@author Kacper Bohaczyk
@version 13-09-2023

The protocol is structured into two main sections: 'w3school' and 'Einführung in R.' Each of these sections contains several subsections. Additionally, there is a 'Data Camp' section, which remains incomplete due to a lack of access to the license.

w3schools

> https://www.w3schools.com/r/

!!!Most of the lessons thought in the Tutorial section are already present if you are a experienced developer. I recommend to just take a quick look through the Tutorial section and skip to the R- Data-Structures section.!!!
NOTE:! Many sections like arrays or matrices have the same functions. They are shown many times to learn better but the syntax stays the same.

Tutorial

Why should you use R?

- It is a great resource for data analysis, data visualization, data science and machine learning
- It provides many statistical techniques (such as statistical tests, classification, clustering and data reduction)
- It is easy to draw graphs in R, like pie charts, histograms, box plot, scatter plot, etc++
- It works on different platforms (Windows, Mac, Linux)
- It is open-source and free
- It has a large community support
- It has many packages (libraries of functions) that can be used to solve different problems

Syntax

To output text in R, use single or double quotes:

```
"Hello World!"
```

you can also use the `print()` function

```
print("Hello World!")
```

print is necessary when using for loops

```
for (x in 1:10) {
  print(x)
}
```

Comments

Comments starts with a `#`. When executing code, R will ignore anything that starts with `#`

```
# This is a comment
# This lines will be ignored my the compiler
"Hello World!"
```

Variables

Declaring Variables

To assign a value to a variable, use the `<-` sign

```
name <- "John"
age <- 40

name   # output "John"
age    # output 40
```

print function

```
name <- "John Doe"

print(name) # print the value of the name variable
```

You can assign the same value to more than 1 variable at the same time

```
# Assign the same value to multiple variables in one line
var1 <- var2 <- var3 <- "Orange"

# Print variable values
```

```
var1
var2
var3
```

Combining Variables

You can join two or more elements, by using the `paste()` function

```
text <- "awesome"

paste("R is", text)
```

```
text1 <- "R is"
text2 <- "awesome"

paste(text1, text2
```

```
num1 <- 5
num2 <- 10

num1 + num2
```

Combining Text with numbers does not work and throws an exception

Variable names

- A variable name must start with a letter and can be a combination of letters, digits, period(.) and underscore(_). If it starts with period(.), it cannot be followed by a digit.
- A variable name cannot start with a number or underscore (_)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Reserved words cannot be used as variables (TRUE, FALSE, NULL, if...)

```
# Legal variable names:
myvar <- "John"
my_var <- "John"
myVar <- "John"
MYVAR <- "John"
myvar2 <- "John"
.myvar <- "John"

# Illegal variable names:
2myvar <- "John"
```

```
my-var <- "John"
my var <- "John"
_my_var <- "John"
my_v@ar <- "John"
TRUE <- "John"
```

Data-Types

Variables are dynamic. They do not need to be declared with any particular type, and can even change type after they have been set:

```
my_var <- 30 # my_var is type of **numeric**
my_var <- "Sally" # my_var is now of type **character** (aka string)
```

- `numeric` - (10.5, 55, 787)
- `integer` - (1L, 55L, 100L, where the letter "L" declares this as an integer)
- `complex` - (9 + 3i, where "i" is the imaginary part)
- `character` (a.k.a. string) - ("k", "R is exciting", "FALSE", "11.5")
- `logical` (a.k.a. boolean) - (TRUE or FALSE)

With the function `class()` we can check the datatype of an variable

```
# numeric
x <- 10.5
class(x)

# integer
x <- 1000L
class(x)

# complex
x <- 9i + 3
class(x)

# character/string
x <- "R is exciting"
class(x)

# logical/boolean
x <- TRUE
class(x)
```

R Numbers#

There are three number types in R:

- `numeric`
- `integer`
- `complex`

```
x <- 10.5   # numeric
y <- 10L    # integer
z <- 1i     # complex
```

Numeric

A `numeric` data type is the most common type in R, and contains any number with or without a decimal, like: 10.5, 55, 787:

```
x <- 10.5
y <- 55

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```

Integer

Integers are numeric data without decimals. This is used when you are certain that you will never create a variable that should contain decimals. To create an `integer` variable, you must use the letter `L` after the integer value:

```
x <- 1000L
y <- 55L

# Print values of x and y
x
y

# Print the class name of x and y
```

```
class(x)
class(y)
```

complex

```
x <- 3+5i
y <- 5i

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```

Type conversion

You can convert variables with the following functions

- `as.numeric()`
- `as.integer()`
- `as.complex()`

```
x <- 1L # integer
y <- 2 # numeric

# convert from integer to numeric:
a <- as.numeric(x)

# convert from numeric to integer:
b <- as.integer(y)

# print values of x and y
x
y

# print the class name of a and b
class(a)
class(b)
```

R Math

- Addition: `+`

- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `^` - The `^` operator raises the number to its left to the power of the number to its right: for example `3^2` is 9.
- Modulo: `%%` - The modulo returns the remainder of the division of the number to the left by the number on its right, for example 5 modulo 3 or `5 %% 3` is 2.

