Statistics with R

LV-Nr. 851309 (2 VU - 2.0 ECTS)

Winter term 2019/2020

05 Functions and Programming with/in R

Simona Jokubauskaite and Theresa Scharl
(with course material provided by Michael Melcher)

Institute of Statistics (STAT)
University of Natural Resources and Life Sciences

Contents

Introduction

- R is a functional programming language – each function call performs well defined operations depending only on the arguments of the function.
- A call of a function is performed with the function's name followed by the function's arguments in parentheses, e.g.

```
# function matrix(), input = data vector, no. of columns
matrix(data = c(1, 5, 8, -1, 4, 0), ncol = 2)
```

```
     [,1] [,2]
[1,]    1   -1
[2,]    5    4
[3,]    8    0
```

- Often we don't know (all) arguments (and their names) – we can use another function:

```
# function formalArgs
formalArgs(matrix)
```

```
[1] "data"     "nrow"     "ncol"     "byrow"    "dimnames"
```

- More information on the arguments can be obtained on the function's help page.

Example – `matrix()` Function

- In the `Usage` section we see that there are 5 arguments to the function `matrix()`:

  ```
  matrix(data, nrow, ncol, byrow, dimnames)
  ```

- In the `Arguments` section we are briefly informed about the meaning of the arguments (what type of input R expects from us):
  - `data`: a data vector
  - `nrow`: the desired number of rows of the matrix
  - `ncol`: the desired number of column of the matrix
  - `byrow`: logical (`TRUE` or `FALSE`), filling direction (row- or columnwise)
  - `dimnames`: either `NULL` or a list of length 2 with the row and column names of the matrix.
- Details on the arguments and further information can be found in the `Details` section.

Example – `matrix()` Function

- In the last example only 2 (of 5) arguments were specified (`data` and `ncol`).
- A closer look shows that for all arguments default values are specified (usually, only for some arguments default values exist):

  `matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)`

- A call of the function `matrix()` can therefore be done without any arguments (the defaults are taken) – the result is a matrix with entry `NA` (because `data = NA`) with 1 row (`nrow = 1`) and one column (`ncol = 1`) and without any names of columns/rows (`dimnames = NULL`). For a matrix of size 1×1 we won't see the difference between `byrow = TRUE` and `byrow = FALSE`.

```
# don't forget the parentheses
matrix()
```

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

Example – `matrix()` function

Task: Create a matrix with entries 1 to 4 (filled by row) of dimension 2×2, the rownames `A` and `B` and the column names `X` and `Y`.

If the arguments are named, we can specify them in any order:

```
# all arguments named
matrix(data = 1:4, nrow = 2, ncol = 2, byrow = TRUE,
       dimnames = list(c("A", "B"), c("X", "Y")))

  X Y
A 1 2
B 3 4
```

or

```
# different order of named arguments
matrix(dimnames = list(c("A", "B"), c("X", "Y")), data = 1:4,
       byrow = TRUE, ncol = 2, nrow = 2)

  X Y
A 1 2
B 3 4
```

and receive the same result.

Example – `matrix()` function

If we do not name the arguments, the order is of importance (see the help page):

```
# first "data", then "nrow", "ncol", "byrow", last "dimnames"
matrix(1:4, 2, 2, TRUE, list(c("A", "B"), c("X", "Y")))

  X Y
A 1 2
B 3 4
```

whereas

```
# wrong order, no names
matrix(2, 2, 1:4, list(c("A", "B"), c("X", "Y")), TRUE)

Error in matrix(2, 2, 1:4, list(c("A", "B"), c("X", "Y")), TRUE): ungÃ¼ltiges
'byrow' Argument
```

gives an error (for `byrow` either `TRUE` or `FALSE` is expected, but a list of character vectors is supplied).

Advice: always name the arguments (except in trivial cases, e.g. `mean(x)`) supplied to a function (takes longer, but you make fewer errors and the readability is increased).

Argument Matching

Performing a function call means matching the formal arguments with the actual arguments. Matching means applying the following rules

1. The name in the call matches exactly (!) the name of the formal arguments.
2. The name in the function call matches an initial substring of exactly one formal argument (known as partial matching)

```
# named argument "da" instead of "data" works
matrix(da = 1:4, nrow = 2, ncol = 2, byrow = TRUE,
       dimnames = list(c("A", "B"), c("X", "Y")))
```

```
  X Y
A 1 2
B 3 4
```

As there is a second argument (`dimnames`) starting with `d`, partial matching with simply `d = 1:4` does not work if the argument `dimnames` is not named:

```
# "d" does not work due to ambuguity (data, dimnames)
matrix(d = 1:4, nrow = 2, ncol = 2, byrow = TRUE,
       list(c("A", "B"), c("X", "Y")))
```

```
Error in matrix(d = 1:4, nrow = 2, ncol = 2, byrow = TRUE, list(c("A", :
Argument 1 passt auf mehrere formale Argumente
```

Argument Matching

3. Unnamed actual arguments are matched in order to formal arguments, which were not already matched in one of the first 2 steps.

```
# first matching "byrow" (named argument)
# nc is a valid abbreviation for "ncol"
# remaining 3 arguments are matched in order:
# 1:4 with "data", 2 with "nrow" and the list with "dimnames"
matrix(1:4, 2, nc = 2, byrow = TRUE,
       list(c("A", "B"), c("X", "Y")))
```

```
  X Y
A 1 2
B 3 4
```

So, the following function call is valid (but not really readable):

```
matrix(b = TRUE, 1:4, list(c("A", "B"), c("X", "Y")),
       nc = 2, nro = 2)
```

```
  X Y
A 1 2
B 3 4
```

Contents

Our first R function . . .

