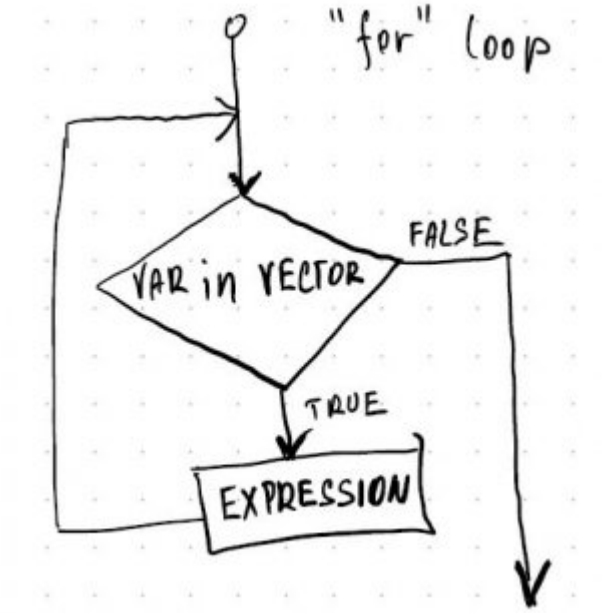
Table of Contents

R constructs

This section contains a list of “R constructs” which we will refer to in the class (e.g. how to define a function, or how to code a loop executing certain code repeatedly).

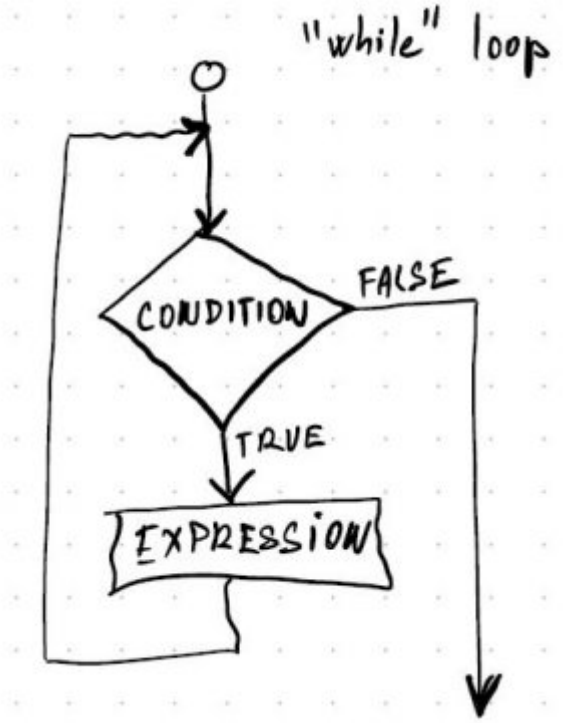
"for" loop




```
for (VAR in VECTOR)
{
  STATEMENT_1
  STATEMENT_2
  ETC
}
```

Create an element VAR, which takes (sequentially) values of elements in VECTOR. The length of VECTOR defines how many times the loop will be executed (= number of elements in `vector = length (VECTOR)`). The body of the function (statements which are executed in each cycle) are wrapped together into curly brackets ({ and }); if only one STATEMENT is inside the body of the loop, the curly brackets can be ommited. The VAR can but does not need to be used in the body of the function. Variables defined within the body of the function and also the VAR variable will stay in the Global Environment (this is different from the `function`, which will create temporary environment, create variables there, and close it after the end of executing the lines of code in the function's body (these variables are not visible from Global Environment).

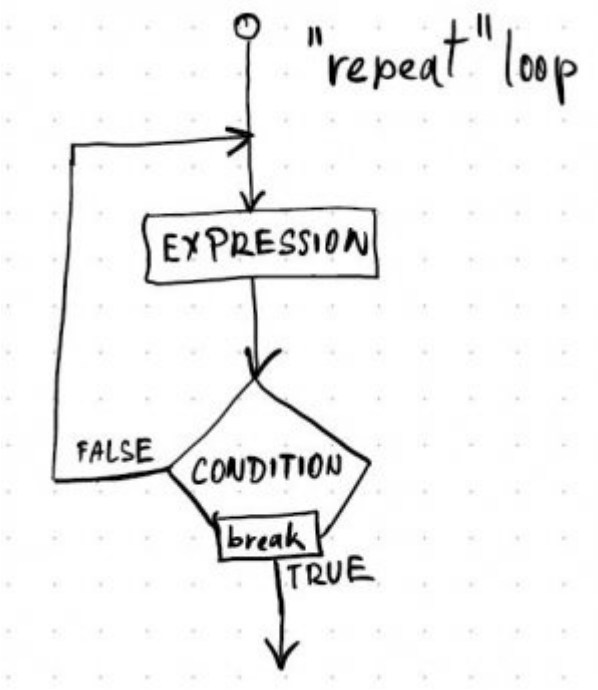
"while" loop




```
while (CONDITION)
{
  STATEMENT_1
  STATEMENT_2
  ETC
}
```

