Day_2_session_02

Module VII:

Conditional Execution/statements in R
- If statements - If-else statements - Multiple if statements - switch statement

In R programming like that with other languages, there are several cases where you might wish for conditionally execute any code. There are three 'different' conditional statements we will go trough in this module.

If statements: Single expression with corresponding instruction.

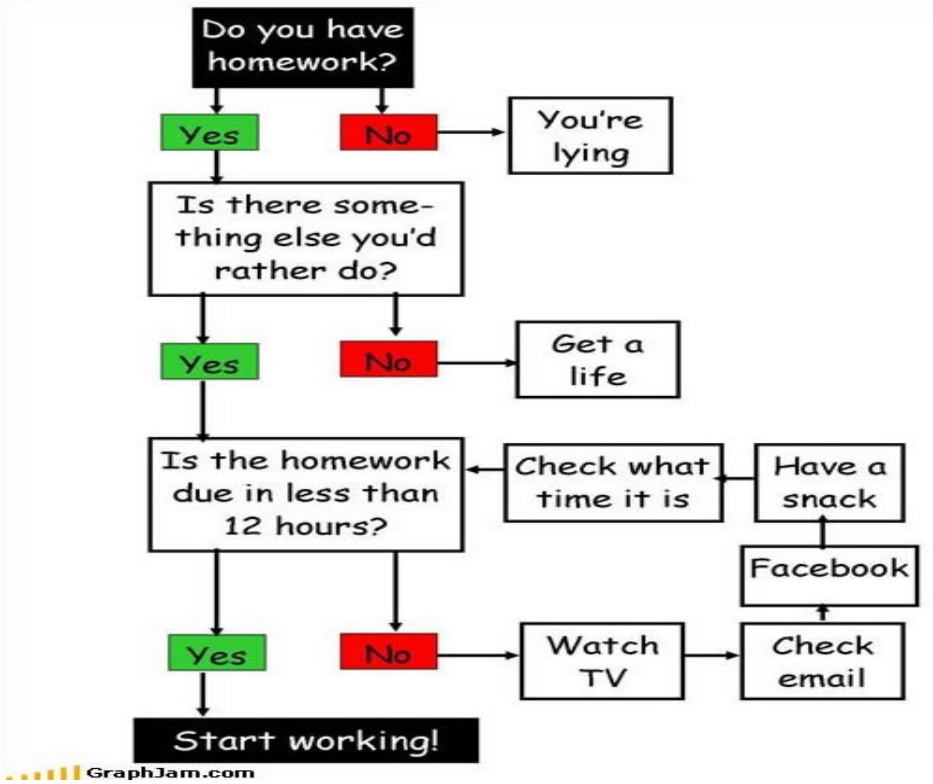
- **If** you are hungry: Eat something.
- **If** the alarm clock rings: Get up.

If-else statements: Single expression with corresponding and alternative "else" instruction.

- **If** the traffic light is green: Walk. **Else:** Stop.
- **If** the coffee cup is empty: Refill. **Else:** Drink.

Multiple if statatements: Multiple expressions with corresponding instructions.

- **If** warm and dry outside: Wear a T-shirt. **Else if** warm and rainy: T-shirt and a jacket. **Else if** cold and dry: Sweater. **Else:** Stay home.



If statements

Basic usage: if (<condition>) { <action> }.

- The <condition> has to be a single logical TRUE or FALSE.
- If <condition> is evaluated to TRUE, the <action> is executed.

```
x <- 8
if (x < 10) { cat("x is smaller than 10\n") }

x <- 12
if (x < 10) { cat("x is smaller than 10\n") }
```

Version 1: In separate lines.

```
if (x < 10) {
 cat("x is smaller than 10\n")
}
```

Version 2: In a single line.

```
if (x < 10) { cat("x is smaller than 10\n") }
```

Version 3: In a single line without brackets.

```
if (x < 10) cat("x is smaller than 10\n")
```

If-else statements

Basic usage:

- Structure: if (<condition>) { <action 1> } else { <action 2> }.
- If <condition> is evaluated to TRUE, <action 1> is executed. Else <action 2> is executed.