When learning a new programming language it is common to design a simple program/function, which prints the text `Hello World` on the screen.

We do this in R now . . .

```
# we create a function
hello_world <- function() {
  print("Hello World")
}
```

We analyse these few lines:

- We decided to give the function the name `hello_world`.
- The word `function` tells R that `hello_world` shall be a function (and not a list, vector, etc.)
- In round brackets we put the formal arguments of our function (eventually with default values). We decided that this function shall have no arguments.
- The body of the function (what the function shall actually do) is enclosed in curly brackets (not necessary if the body consists of a single line). In this case we want that R prints the string `Hello World` on the screen.

Our first R function . . .

- A slightly shorter version is therefore

```
hello_world <- function() print("Hello World")
```

- We call the function `hello_world()`:

```
# we call our function with no arguments
hello_world()
```

```
[1] "Hello World"
```

- If we want to see the code behind the function `hello_world()`, we simply type the function name

```
# function name without brackets/arguments shows code
hello_world
```

```
function() print("Hello World")
```

Our first R function . . .

When having written a function, we want to use it in later R sessions and therefore have to save it. A recommended procedure would be:

- Open a new script window in your editor. In `RStudio`:

  `File - New File - R Script`

- Create one or more functions and save this file as `filename.R`, e.g. in our case `hello_world.R` would make sense (name of the `.R` file not necessarily identical to function name).
- In a future R session either
  - `RStudio`: Choose `File - Open File`, select the file (e.g. `hello_world.R`), press `Ok` and click on `source` in the right upper corner or
  - `Console`:

```
# source an existing function in working directory
source("hello_world.R")
# from now on R knows your function and you can use it
hello_world()
[1] "Hello World"
```

Some Comments . . .

- A typical R function will have the following structure

```
function_name <- function(arg1, arg2, ...) {

  # take the arguments and do something
  #
  # return something, e.g. text on the screen or
  # the result(s) from more or less complex calculations
  # or a plot or ...
}
```

- Choose a meaningful function name, not too complicated, not too simple. The function name shall not start with a number.
- When writing your function, use comments/statements starting with

  #

  (the rest of the line is then ignored by R) to increase the readability of your function.
- In R only one object can be returned. If you want multiple output, collect the results in a list object.
- The value of the last statement R executes is returned by the function, but you should explicitly state the object to be returned using the `return()` function.

Contents

Contents

A Lotto Function

- Assume we want to play Lotto (draw of 6 numbers out of 45 every Wednesday and Sunday in Austria, we ignore the *Zusatzzahl* for now). We want to simulate a Lotto draw in R.
- Our argument(s) to the function `lotto()` shall be
  - `x` ... our 6 numbers we want to put our money on
- We want that our function `lotto()` returns
  - The 6 numbers actually drawn.
  - The number of correct guesses we have $(0 \ldots 6)$

A Lotto Function

We design our function

```r
# lotto function with 1 argument x (= our numbers)
lotto <- function(x) {

  # draw 6 numbers randomly
  numbers <- sample(x = 1:45, size = 6, replace = FALSE)

  # how many correct guesses
  correct <- sum(x %in% numbers)

  # return these 2 objects in one list
  return(list(guess = correct, numbers = sort(numbers)))
}
```

Let's test it

```r
set.seed(123)
lotto(x = c(4, 16, 21, 30, 36, 44))

$guess
[1] 0

$numbers
[1]  3 14 15 31 37 43

# not really lucky
```

A Lotto Function

If you do not want the text output on the screen, you can use the `invisible()` function around the returned object:

```
# Lotto
lotto <- function(x) {

  # x ... draw 6 numbers randomly
  numbers <- sample(x = 1:45, size = 6, replace = FALSE)

  # how many correct guesses
  correct <- sum(x %in% numbers)

  # return list invisibly
  return(invisible(list(guess = correct, numbers = sort(numbers))))
}
```

Now we have no direct text output (does this make sense?):

```
set.seed(345)
lotto(x = c(3, 14, 29, 31, 33, 44))
```

A Lotto Function

We assign the object returned by the `lotto()` function (which is a list) to an object:

```
set.seed(345)
# returned object (a list with 2 elements) stored in variable "lotto_results"
lotto_results <- lotto(x = c(3, 14, 29, 31, 33, 44))
lotto_results

$guess
[1] 2

$numbers
[1] 19 20 21 23 29 31
```

if...else

Sometimes we want conditional expressions: if a certain condition is met, the function shall proceed in a different way than if the condition is not met. We use the `if` or `if...else` construct.

- if:

```
...
if (condition) {
  # do something
}
...
```

- if else:

```
...
if (condition) {
  # do something
} else {
  # do something different
}
...
```

`condition` must be logical (`TRUE`/`FALSE`) or an R expression, which evaluates to a single `TRUE` or `FALSE` (not a logical vector of length > 1).

Example if...else

We add the optional argument print.out (and set the default to FALSE, i.e. no text output) and add an if statement

```
lotto <- function(x, print.out = FALSE) {
  ...
  # text output only if print.out is TRUE
  if (print.out == TRUE) {
    cat("You have", correct, "correct guess(es)", sep = " ")
  }
  ...
}
```

We test the function

```
set.seed(345)
# change default value of print.out
result <- lotto(x = c(3, 14, 29, 31, 33, 44), print.out = TRUE)

You have 2 correct guess(es)
```

Example if...else

Note: instead of

```
...
if (print.out == TRUE)
...
```

we could simply use

```
...
if (print.out)
...
```

Example if...else

To give an example for the else in an if...else statement, we want the function to stop, if the actual argument x is not of length 6 (and proceed as usual, if x has correct length).

```
if (length(x) == 6) {
  # proceed as usual
} else {
  stop("The supplied vector x has the wrong length")
}
```

We check our function by supplying a vector x of length 4:

```
set.seed(123)
# supply wrong vector x
lotto(x = c(3, 22, 33, 39))

Error in lotto(x = c(3, 22, 33, 39)):  The supplied vector x has the wrong
length.
```

and receive an error message.

