [1] 3 7 2
##
## $A
##      [,1] [,2] [,3]
## [1,]    9    0    0
## [2,]    0    2    0
## [3,]    0    0    1
```


Indexing

When we want to extract certain elements from an object, we have to indicate their position. An element is extracted from an object by adding the information about the location in square brackets `[i]`. The location is either the number of the position in a vector, or a logical vector, where elements corresponding to TRUE will be extracted. Alternatively, we can remove an element from the vector with a negative number as an index or a logical vector showing FALSE at the position that should be omitted.

```
x <- c(2.1, 1.5, 6.7, 3.8, 4.2)
# extracting the 3rd element
x[3]
```

```
## [1] 6.7
```

```
# extracting element 3 and 5
x[c(3, 5)]
```

```
## [1] 6.7 4.2
```

```
x[c(FALSE, FALSE, TRUE, FALSE, TRUE)]
```

```
## [1] 6.7 4.2
```

```
# alternatively removing elements 1, 2, and 4
x[c(-1, -2, -4)]
```

```
## [1] 6.7 4.2
```

For objects of higher dimensionality, each index vector is separated by a comma. For example for a matrix or data.frame, within square brackets, first, the rows and then the columns are selected. If no index vector is given, the full row or column is selected.

```
A <- matrix(1:6, nrow=3)
# selecting element in row 3 column 2
A[3, 2]
```

```
## [1] 6
```

```
# selecting every element in row 3
A[3, ]
```

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



```
# removing row 1
A[-1, ]
```



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


When only a single row or column vector is removed from a matrix, the resulting object is automatically a vector without additional information about any other dimension.

```
# extracting column 2
(x1 <- A[, 2])
```

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

```
dim(x1)
```

```
## NULL
```

The argument `drop=FALSE` ensures that the information about the previous number of dimensions is not dropped from the resulting object.

```
(x2 <- A[, 2, drop=FALSE])
```

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

```
dim(x2)
```

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

When we extract a slot from a list with the square bracket function, the resulting object has still the class list, containing the indicated slot.

```
listobj <- list(x = c(3, 7, 2),
                A = diag(c(9, 2, 1)))
# extract the 2nd slot of the list
(x1 <- listobj[2])
```

```
## $A
##      [,1] [,2] [,3]
## [1,]    9    0    0
## [2,]    0    2    0
## [3,]    0    0    1
```



```
class(x1)
```

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

With double square brackets instead, we are able to directly extract the object ot of the slot of the list.

```
(x2 <- listobj[[2]])
```

```
##      [,1] [,2] [,3]
## [1,]    9    0    0
## [2,]    0    2    0
## [3,]    0    0    1
```

```
class(x2)
```

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

The same applies to extracting columns from a data.frame object. With single square brackets, a column is extracted and the resulting object has still the class data.frame.

```
dfr <- data.frame(x = c(5, 2, 7, 4),
                  y = c(TRUE, FALSE, FALSE, TRUE),
                  z = factor(c("A", "B", "A", "B")))
# extracting column 2 from the data.frame
dfr[2]
```

```
##       y
## 1  TRUE
## 2 FALSE
## 3 FALSE
## 4  TRUE
```

With double square brackets instead, the resulting object is a vector.

```
dfr[[2]]
```

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

But, we can also extract columns and rows by supplying an index vector for each dimension. Then the resulting object is a vector, when `drop=FALSE`, similar to extracting rows and columns from a matrix.


```
dfr[,2]
```



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


Names and attributes

Sometimes, we want to give elements in an object a name. We can do this with the function `names()`. This function is interesting, because you can directly assign a character vector to a function call.

```
x <- c(2.1, 1.5, 6.7)
names(x) <- c("Justus", "Peter", "Bob")
print(x)
```

```
## Justus  Peter    Bob
##    2.1    1.5    6.7
```

The name can be used similar to an index for extracting an element from an object.

```
x["Bob"]
```

```
## Bob
## 6.7
```

The names of a data.frame object can be changed in a similar way.

```
names(dfr) <- c("Justus", "Peter", "Bob")
print(dfr)
```

```
##   Justus Peter Bob
## 1      5  TRUE   A
## 2      2 FALSE   B
## 3      7 FALSE   A
## 4      4  TRUE   B
```

But we can also give unique names for each row.

```
rownames(dfr) <- c("a", "b", "c", "d")
print(dfr)
```

```
##   Justus Peter Bob
## a      5  TRUE   A
## b      2 FALSE   B
## c      7 FALSE   A
## d      4  TRUE   B
```


We can also attach a list as an attribute to another object.

```
attributes(dfr) <- list(myattribute=x)
attributes(dfr)
```



```
## $myattribute
## Justus  Peter    Bob
##    2.1    1.5    6.7
```


Missing data and undefined values

The null object in R is defined as `NULL`. Any missing observations are defined as `NA`, not available.

```
NA + 5
```

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

```
# check if an object is equal to NA
x <- c(2, NA, 0)
is.na(x)
```

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

R has also special values for `NaN` (Not a Number) and `Inf` (infinity).