Let us take the same example as above where we check if a certain number is smaller than 10.

```
x <- 8
if (x < 10) {
    print("x is smaller than 10")
} else {
    print("x is larger or equal to 10")
}
```

Version 1: The preferred one.

```
if (x < 10) {
   print("x is smaller than 10")
} else {
   print("x is larger or equal to 10")
}
## [1] "x is smaller than 10"
```

Version 2: One-liner.

```
if (x < 10) { print("x is smaller than 10") } else { print("x is larger or equal to 10") }
## [1] "x is smaller than 10"
```

Version 3: Without brackets.

```
if (x < 10) print("x is smaller than 10") else print("x is larger or equal to 10")
## [1] "x is smaller than 10"
```

Nested conditions

If-else statements can also be nested. Nested means that one of the actions itself contains another if-else statement. Important: the two if-else statements are independent.

```
x <- 10
# 'Outer' if-else statement.
if (x < 10) {
    print("x is smaller than 10")
} else {
    # 'Inner' if-else statement.
    if (x > 10) {
        print("x is larger than 10")
    } else {
        print("x is exactly 10")
    }
}
```

- Outer if-else statement: Is x < 10? FALSE: execute action in the else-block of the outer if-else statement.
- Inner if-else statement: Is x > 10? FALSE: execute action in the else-block of the inner if-else statement, print "x is exactly 10").

Multiple if-else statements

Instead of nested (independent) if-conditions we can once again extend the concept by adding multiple if-else conditions in one statement. The difference to nested conditions is that this is one single large statement, not several smaller independent ones.

Basic usage:

- Structure: if (<condition 1>) { <action 1> } else if (<condition 2>) { <action 2> } else { <action 3> }.
- If <condition 1> evaluates to TRUE, <action 1> is executed.
- Else <condition 2> is evaluated. If TRUE, <action 2> is executed.
- Else, <action 3> is executed (if both, <condition 1> and <condition 2>, evaluate to FALSE).

Required parts:

- Not limited to only 2 conditions.
- Always needs one (and no more than one) if.
- Can have 1 or more else ifs.
- Can have no or 1 else-block (optional).

```
x <- 10
if (x < 10) {
    print("x is smaller than 10")
} else if (x > 10) {
    print("x is larger than 10")
} else {
    print("x is exactly 10")
}
```

```
x <- 10
if (x < 10) {
    print("x is smaller than 10")
} else if (x > 10) {
    print("x is larger than 10")
} else if (x == 10) {
    print("x is exactly 10")
}
```

Return values

If-statements are not functions, but they still have a return value. By default, this return is not visible, but we can make use of it.

```
x <- 200
desc <- if (x < 10) {
    "x is smaller than 10"
} else if (x > 10) {
    "x is larger than 10"
```

```
} else if (x == 10) {
    "x is exactly 10"
}
# Print return value
print(desc)
```

Vectorized if

There is a special function which allows perform an if-else statement element by element for each element of a vector (or matrix).

Function: ifelse() for conditional element selection.

- **Arguments:** ifelse(test, yes, no), where all arguments can be vectors of the same length (recycled if necessary). Works with matrices as well.
- **Return:** Vector which contains yes elements if test is TRUE, else no elements.
- Note: All elements of yes and no are always evaluated.

```
(x <- 1:6)
## [1] 1 2 3 4 5 6
x %% 2 == 0      # Rest 0?
```

```
#         test       yes      no
ifelse(x %% 2 == 0, "even", "odd")
```

```
#          test     yes  no
ifelse(x %% 2 == 0,  x,  -x)
```

Matrices: ifelse() also works with matrices. In case the input for the condition/test is a matrix, a matrix of the same size will be returned. The vectorized if works on the underlying vector, but adds the matrix attributes again at the end.

```
(x <- matrix(1:12, nrow = 3,
             dimnames = list(paste("Row", 1:3), paste("Col", LETTERS[1:4]))))
```

```
ifelse(x %% 2 == 0, x, -x)
```

Sanity checks

A typical application for single if-statements are input checks of a function or before proceeding to the computations and are often used in combination with stop() or warning().