Example if...else

So finally our function lotto() looks like

```
lotto <- function(x, print.out = FALSE) {
  if (length(x) == 6) {

    # x...our numbers, draw 6 numbers randomly
    numbers <- sample(x = 1:45, size = 6, replace = FALSE)

    # how many correct guesses
    correct <- sum(x %in% numbers)

    # text output only if print.out is TRUE
    if (print.out == TRUE) {
      cat("\n You have", correct, "correct guess(es)\n", sep = " ")
    }

    # return 2 objects as items of a list
    return(invisible(list(guess = correct, numbers = sort(numbers))))
  } else {
    stop("The supplied vector x has the wrong length.")
  }
}
```

Contents

if...else versus ifelse()

- The former function `if...else` requires a single `TRUE` or `FALSE` (or an expression, which evaluates to a single `TRUE` or `FALSE`) in the `condition` part:

```
...
if (condition) {
  # do something
} else {
  # do something different
}
...
```

- The function `ifelse()` allows a logical vector as condition (`ifelse()` is a vectorized version of `if...else`):

```
# x ... integer vector
x <- c(2, 5, 3, 9, 16, 4)
# odd or even integer?
ifelse(x %% 2 == 0, "even", "odd")
[1] "even" "odd"  "odd"  "odd"  "even" "even"
```

ifelse()

The usage of ifelse() is (see also the help page)

```
ifelse(test, yes, no)
```

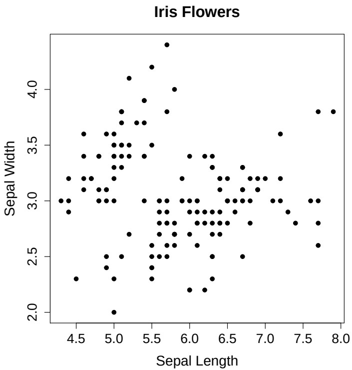
- ifelse() returns an object of the same shape as test (e.g. if test is a 5-element vector, so will be the return value of ifelse).
- test is a logical vector (or an expression evaluating to a logical vector)
  - If the first element in test evaluates to TRUE, the first element in the result will be the first element of yes.
  - If the first element in test evaluates to FALSE, the first element in the result will be the first element of no.
  - ... (for all elements of test)
- yes and no should have the same length as test, but are recycled if too short.

Example `ifelse`

- `iris` data, we want a scatterplot of the variables `Sepal.Width` (y-axis) versus `Sepal.Length` (x-axis):

```
plot(x, y)
```

```r
plot(x = iris$Sepal.Length,
     y = iris$Sepal.Width, pch = 19,
     xlab = "Sepal Length",
     ylab = "Sepal Width",
     main = "Iris Flowers")
```



- Flowers of `Species setosa` shall now be colored in blue.

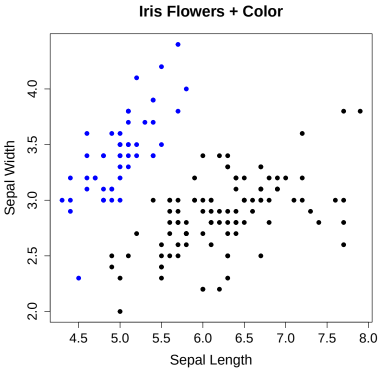
Example ifelse

We create a color vector first with ifelse():

```
# ifelse: if setosa -> blue, otherwise -> black
col_vector <- ifelse(iris$Species == "setosa", "blue", "black")
```

and use this vector as a col argument in plot():

```
# plot, color given by our col_vector
plot(x = iris$Sepal.Length,
     y = iris$Sepal.Width, pch = 19,
     col = col_vector,
     main = "Iris Flowers + Color",
     xlab = "Sepal Length",
     ylab = "Sepal Width")
```

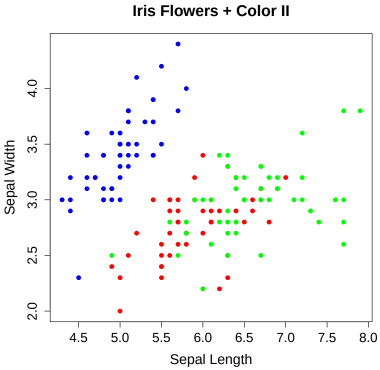


Example ifelse

. . . a little more complicated – 3 different colors: setosa in blue, versicolor in red and virginica in green. We use a nested ifelse():

```
# nested ifelse
col_vector2 <- ifelse(iris$Species == "setosa", "blue",
                      ifelse(iris$Species == "versicolor", "red", "green"))
```

```
# plot, color given by our col_vector
plot(x = iris$Sepal.Length,
     y = iris$Sepal.Width, pch = 19,
     col = col_vector2,
     main = "Iris Flowers + Color II",
     xlab = "Sepal Length",
     ylab = "Sepal Width")
```



Contents

`for` loops

- Sometimes we want to perform certain operations multiple times. If we know in advance how often we want R to repeat certain expressions, we can use a `for` loop.
- Now we want to play our Lotto game for n times. We could
  - write a completely new function with an additional argument `n` (the number of times we want to play the game) or
  - write a (shorter) `wrapper`, which uses our *old function* `lotto()`
- In any case, we want to collect all single results (i.e. the number of correct guesses and the numbers actually drawn in each game).