This function can be also translated as “do while the CONDITION is true”. The CONDITION needs to be a logical statement returning either TRUE or FALSE; if FALSE, the loop is interrupted and code jumps out of the loop. In case that the CONDITION does not eventually return FALSE, the looping may last forever (click ESC in that case to stop it).

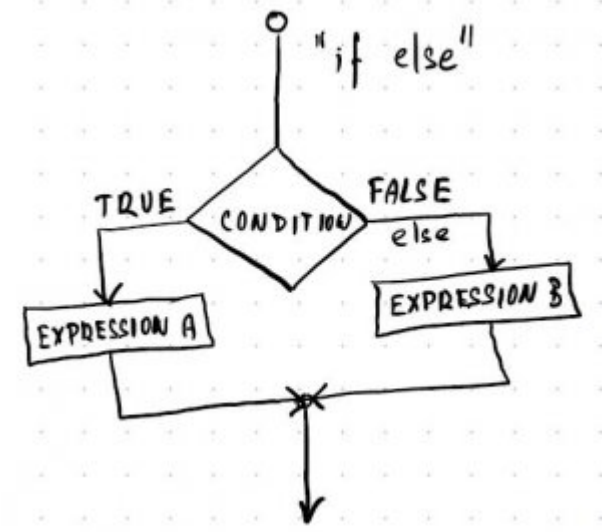
"repeat" loop




```
repeat
{
  STATEMENT1
  STATEMENT2
  IF (CONDITION) break
}
```

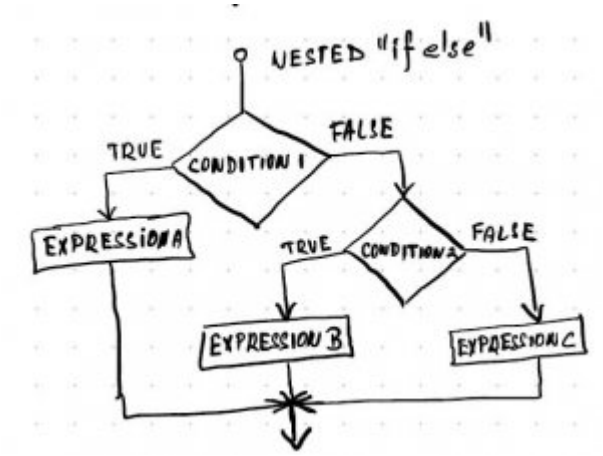
The “repeat” loop has a different logic from the previous two; it “repeats until the CONDITION is true”, after which `break` function needs to be called to *break* out of the loop. It repeats the expression at least once before it evaluates the CONDITIONS. This one is the most dangerous, because, similarly to “while” loop, it may keep repeating forever if the CONDITION does not turn TRUE.

"if" and "else" conditional function



```
if (CONDITION)
{
  STATEMENT_1
  STATEMENT_2
  ETC_1
} else {
  STATEMENT_3
  STATEMENT_4
  ETC_2
  }
```

The “if else” conditional function evaluates whether the CONDITION is TRUE or FALSE, and executes relevant statement. The the `else` part does not need to be present. CONDITION needs to be a single logical value (it can be a complicated logical expression, but if more than one logical value is returned by it, only the first one will be used (this is different from `ifelse` function (below), in which CONDITION can be a vector (or a matrix) with more than one logical value and all are used.