- stop(): Will show an error warning and immediately stop execution.
- warning(): Issues a warning, but the program will still be executed.

When used inside functions, this is called a sanity check. Sanity checks should be at the very beginning of the instructions of a function and check if the arguments are sane, or if the function should throw an error because the inputs are wrong.

```
# Custom square root function
custom_sqrt <- function(x) {
    # Sanity check
    if (!is.numeric(x))   stop("Argument 'x' must be numeric!")
    if (!length(x) == 1L) stop("Argument 'x' must be of length 1!")
    if (x < 0)            stop("Argument 'x' must be positive (>= 0)!")

    # If not yet run into an error, do the calculation and return result
    res <- sqrt(x)
    return(res)
}
```

```
# Working examples
custom_sqrt(9.0)
```

```
custom_sqrt(25L)                # Remember: integers are also numeric
```

```
custom_sqrt("17")               # Error: not numeric
```

```
custom_sqrt(vector("numeric", 0)) # Error: length is zero (not equal to 1)
```

```
custom_sqrt(c(1, 2, 3))         # Error: length > 1
```

```
custom_sqrt(-3)                 # Error: no positive numeric value
```

Instead of using three different checks in custom_sqrt() and three different error messages we could also combine everything in one single check using a logical | (or ||). Let us re-declare the function:

```
# Custom square root function
custom_sqrt <- function(x) {
    if (!is.numeric(x) || !length(x) == 1 || x < 0) stop("wrong input")
    return(sqrt(x))
}
```

Switch

Switch case statements are a substitute for long if statements that compare a variable to several integral values. Switch case in R is a multiway branch statement. It allows a variable to be tested for equality against a list of values.

```
# Following is a simple R program
# to demonstrate syntax of switch.
val <- switch(
   4,
   "Geeks1",
   "Geeks2",
   "Geeks3",
   "Geeks4",
   "Geeks5",
   "Geeks6"
)
print(val)
```

```
# Following is val1 simple R program
# to demonstrate syntax of switch.

# Mathematical calculation

val1 = 6
val2 = 7
val3 = "s"
result = switch(
    val3,
    "a"= cat("Addition =", val1 + val2),
    "d"= cat("Subtraction =", val1 - val2),
    "r"= cat("Division = ", val1 / val2),
    "s"= cat("Multiplication =", val1 * val2),
    "m"= cat("Modulus =", val1 %% val2),
    "p"= cat("Power =", val1 ^ val2)
)

print(result)
```

Module VIII

Loops

In this module, we are looking at repetitive execution, often simply called loops. As if-statements, loops are not functions, but control statements.

for loops

The simplest and most frequently used type of loops is the for loop. For loops in R always iterate over a sequence (a vector), where the length of the vector defines how often the action inside the loop is executed.

Basic usage: for (<value> in <values>) { <action> }

- <value>: Current loop variable.
- <values>: Set over which the variable iterates. Typically an atomic vector but can also be a list.
- <action>: Executed for each <value> in <values>.
- {...}: As for functions or if-statements – necessary when multiple commands are executed, optional for a single command.

A typical use is to loop over an integer sequence i=1,2,3,..., n=1,2,3,...,. The corresponding for-loop looks as follows:

- In *R*: for (i in 1:n) { ... }.

To see how this works, the two code chunks below show two examples where we once loop over an integer sequence 1:3 (1:3) and a character vector c("Reto", "Ben", "Lea").

```
# Creating 1:3 on the fly.
for (i in 1:3) {
    print(i)
}
```

```
# Creating the vector on the fly.
for (i in c("Reto", "Ben", "Lea")) {
    print(i)
}
```

Explanation: *R* loops over the entire vector, element by element.