Example `for` loops

Our function for n lotto games:

```
mult_lotto <- function(n, x) {

  # prepare (initialize) objects for results
  correct_guesses <- numeric(length = n)
  drawn_numbers <- matrix(NA, nrow = n, ncol = 6)

  # repeat for n times
  for (i in 1:n) {

    # one Lotto game
    one_draw <- lotto(x = x, print.out = FALSE)

    # save single results in position/row i
    correct_guesses[i] <- one_draw$guess
    drawn_numbers[i, ] <- one_draw$numbers
  }

  # return results as list
  return(invisible(list(guess = correct_guesses,
                        numbers = drawn_numbers)))
}
```

Example `for` loops

We play $n = 1000$ times (which is playing ≈ 10 years):

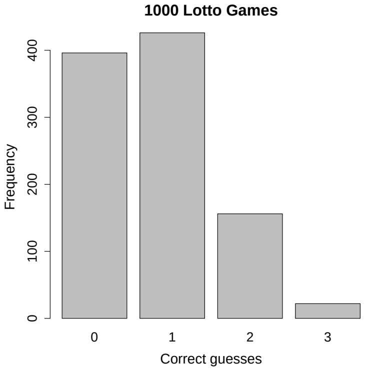
```
set.seed(299)
results <- mult_lotto(n = 1000, x = c(8, 12, 23, 29, 38, 42))
```

The number of correct guesses is stored in the list element `guess`, which is a vector of length `n`.

```
# table
table(results$guess)


  0   1   2   3
396 426 156  22
```

```
# barplot
barplot(table(results$guess),
        xlab = "Correct guesses",
        ylab = "Frequency",
        main = "1000 Lotto Games")
```



for loops

The structure of a for loop is simple:

```
...
for (name in vector) {
  # do something
}
...
```

- The expression(s) in curly braces is/are repeated a fixed number of times (corresponding to the length of vector):
  - in the first run, the variable name is set equal to the first element in vector
  - in the second run, the variable name is set equal to the second element in vector
  - ...
  - in the last run, the variable name is set equal to the last element in vector
- The current value of name (changes in every iteration) can e.g. be used for saving the results in certain positions of a vector/matrix/list/data frame (see previous example, where results of iteration i are stored in position i of the vector correct_guesses or in row i of the matrix drawn_numbers).

for loops & next

- Sometimes we want to skip certain iterations in a `for` loop, which we achieve with `next`.

```
# print only the odd values of vector in a loop
for (i in 1:10) {

  # if i is even -> skip
  if (i %% 2 == 0) next

  # if i is odd -> print the value
  else cat("The current value of i is", i, "\n", sep = " ")
}
```

```
The current value of i is 1
The current value of i is 3
The current value of i is 5
The current value of i is 7
The current value of i is 9
```

- In the `if...else` statement we see that we do not have to use curly braces, if only one statement (`next` in the `if` part or `cat(...)` in the `else` part) follows.
- `cat()` concatenates its arguments to a single string, separated by `sep`.
- `\n` is called an escape sequence (starts a new line); `\t` prints a tab.

for loops & break

If we want to completely break out of a for loop (or while or repeat loop - see later), we use the break statement.

```
# print elements of vector, but to avoid long output, print max. 5 elements

# create vector of a-prori unknown length
set.seed(123)
vec <- sort(unique(sample(x = 1:100, size = 100, replace = TRUE)))

# set counter
ct <- 0
for (i in vec) {

  # increase counter by 1
  ct <- ct + 1
  cat("The current value of i is", i, "\n", sep = " ")

  if (ct == 5) break
}

The current value of i is 4
The current value of i is 6
The current value of i is 7
The current value of i is 9
The current value of i is 12
```

Misuse of `for` loops

Whenever possible, avoid `for` loops, which are rather slow. Imagine we have a large (e.g. 3000×3000) matrix and are interested in the columnwise means. We measure the time needed by R for the calculation with `system.time()`

```
# create very large matrix
set.seed(123)
M <- matrix(rnorm(9000000), ncol = 3000)

# possibility 1: a for loop
# vector results growing larger in every iteration
results <- c()
system.time(
  for (i in 1:ncol(M)) {
    results <- c(results, mean(M[, i]))
  }
)

   user  system elapsed
   0.18    0.01    0.20
```

Misuse of `for` loops

```
# possibility 2: another for loop
# results vector initialized
results <- numeric(length = ncol(M))
system.time(
  for (i in 1:ncol(M)) {
    results[i] <- mean(M[, i])
  }
)

   user  system elapsed
   0.18    0.04    0.20

# possibility 3: apply (not faster, but shorter and much higher readability)
system.time(apply(M, 2, mean))

   user  system elapsed
   0.31    0.00    0.33
```

Misuse of `for` loops

```
# possibility 4: colMeans
system.time(colMeans(M))

   user  system elapsed
   0.02    0.00    0.01
```

Conclusions:

- Whenever possible, use a built-in function, such as `colMeans` in the previous example.
- For a better readability (and sometimes speed) of your code, use the `apply` (or `lapply` or `tapply`) function family.
- Use vectorized functions
- Initialize your result vector (or matrix/array) instead of increasing its size in every iteration. Reason: not speed, but memory.