The “if else” functions can be also nested:

```
if (CONDITION_1) {
  STATEMENT_1
  STATEMENT_2
  ETC_1
} else {
  if (CONDITION_2) {
    STATEMENT_3
    STATEMENT_4
    ETC_2
  } else {
      STATEMENT_5
      STATEMENT_6
      ETC_3
    }
  }
```

"ifelse" conditional function

```
ifelse (CONDITION, VAR_IF_TRUE, VAR_IF_FALSE)
```

... where CONDITION can be a vector (or matrix) with more than one logical element - than the function “loops” through the whole vector/matrix and returns the object of the same dimension as CONDITION (dimension = the same number of elements in case of vector, or the same number of rows/columns in the case of matrix)

"function" construct

```
NEW_FUNCTION <- function (ARG1, ARG2, ARG3)
{
  STATEMENT_1
  STATEMENT_2
  STATEMENT_3
  ETC
  return (RESULT)
}
```

The `function` will define a new object, which has the ability to execute the statements inside its definition, while (optionally) supplying values through the arguments of the function (here e.g. ARG1). The function usually has some side effect, e.g. it does a calculation and exports an output (e.g. numerical values), or it does something else (e.g. plotting the figure).

The `return` function (here at the end of the function definition) exports values into the global environment. When you run the function, the function creates a temporary environment and new variables, which are not accessible from the Global Environment (so as they do not mix with those defined in the Global Environment). If you want the function to export something to the global environment, you need to *return* it from the function. This is done either explicitly by using `return` function, or by typing the name of the variable at the end of the function as if you want to print it into the console (means that instead of `return (RESULT)` you may type simply RESULT and it does the same thing; the difference is that `return (RESULT)` can theoretically be somewhere inside the script, not necessarily in the end, while if you type only the name of the variable, the function returns the one which is the last one). You can return only a single object, so if more than one object needs to be returned, wrap them e.g. into a list and return the list of objects. If `return` is used in the body of the function but is not at the end of the script, all lines of code after `return` are ignored and not executed.

Arguments can have *default* value, i.e. the value which will be used if the argument does not have assigned value when the function is called. For example, the function

```
log_sqrt <- function (x)
   log (sqrt (x))
```

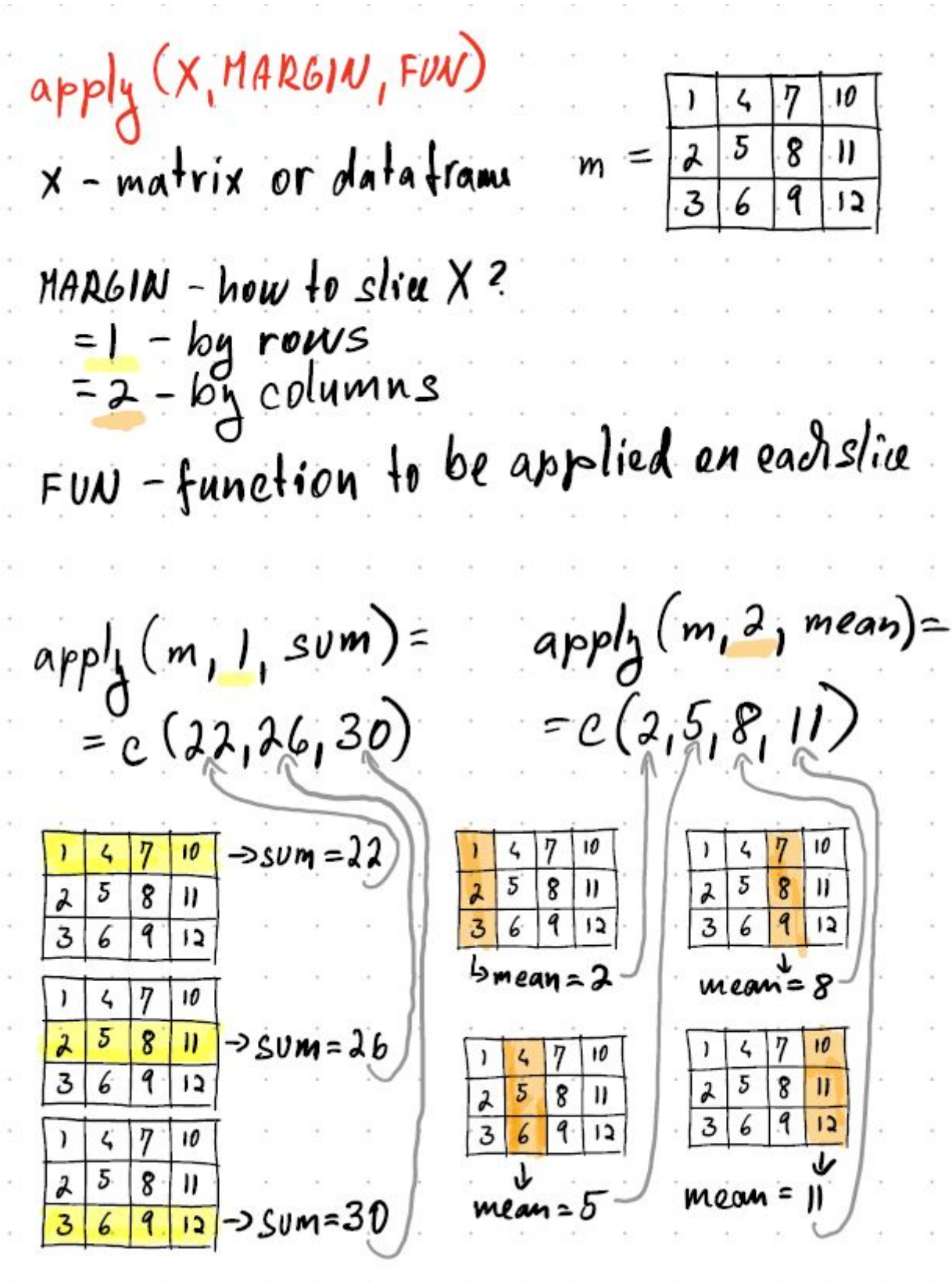
when called, will require the value of the argument x to be defined; but if we modify the definition into

```
log_sqrt <- function (x = 64)
   log (sqrt (x))
```