1. For the first iteration, the first element of the vector is assigned to the loop variable i.
2. After reaching the end, the loop continues by assigning the second value to the loop variable i (second iteration).
3. This is done until there are no elements left – in this case three iterations. This ends the loop.

```
# Integer sequence
x <- 1:3
# Use vector 'x' for the loop.
for (i in x) {
    print(i)
}
```

```
# Character vector
participants <- c("Reto", "Ben", "Lea")
# Use vector 'participants' for the loop.
for (name in participants) {
    print(name)
}
```

Backward loops

```
(x <- 8:11)
## [1]  8  9 10 11
rev(x)
for (i in 3:1) { print(i) }
## [1] 3
## [1] 2
## [1] 1
for (name in rev(c("Reto", "Ben", "Lea"))) { print(name) }
```

Loops and subsetting

Loops are often used in combination with subsetting. We have a named vector info with two elements:

```
info <- c(name = "Innsbruck", country = "Austria")
```

and would like to loop over all elements (1:2) of the vector. Instead of only printing 1 and 2 we use subetting by index to extract the values from the vector above.

```
for (i in 1:2) {
    print(paste("Element", i, "contains", info[i]))
}
```

Instead of looping over the indices (1:2) we could also loop over the names of the vector (using names() to extract the character vector) and use subsetting by name.

```
for (elem in names(info)) {
    print(paste("Element", elem, "contains", info[elem]))
}
```

Nested for loops

```
for (i in 1:2) {
    for (j in 1:3) {
        print(paste("i =", i, "j =", j))
    }
}
```

Loops and matrices

```
(x <- matrix(c(9, 0, 3, 17, 5, 2), ncol = 3))
# Loop
for (i in 1:2) {
    for (j in 1:3) {
        print(paste0("Element x[", i, ", ", j, "] is ", x[i, j]))
    }
}
```

Note that it is crucial to not mix up the dimensions and/or indices. The following loop

```
for (i in 1:3) {
    for (j in 1:2) {
        print(paste0("Element x[", i, ", ", j, "] is ", x[i, j]))
    }
}
```

while loops

The second type of loop is while. In contrast to a for-loop which runs for a fixed number of iterations, a while-loop runs *while a condition is true*.

Basic usage: while (<condition>) { <action> }.

- <condition>: Logical condition, has to be FALSE or TRUE.
- <action>: Executed as long as the <condition> is TRUE.
- {...}: Necessary for multiple commands, optional for single ones.

Beware of infinite loops! A simple example for an infinite loop is the following simple while-loop.

```
x <- 1
while (x > 0) {
    x <- x + 1
}
```

Useful while loop example

The following shows an example where a while-loop is useful in practice. We want to print all numbers x in 1,2,...,∞1,2,...,∞ as long as x^2 is lower than 20, starting with x <- 0.

```
# Start with 0
x <- 0
# Loop until condition is FALSE
while (x^2 < 20) {
  print(x)      # Print x
  x <- x + 1    # Increase x by 1
}
```

repeat loops

The last one is a repeat-loop. In contrast to the other two the repeat loop runs forever – until we explicitly stop it by calling break.
Basic usage: repeat { <action> }.

- <action>: Executed until the break statement is called. Thus, don't forget to include break.
- {...}: Necessary for multiple commands, optional for single ones.

Remarks:

- More rarely used compared to for and while loops.
- Not necessary for any task in this course!
- (But super simple to write).

Example: We could use a repeat loop to solve the same task as shown above where we would like to get all numbers x∈[0,1,...,∞] where x2<20 like this:

```
# Initialization
x <- 0
# Repeat loop
repeat {
    if (x^2 > 20) break     # Break condition (important)
    print(x)                # print(x)
    x <- x + 1              # Increase x by 1
}
```

Ref:

- Hadley Wickham's Advanced R book
- Roger Peng's R Programming for Data Science book
- DataCamp's Intermediate R course
- Coursera's R Programming course
- Data Carpentry (http://datacarpentry.org/), data camp, data quest, Kaggle
- Harvard Chan Bioinformatics Core (HBC) under the open access terms of the Creative Commons Attribution license (CC BY 4.0),
- The Book of R: A First Course in Programming and Statistics by Tilman M. Davies
- UC Business Analytics R Programming Guide and R_bootcamp