Contents

while loops

- The for loop can be used, if the number of required iterations is known in advance.
- Sometimes we want to repeat as long as a certain condition holds, for which we can use the while loop

```
...
while (condition) {
  # do something
  # value of condition might/should change in this block
}
...
```

- It might happen, that the while loop is not executed at all, if the condition evaluates to FALSE at the beginning.
- On the other hand: if condition is TRUE at the beginning and does not change during execution of the while body, the *do something* part is repeated ∞ often.

`while` loops

- Example Fibonacci Numbers – a sequence of integers $(a_n)_{n \in \mathbb{N}}$ defined by

$$a_1 := 1 \quad a_2 := 1 \;\text{ and }\; a_n := a_{n-2} + a_{n-1}$$

So the value of an element of this sequence is the sum of the two preceeding elements. The first 7 elements are for example

$$a_1 = 1 \quad a_2 = 1 \quad a_3 = 2 \quad a_4 = 3 \quad a_5 = 5 \quad a_6 = 8 \quad a_7 = 13$$

while loops

We want to design a function `fib(M)`, which returns all Fibonacci numbers $\leq M$. We want to create these numbers with a loop, but we do not know, how many repetitions are required.

```
fib <- function(M) {

  # first two FN in vector
  fn <- c(1, 1)

  # condition: last value <= M
  while (fn[length(fn)] <= M) {

    # sum of last 2 FN
    sum_2 <- fn[length(fn) - 1] + fn[length(fn)]

    # append
    fn <- c(fn, sum_2)
  }

  # last value might exceed M, remove it
  if (fn[length(fn)] > M) {
    fn <- fn[-length(fn)]
  }

  return(fn)
}
```

while loops

- A short test – all Fibonacci numbers ≤ 300:

```
fib(M = 300)
 [1]   1   1   2   3   5   8  13  21  34  55  89 144 233
```

repeat loops

We can solve the previous Fibonacci problem using a `repeat` loop:

```
...
repeat {
  # do something
}
...
```

If no explicit `break` statement is given in the `repeat` loop, the statements in curly braces are repeated infinitely often (which is surely not what we want).

repeat loops

The Fibonacci example with a repeat loop

```
fib2 <- function(M) {

  # first 2 FN
  fn <- c(1, 1)

  repeat {

    # sum of last 2 FN
    sum_2 <- fn[length(fn) - 1] + fn[length(fn)]

    # break if sum_2 > n
    if (sum_2 > M) break

    # otherwise append to existing vector of FN
    fn <- c(fn, sum_2)
  }

  return(fn)
}
```

We test fib2:

```
fib2(M = 300)

 [1]   1   1   2   3   5   8  13  21  34  55  89 144 233
```

Contents

Operators

- We are used to specify a function call in the following way:

```
function_name(arg_1, arg_2, ...)
```

```
# Example
set.seed(123)
mean(x = rnorm(10, mean = 0, sd = 2))
```

```
[1] 0.1492513
```

- It seems that for (binary) operators (e.g. $+$, $-$, $\%\%$, $\ldots$) the syntax is different:

```
arg_1 operator_name arg_2
```

```
# Examples
4 + 2
```

```
[1] 6
```

```
# remainder in division by 2
11 %% 2
```

```
[1] 1
```

```
# set operation
c(1, 4, 7) %in% c(4, 19, 34)
```

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

Operators

- This is only partly true, as we could specify e.g. an addition also in the following (more complicated) way:

```
# Addition
"+"(4, 2)
[1] 6
# Remainder
"%%"(11, 2)
[1] 1
# Set Operation
"%in%"(c(1, 4, 7), c(4, 19, 34))
[1] FALSE  TRUE FALSE
```

- We see that $+$, $-$, $\ldots$ are functions in R like any other.

Operators

We will define our own binary operator for the Δ-operation (symmetric set difference). Already existing set operations in R:

- The union (*Mengenvereinigung*) of two sets A and B:

$$A \cup B := \{x : x \in A \ \lor \ x \in B\}$$

```
# define sets A and B
A <- c(1, 4, 6, 8)
B <- c(4, 8, 13, 20)
union(A, B)
```

```
[1]  1  4  6  8 13 20
```

- The intersect (*Durchschnitt*) of two sets A and B:

$$A \cap B := \{x : x \in A \ \land \ x \in B\}$$

```
intersect(A, B)
```

```
[1] 4 8
```

Operators

- We can check two sets A and B for equality with `setequal()`:

```r
# 2 equal sets A and B
A <- c(1, 2, 5)
B <- c(1, 5, 2)
setequal(A, B)
```

```
[1] TRUE
```

```r
# 2 unequal sets A and B
A <- c(1, 2, 5)
B <- c(2, 7, 11, 19)
setequal(A, B)
```

```
[1] FALSE
```

- We can check, if an element `x` is contained in a set with the function

  `is.element(el, set)`

  The argument `el` can be a single element or a set. The result is a logical vector of the same length as `el`.

```r
is.element(el = c(2, 3), set = c(1, 3, 6))
```

```
[1] FALSE  TRUE
```

Operators

- The operator %in% is identical to the function `is.element()`.

```
# the operator %in%
c(1, 4, 7) %in% c(2, 4, 14)
[1] FALSE  TRUE FALSE
```

- The result of

  `A %in% B`

  is again a logical vector x with the same length as A. The i^{th} element of x is `TRUE`, if the i^{th} element of A is contained in B.