when we call it without specifying the argument, the value 64 will be used as the default one.

"apply" implicit loop

Function `apply` takes matrix or data frame as an argument X, and cuts in into slices (either by row or by column, depending on the value in argument `MARGIN`). The function `FUN` is then applied on each slice separately, and connected together into resulting vector. “Implicit” loops are doing a very similar thing as “explicit” loops (`for`, `while`, `repeat`), but instead of explicitly typing the whole loop, the whole command has usually very simple structure, and the looping is done implicitly.



There are several other functions in *`apply` family. For example, `lapply` takes as an argument list, applies FUN on each component of the list, and returns a list of the same length as the one which was used as an argument. `sapply` is very similar to `lapply`, but it attempts to return the values in the most simple object (e.g. as a vector instead of the list). Other functions exists (`vapply`, `mapply`, `tapply`, `rapply` etc.).

Plot into a file

```
tiff (filename = 'FILENAME.tiff', width = WIDTH, height = HEIGHT, units =
UNITS, pointsize = POINTSIZE, ...)
PLOT_THE_FIGURE
dev.off ()
```

The logic is the following:

1. To plot into a file, you need to first open appropriate graphics device (`tiff` opens TIFF graphics device, but there is a number of others, e.g. `bmp`, `jpeg`, `png`, `pdf`, `postscript` etc.). Give the file a name with appropriate extension (e.g. `*.tif` in the case of TIFF format, `*.jpg` or `*.jpeg` in the case of JPG, etc) and set up plotting parameters (size of the image by WIDTH or/and HEIGHT, etc.).
2. After you open the graphics device, plot the figure itself (use high-level plotting functions like `plot`, `boxplot`, `barplot`, `hist` etc., and optionally add low-level plotting functions, such as `points`, `text`, `lines`, `axis`, `box`, etc.).
3. When you finish plotting, close the graphics device by `dev.off ()` function, to 1) save the file, 2) to return to the R graphics device. If you forgot to close the device by `dev.off ()`, you cannot open the saved graphical file (and it gets zero bit size). If this happen, type `dev.off ()` several times, untill you see the message `null device`.

```
# Example of plotting into the file:
# 1) Open required graphics device first (PNG format in this case):
png ('cars.png', width = 6, height = 6, units = 'cm', res = 600, pointsize =
8)

# 2) Plot everything you want to plot:
plot (dist ~ speed, data = cars, las = 1, xlab = 'Speed [mph]', ylab =
'Distance [ft]')
abline (lm (dist ~ speed, data = cars))

# 3) Close the graphics device
dev.off ()
```

Plot regression line/curve into the scatterplot

```
## A) Linear regression with linear regression line
# 1) fit the model:
LM <- lm (Y ~ X, data = DATA.FRAME)
FOR.PREDICT <- c(min (DATA.FRAME$X), max (DATA.FRAME$X))
PREDICTED.VALS <- predict (LM, newdata = list (X = FOR.PREDICT))
```

```
# 2) plot the data:
plot (Y ~ X, data = DATA.FRAME)
lines (PREDICTED.VALS ~ FOR.PREDICT)

## B) Linear regression with non-linear (polynomial) regression line
# 1) fit the model:
LM2 <- lm (Y ~ poly (X, 2), data = DATA.FRAME)
FOR.PREDICT <- seq (from = min (DATA.FRAME$X), to = max (DATA.FRAME$X),
length = 100)
PREDICTED.VALS <- predict (LM2, newdata = list (X = FOR.PREDICT))
# 2) plot the data:
plot (Y ~ X, data = DATA.FRAME)
lines (PREDICTED.VALS ~ FOR.PREDICT)

## C) "abline" solution (only for linear regression line)
# 1) fit the model:
LM <- lm (Y ~ X, data = DATA.FRAME)
# 2) plot the data:
plot (Y ~ X, data = DATA.FRAME)
abline (LM)
```