```
1/0
```

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

```
log(-1)
```

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

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

Functions

For many tasks there is already a specific function available in R, for example calculating the sum of elements in a numeric vector.

```
x <- c(2, 7, 1, 3)
sum(x)
```

```
## [1] 13
```


But R allows us also to construct a user-defined function. We can write our own function to calculate a sum by

```
mysum <- function(obj){
  obj[1] + obj[2] + obj[3] + obj[4]
}
class(mysum)
```



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

```
mysum(x)
```

```
## [1] 13
```

We create an object of class function with a single argument `obj`. The curly brackets `{ }` contain the code that will be evaluated when calling the function, extracting each element of a vector of length four and computing the sum of these elements. Of course, we would prefer to use the function `sum()` insteasd of our own `mysum()`, as it will work for vectors of any length and not just for four elements. The function `sum()` is also a bit faster compared to `mysum()`.

We can also write a function with two arguments, e.g. dividing two numbers.

```
divab <- function(a, b){
  a / b
}
divab(a=2, b=4)
```

```
## [1] 0.5
```

When we know the order of the two arguments, we can omit the name of the arguments at every function call.

```
divab(2, 4)
```

```
## [1] 0.5
```

```
divab(4, 2)
```

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

```
divab(b=4, a=2)
```

```
## [1] 0.5
```

We can also define a default value for an argument, for example, let's modify the function that it will always divide by two if we do not specify a value for the argument `b`.

```
divab <- function(a, b=2){
  a / b
}
divab(a=3)
```



```
## [1] 1.5
```

```
divab(a=3, b=3)
```

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

Scoping rules

What happens in a function, stays in a function.

All the code that is evaluated when calling a function, is evaluated in a new environment. It means, that all objects that are created in the body of the function are not directly available. The function will only return the last called object, or the object that is returned by the function `return()`.

```
# take object x and add one
addone <- function(x){
  x <- x + 1
  return(x)
}
x <- 1
addone(x)
```

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

```
print(x)
```

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

Although, the value `x+1=2` is assigned to `x` within the environment of the function, this does not affect the object `x` in the global environment.

This behaviour can be changed by the assignment operator `<<-` that assigns an object in the global environment.

```
addone <- function(x){
  x <<- x + 1
  return(x)
}
x <- 1
addone(x)
```

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

```
print(x)
```



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


The value 2 is assigned to `x` in the global environment, but within the function, `x` has still a value of 1.

if … then … else …

Relational operators can be used together with a control-flow constructs, that is, evaluating a specific expression if a certain condition is satified; if not, a different expression will be evaluated.

```
x <- 5
if (x < 10){
  "x is smaller than 10"
} else {
  "x is not smaller than 10"
}
```

```
## [1] "x is smaller than 10"
```

When the else command is omitted, it is automatically set to `NULL`. This statement also can be written in a shorter form.

```
ifelse(x < 10, "x is smaller than 10", "x is not smaller than 10")
```

```
## [1] "x is smaller than 10"
```

Loops

A loop allows us to iteratively evaluate functions and store the results.

```
# creating a numeric vector of length 4 to store the results
result <- numeric(length=4)
x <- c(2, 7, 1, 3)
# selecting iteratively each element i from an index vector of length 5
for (i in 1:4){
  # selecting each element of x and adding 1
  # then store the result in the result vector
  result[i] <- x[i] + 1
}
print(result)
```

```
## [1] 3 8 2 4
```

Of course, we can perform this without a loop much simpler and faster by a vector operation.

```
x + 1
```

```
## [1] 3 8 2 4
```


Our self-written function to compute the sum of elements in a vector had the disadvantage that we could only use vectors of length four.


```
x <- c(2, 7, 1, 3)
mysum <- function(obj){
  obj[1] + obj[2] + obj[3] + obj[4]
}
```


We can generalise the function by adding a loop, defining an index from 1 to the length of the vector and incrementally adding one element to the next.

```
mysum <- function(obj){
  objsum <- 0 # initialise sum object
  # looping with i taking values from 1 to length of obj
  for (i in 1:length(obj)){
    objsum <- objsum + obj[i] # adding each element incrementally to objsum
  }
  return(objsum) # returning the sum
}
mysum(x)
```

```
## [1] 13
```

But it is best to avoid loops in R if we can, because loops can be really slow. The `system.time()` function returns the time it took to evaluate the R code.

```
x <- 1:9999999 # creating a vector with lots of elements
system.time(sum(x))
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(mysum(x))
```

```
##    user  system elapsed
##   0.292   0.000   0.292
```

Packages

The functionality of R can be extended by installing add-on packages, which are provided by the R community. These packages contain functions that implement methodology, which is not directly available in the base R program.

The RStudio IDE includes a graphical user interface to install add-on packages from the Comprehensive R Archive Network (CRAN). But any package that is available on CRAN can also be installed by a simple function call `install.packages("package_name")`.

Of course, packages can also be installed from a hard drive as an archived .zip file or from source as a .tar.gz.

There are additional packages available, e.g. package devtools, that allows to install packages from different platforms, like GitHub.