Operators

- The set difference (*Mengendifferenz*) of two sets A and B:

$$A \setminus B := \{x : x \in A \ \wedge \ x \notin B\}$$

```
# define sets
A <- c(1, 4, 6, 8)
B <- c(4, 8, 13, 20)
setdiff(A, B)
[1] 1 6
```

- The symmetric difference (*symmetrische Mengendifferenz*) of two sets A and B:

$$A \,\Delta\, B := \{x : x \text{ in exactly one of A and B}\}$$

or in other words:

$$A \,\Delta\, B := (A \setminus B) \cup (B \setminus A)$$

Operators

- We use
$$A \,\Delta\, B := (A \setminus B) \cup (B \setminus A)$$
for our R-operator %delta%:

```
"%delta%" <- function(A, B) {

  # A\B = A without B
  A_wo_B <- setdiff(A, B)

  # B\A = B without A
  B_wo_A <- setdiff(B, A)

  # symmetric difference
  A_delta_B <- union(A_wo_B, B_wo_A)

  # return symm. diff.
  return(A_delta_B)
}
```

- We test our function (operator):

```
A <- c(1, 2, 3)
B <- c(1, 3, 7, 8, 9)

# symmetric difference of A and B
A %delta% B

[1] 2 7 8 9
```

The ... argument

- Some functions in R (e.g. the `apply()` function) contain a ... (*dots* or *ellipsis*) argument:

```
apply(X, MARGIN, FUN, ...)
```

- From the help page we obtain the following information
  - `X` ... an array (but most often a matrix)
  - `MARGIN` ... the dimension, to which the function `FUN` shall be applied. 1 stands for rows, 2 for columns.
  - `FUN` ... the function to be applied on `X`.
  - `...` ... (as the R help page tells us) optional arguments passed to `FUN`
- The ... argument gives us the possibility of a finer control of the function `FUN`.

Example: The ... argument

- Columnwise means of a matrix $\mathbf{M}$:

```
# 3 x 3 matrix M with random numbers
set.seed(222)
M <- matrix(data = rnorm(9), ncol = 3)
apply(X = M, MARGIN = 2, FUN = mean)
```

```
[1]  0.9556287 -0.1476578  0.2577181
```

- We can pass additional arguments to the function `mean()` via the ... argument, e.g. if we want a trimmed mean (argument `trim`) or if we want to remove missing values (argument `na.rm`).

```
# introduce NAs in 1st and 2nd column of M
M[1, 1] <- M[1, 2] <- NA
apply(X = M, MARGIN = 2, FUN = mean)
```

```
[1]        NA        NA 0.2577181
```

```
# pass na.rm = TRUE to mean() via ...
apply(X = M, MARGIN = 2, FUN = mean, na.rm = TRUE)
```

```
[1]  0.68956444 -0.03137983  0.25771813
```

Argument matching for ... functions

For functions with a ... argument, argument matching is done in the following three-stage process:

1. Exact matching: happens in the same way as for functions without ... argument.
2. Partial matching: partial matching does not work for arguments after the ... argument.
3. Matching in order: unnamed actual arguments are matched in order. All actual arguments remaining after that are matched with the ... argument.

Contents

Environments

- When working with functions, it is important to know, when which objects (e.g. variables) exist.
- We are used to create (and eventually overwrite/remove) objects in the console, e.g.

```
# create object x
(x <- 12)
```

```
[1] 12
```

```
# ...
# do something else
# ...
# later redefine x
x <- 15
x
```

```
[1] 15
```

- These objects are in the so-called workspace of R. Calling `ls()` lists them:

```
ls()
```

```
 [1] "%delta%"      "A"            "B"            "col_vector"
 [5] "col_vector2"  "ct"           "fib"          "fib2"
 [9] "hello_world"  "i"            "lotto"        "lotto_results"
[13] "M"            "mult_lotto"   "par1_off"     "par1_on"
[17] "par2_off"     "par2_on"      "result"       "results"
[21] "vec"          "x"
```

Environments

If we define objects within a function (e.g. a vector `m`), we expect, that objects with the same name sitting in our workspace are not overwritten, so that – after using the function – they have the same value as before

```
# create vector m in workspace
m <- 1:5

# define a function (Euklidean length)
euclid <- function(x) {
  m <- x^2
  cat("The current value of m is:", m, "\n")
  return((sum(m))^0.5)
}

# use function
euclid(x = 5:8)

The current value of m is: 25 36 49 64
[1] 13.19091

# value of m
m

[1] 1 2 3 4 5
```

Environments

- Functions are evaluated in own environments – objects created there are only temporarily available (and are not saved in the workspace). R's name for the workspace is `global environment`:

```
# create function which returns input, increased by 1
add1 <- function(x) {
  d <- x + 1
  cat("The current value of d is: ", d, "\n")
  return(d)
}

# call function
add1(5)

The current value of d is:  6
[1] 6

# show value of variable d -> not existing any more
d

Error in eval(expr, envir, enclos):  Objekt 'd' nicht gefunden
```

- After evaluation, the environment of a function (and all objects in it) is deleted. Exception: the objects returned by the function.

Environments

- If a function needs access to an object, it is searched along a so called search path. First, the environment of the function itself is searched. If the function was called within another function, then the environment of this function is searched, . . . , finally the workspace (global environment) and some R-packages.
- The search path can be shown with `search()`:

```
search()
```

```
 [1] ".GlobalEnv"        "package:knitr"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

Environments

- Imagine you have a function, which simply adds the two arguments and returns the sum:

```
# create function
add_args <- function(a, b) {
  return(a+b)
}
# test
add_args(3, 7)
```

```
[1] 10
```

- You make an error in programming the function `add_args()` and do the following:

```
# create erroneous function
add_args <- function(a, b) {
  # d instead of b
  return(a+d)
}
```

Environments

- If you are lucky, a test of your function will give this result:

```
# test
add_args(3, 7)
Error in add_args(3, 7):  Objekt 'd' nicht gefunden
```

- If you are not so lucky, then you might have defined a variable with the name `d` in your workspace (perhaps some hours ago).