Combination of `lm` (linear model `lm (y ~ x, data)`) and `predict` (predicts the values of x variables lying on the regression curve), and then high-level graphical function (`plot`, plots scatterplot) and low-level one (`lines`, plots the regression line). Although it may feel a bit too complicated, it is the most general option, which allows for plotting any type of fitted curve (not only linear as `abline`, but also non-linear).

Note: the function `predict`, if without argument `newdata`, will return the vector of predicted values for all values in variable x in the model. This may not always be useful, especially if the values in x are not sorted from lowest to highest, and we want to plot the curve (will produce the scatter of lines instead). Also, make sure that the values going to `newdata` argument of `predict` function is a `list`, containing the variable with exactly the same name as in the regression model (X in the example above), otherwise it won't work (it will behave as if the `newdata` argument is not supplied, without any warning).

The `abline` solution may seem as the simplest, but it is the least general - it plots only linear regression line (no non-linear shape), and it plots the curve always across the whole figure space, not only across the span of the data points. This may sometimes be fine, sometimes not, depends.

Monte Carlo permutation test of linear regression

```
X <- trees$Height  # independent (explanatory) variable
Y <- trees$Volume  # dependent variable
NPERM <- 999       # the number of permutations

# 1) Plot observed data, fit the linear regression, and calculate r2
(observed r2)
plot (Y ~ X)
abline (lm (Y ~ X))
```

```
LM <- lm (Y ~ X)
F_OBS <- summary (LM)$fstatistic[1]

# 2) calculate F-value of regression on the original variables after
randomizing one of them (randomized F);
# repeat NPERM-times
F_RAND <- replicate (NPERM, expr = summary (lm (Y ~ sample
(X)))$fstatistic[1])
# or, a bit less condense:
# F_rand <- replicate (NPERM, expr = {
#   LM_rand <- lm (Y ~ sample (X))
#   SU_rand <- summary (LM_rand)
#   F <- SU_rand$fstatistic[1]
#   F
# })


# 3) merge the randomized F-values and observed F-values into one vector
F_ALL <- c (F_RAND, F_OBS)

# 4) calculate the probability that observed F can be generated in case that
the null hypothesis
# is true (ie by regression on randomized original variables)
P <- sum (F_ALL >= F_OBS)/(NPERM+1)
```

Linear regression (`lm` function in R) is the relationship between a dependent (Y) and explanatory (X) variable, fitted by the least square algorithm. The effect size (how good is the fit) of the regression is the r^2 (coefficient of determination, see the definition on Wikipedia), expressing the amount of total variation in the dependent variable explained by given explanatory variable. This explained variation can be tested by parametric test (*F*-test in case of linear regression), or by permutation test (Monte Carlo permutation test, detailed here). Parametric test (*F*-test) uses the results of regression and calculates F-value, which together with given number degrees of freedom allows to lookup appropriate *P*-value (level of significance). The limitation of the parametric test is that it applies only for a situation that Y for each X has a normal distribution, which is often not the case. Alternatively, one can calculate significance using Monte Carlo permutation test. In this test, we create data for which the null hypothesis ("no relationship between X and Y") is true, by permitting one of the variable, and thus breaking any relationship which may exist between these two variables. From such variables (one original and one permuted) we then calculate regression and the *F*-value (or r^2 or other test statistics). If this permutation is replicated many times (e.g. 999 times, i.e. the number of permutations), we get a reasonable estimation of the test statistic (let's talk about F-value here, but we could as well use e.g. r^2) in case that the null hypothesis is true (i.e. the variables are random, without relationship). We can compare this distribution with the real F-value (from regression based on the real data without randomization) and calculate the probability that the observed F-value originates from the distribution of F-values representing the null hypothesis. If this probability (*P*-value) is low enough (e.g. $P < 0.05$), we can reasonably assume that the null hypothesis (of no relationship) can be rejected.