```
d <- 17
# ...
# some hours later
# ...
add_args(3, 7)
[1] 20
```

- The global environment is in the next position to search for the variable – as it does not exist in the function's environment, this value is taken by R.
- For a simple function like this, you will find the error soon, but what if your function is more complicated?

Contents

S3 Classes

We already know the `summary()` function – not really astonishing it gives us a summary of the object we supply as an argument.

- Summary of a `numeric` vector:

```
# iris data - variable Sepal.Width
summary(object = iris$Sepal.Width)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  2.000   2.800   3.000   3.057   3.300   4.400 
```

  We obtain some descriptive statistics (median, mean, . . .)

- Summary of a `factor` variable:

```
# iris data
summary(object = iris$Species)

    setosa versicolor  virginica 
        50         50         50 
```

  In this case we receive a table telling us how often each factor levels occurs in the data set (but no mean, median, which would not make sense here).

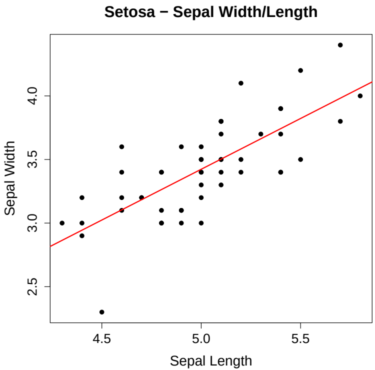
S3 Classes

Summary of a linear model: We plot the variable `Sepal.Width` versus `Sepal.Length` for the observations of species `setosa`:

```
# scatterplot
plot(x = iris$Sepal.Length[iris$Species == "setosa"],
     y = iris$Sepal.Width[iris$Species == "setosa"],
     xlab = "Sepal Length", ylab = "Sepal Width",
     main = "Setosa - Sepal Width/Length",
     pch = 19)

# linear model
lin_mod <- lm(Sepal.Width ~ Sepal.Length, data = iris,
              subset = (Species == "setosa"))

# plot regression line
abline(lin_mod, col = "red", lwd = 2)
```



There might be an approximate linear relationship between these two variables (red line). The linear model (see later) is calculated with the R-function `lm()`.

S3 Classes

If we apply the `summary()` function to the linear model object, we get

```
# linear model
lin_mod <- lm(Sepal.Width ~ Sepal.Length, data = iris,
              subset = Species == "setosa")
# model summary
summary(lin_mod)


Call:
lm(formula = Sepal.Width ~ Sepal.Length, data = iris, subset = Species ==
    "setosa")

Residuals:
     Min       1Q   Median       3Q      Max
-0.72394 -0.18273 -0.00306  0.15738  0.51709

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   -0.5694     0.5217  -1.091    0.281
Sepal.Length   0.7985     0.1040   7.681 6.71e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2565 on 48 degrees of freedom
Multiple R-squared:  0.5514,Adjusted R-squared:  0.542
F-statistic: 58.99 on 1 and 48 DF,  p-value: 6.71e-10
```

S3 Classes

- The reason for this *strange (?) behaviour* (the same function does different things) of `summary()` are the different classes of the (first) arguments to `summary()`:

```
# class of our data vector
class(iris$Sepal.Width)
```

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

```
# class of variable "Species"
class(iris$Species)
```

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

```
# class of Linear Model
class(lin_mod)
```

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

- A class is just an attribute of our object:

```
attributes(lin_mod)
```

```
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

$class
[1] "lm"
```

S3 Classes

- Depending on the class of the object, a suitable so-called method of the function `summary()` is applied. We can see the available methods:

```
methods(summary)
```

```
 [1] summary.aov                    summary.aovlist*               summary.aspell*
 [4] summary.check_packages_in_dir* summary.connection             summary.data.frame
 [7] summary.Date                   summary.default                summary.ecdf*
[10] summary.factor                 summary.glm                    summary.infl*
[13] summary.lm                     summary.loess*                 summary.manova
[16] summary.matrix                 summary.mlm*                   summary.nls*
[19] summary.packageStatus*         summary.POSIXct                summary.POSIXlt
[22] summary.ppr*                   summary.prcomp*                summary.princomp*
[25] summary.proc_time              summary.srcfile                summary.srcref
[28] summary.stepfun                summary.stl*                   summary.table
[31] summary.tukeysmooth*           summary.warnings
see '?methods' for accessing help and source code
```

- If no suitable method is found, the method `summary.default()` is applied. We call the function `summary()` a generic function.

S3 Classes

The object `lin_mod` is essentially a `list` (of length 12) with the items

- `coefficients` ... the regression coefficients
- `residuals` ... the regression residuals
- ...

```
# structure of lm class object
str(lin_mod, give.attr = FALSE, give.head = TRUE, max.level = 1)

List of 12
 $ coefficients : Named num [1:2] -0.569 0.799
 $ residuals    : Named num [1:50] -0.00306 -0.34336 0.01635 -0.0038 0.17679 ...
 $ effects      : Named num [1:50] -24.2396 1.9703 0.0589 0.052 0.1795 ...
 $ rank         : int 2
 $ fitted.values: Named num [1:50] 3.5 3.34 3.18 3.1 3.42 ...
 $ assign       : int [1:2] 0 1
 $ qr           :List of 5
 $ df.residual  : int 48
 $ xlevels      : Named list()
 $ call         : language lm(formula = Sepal.Width ~ Sepal.Length, data = iris, su
 $ terms        :Classes 'terms', 'formula'  language Sepal.Width ~ Sepal.Length
 $ model        :'data.frame': 50 obs. of  2 variables:
```

S3 Classes

If we delete the class attribute of the object `lin_mod`, R sees `lin_mod` no longer as an object of class `lm`, but as an ordinary list and the `summary()` function acts in a different way:

```
# class lin_mod
class(lin_mod)

[1] "lm"

# delete attribute
attributes(lin_mod)$class <- NULL

# class now
class(lin_mod)

[1] "list"
```

S3 Classes

```
# summary on this list
summary(lin_mod)

              Length Class      Mode
coefficients   2     -none-     numeric
residuals     50     -none-     numeric
effects       50     -none-     numeric
rank           1     -none-     numeric
fitted.values 50     -none-     numeric
assign         2     -none-     numeric
qr             5     qr         list
df.residual    1     -none-     numeric
xlevels        0     -none-     list
call           4     -none-     call
terms          3     terms      call
model          2     data.frame list
```

Contents

Example S4 Classes

- Imagine you want to build a student database for a university. Each "item" of this data base is a student, for whom you want to store the following data:
  - first name (character)
  - surname (character)
  - student-ID (*Matrikelnummer*, character)
  - year of birth (numeric)
  - studies (character)
  - ...
- We could do this in a list:

```
# test item for database with 4 slots
test_item <- list(first_name = "Michael", surname = "Melcher",
                  ID = "9425107", year_of_birth = 1976)
class(test_item) <- "student"
```

- We could then create print, summary etc. methods, i.e. functions `print.student()`, `summary.student()`, etc.) for our new class `student`.

Example S4 Classes

- The student ID should be of type character (otherwise, a preceeding "0" would be deleted):

```
# test item fou database with 4 slots
test_item2 <- list(first_name = "Max", surname = "Mustermann",
                   ID = 0827119, year_of_birth = 1989)
test_item2
```

```
$first_name
[1] "Max"

$surname
[1] "Mustermann"

$ID
[1] 827119

$year_of_birth
[1] 1989
```

- The important thing is, that (of course), R did not complain about this wrong entry of the student ID as a numeric vector (instead of e.g. a character).

S4 classes

- The goal of `S4` classes is to prevent such accidents (wrong variable type, misspelled list items ...).
- We create/define a new class with the `setClass()` function:

```
# student example
setClass(Class = "student",
         slots = c(first_name = "character",
                   surname = "character",
                   ID = "character",
                   year_of_birth = "numeric"))
```

- The argument `Class` is a character string (name of the class to be created), `slots` is a named list or character vector containing the names and types of the slots.

S4 classes

- A new instance of this class is generated with `new()`:

```
# create a new instance
test_person <- new(Class = "student", first_name = "Max",
                   surname = "Mustermann", ID = "0827119",
                   year_of_birth = 1989)
test_person
```

```
An object of class "student"
Slot "first_name":
[1] "Max"

Slot "surname":
[1] "Mustermann"

Slot "ID":
[1] "0827119"

Slot "year_of_birth":
[1] 1989
```

S4 classes

- Contrary to list items, which can be accessed with the $-sign, the slots are referenced via the @-symbol or the `slot()`-function:

```
# use @ instead of $
test_person@ID
```

```
[1] "0827119"
```

```
# access via slot function
slot(test_person, "surname")
```

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

```
# changes in the usual way
test_person@year_of_birth <- 1990
```

- S4 classes provide more safety:

```
# slot ID is a character, but we enter a numeric value
test_person@ID <- 1990
```

```
Error in (function (cl, name, valueClass) : assignment of an object of
class "numeric" is not valid for @'ID' in an object of class "student";
is(value, "character") is not TRUE
```

S4 classes

- Assignments in non-existing slots are not possible

```
# non-existing slot "IT" instead of "ID"
test_person@IT <- "0874229"
Error in (function (cl, name, valueClass) :  'IT' is not a slot in class
"student"
```

- Methods for `S4` classes can be implemented with the `setMethod()` function.
- Example: the `S4` analogon of the `S3` `print` function is `show()`:

```
# not a very nice output
show(test_person)
An object of class "student"
Slot "first_name":
[1] "Max"

Slot "surname":
[1] "Mustermann"

Slot "ID":
[1] "0827119"

Slot "year_of_birth":
[1] 1990
```

S4 classes

- We create a show method for our object of class student:

```
setMethod(f = "show", signature = "student",
          definition = function(object) {
            cat("Mr./Mrs. ", object@first_name, " ", object@surname,
                " has student ID ", object@ID, " and was born in ",
                object@year_of_birth, ".\n", sep = "")
            }
          )
```

- We test our method:

```
# alternative: simply type object to show
show(test_person)
```

```
Mr./Mrs. Max Mustermann has student ID 0827119 and was born in 1990.
```

... a much better/nicer output than before for our class student.

Contents

Literature on R-Programming

- Uwe Ligges: *Programmieren mit R*. Springer, 2008.
- W. John Braun & Duncan J. Murdoch: *A first Course in Statistical Programming with R*. Cambridge University Press, 2007.
- Norman Matloff: The Art of R Programming. no starch press, 2011.
- Owen Jones, Robert Maillardet & Andrew Robinson: *Introduction to Scientific Programming and Simulation using R*. CRC Press, 2009.
- W. N. Venables & B. D. Ripley: *Modern Applied Statistics with S*. Springer, 2002.
- Patrick Burns: *The R Inferno*, 2011, available online.
- John M. Chambers: *Software for Data Analysis*. Springer, 2008.