Built-in Math Functions

min & max

The function `min()` gives us the hlowest number and the function `max()` gives the highest

```
max(5, 10, 15)

min(5, 10, 15)
```

Sqrt

The `sqrt()` function gives us the square root of a number

```
sqrt(16)
```

abs

The `abs()` function returns the absolute (positive) value of a number:

```
abs(-4.7)
```

ceiling & floor

The `ceiling()` function rounds a number upwards to its nearest integer, and the `floor()` function rounds a number downwards to its nearest integer, and returns the result:

```
ceiling(1.4)

floor(1.4)
```

R Strings

String Literals

String litterals are suroundert with "" or ''

`"hello"` is the same as `'hello'`

String Lenght

You can check the Lenght of an String with the `nchar()` function

```
str <- "Hello World!"
nchar(str)
```

Check a String

You can check if a Character or a world is inside the string with the function `grepl()`

```
str <- "Hello World!"

grepl("H", str)
grepl("Hello", str)
grepl("X", str)
```

Escape Characters

To insert characters that are illegal in a string, you must use `\`

```
|\\|Backslash|
|\n|New Line|
|\r|Carriage Return|
|\t|Tab|
|\b|Backspace|
```

Boolean

Boolean has two values True or False. It is simular to an binary system only 0 or 1 only off or on.

if statements

You can use if statements to implement logic

```
a <- 200
b <- 33

if (b > a) {
  print ("b is greater than a")
} else {
  print("b is not greater than a")
}
```

Operators

There are many groups of operators in R:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Miscellaneous operators

operators are used on values

Assigment Operators

```
|Operator|Name|Example|
|+|Addition|x + y|
|-|Subtraction|x - y|
|*|Multiplication|x * y|
|/|Division|x / y|
|^|Exponent|x ^ y|
|%%|Modulus (Remainder from division)|x %% y|
|%/%|Integer Division|x%/%y|
```

Comparison Operators

```
|==|Equal|x == y|
|!=|Not equal|x != y|
|>|Greater than|x > y|
|<|Less than|x < y|
|>=|Greater than or equal to|x >= y|
|<=|Less than or equal to|x <= y|
```

Logical Operators

```
|&|Element-wise Logical AND operator. It returns TRUE if both elements are
TRUE|
|&&|Logical AND operator - Returns TRUE if both statements are TRUE|
|\||Elementwise- Logical OR operator. It returns TRUE if one of the
statement is TRUE|
|\||Logical OR operator. It returns TRUE if one of the statement is TRUE.|
|!|Logical NOT - returns FALSE if statement is TRUE|
```

Miscellaneous Operators

```
|:|Creates a series of numbers in a sequence|x <- 1:10|
|%in%|Find out if an element belongs to a vector|x %in% y|
|%*%|Matrix Multiplication|x <- Matrix1 %*% Matrix2|
```

If else

```
|==|Equal|x == y|
|!=|Not equal|x != y|
|>|Greater than|x > y|
|<|Less than|x < y|
|>=|Greater than or equal to|x >= y|
|<=|Less than or equal to|x <= y|
```

like in java or other programming languages if a code is written inside an if statement and the conditions are met the code will be executed if not the code will be skipped.

```
a <- 33
b <- 200

if (b > a) {
  print("b is greater than a")
}
```

`else if` is executed if the previous conditions are not met and the conditions in the else if are met.

```
a <- 33
b <- 33

if (b > a) {
  print("b is greater than a")
} else if (a == b) {
  print ("a and b are equal")
}
```

`else` is executed if the previous conditions are not met.

```
a <- 200
b <- 33
```

```
if (b > a) {
  print("b is greater than a")
} else if (a == b) {
  print("a and b are equal")
} else {
  print("a is greater than b")
}
```

nested if statements
are if statements inside if statements

```
x <- 41

if (x > 10) {
  print("Above ten")
  if (x > 20) {
    print("and also above 20!")
  } else {
    print("but not above 20.")
  }
} else {
  print("below 10.")
}
```

AND OR Operators
The `&` symbol is used to combine 2 statements. It is true if both conditions are met
The `|` symbol is used to combine 2 statements. It is true if one condition is met

Loops

while
while loops are executed till the conditions changes from true to false

```
i <- 1
while (i < 6) {
  print(i)
  i <- i + 1
}
```

for
A `for` loop is used for iterating over a sequence:

```r
for (x in 1:10) {
  print(x)
}
```

```r
fruits <- list("apple", "banana", "cherry")

for (x in fruits) {
  print(x)
}
```

break

with break you can end a loop even if the conditions are met

```r
i <- 1
while (i < 6) {
  print(i)
  i <- i + 1
  if (i == 4) {
    break
  }
}
```

next

With the `next` statement, we can skip an iteration without terminating the loop:

```r
i <- 0
while (i < 6) {
  i <- i + 1
  if (i == 3) {
    next
  }
  print(i)
}
```

nested loops

like with nested if`s you can nest loops

```r
adj <- list("red", "big", "tasty")

fruits <- list("apple", "banana", "cherry")
  for (x in adj) {
    for (y in fruits) {
```

```
      print(paste(x, y))
  }
}
```

Functions

with `function()` you can create a function

```
my_function <- function() { # create a function with the name my_function
  print("Hello World!")
}
```

you can call a function by using the name followed by ``()

```
my_function <- function() {
  print("Hello World!")
}

**my_function()** # call the function named my_function
```

Arguments

Information can be passed through `()

```
my_function <- function(fname) {
  paste(fname, "Griffin")
}

my_function("Peter")
my_function("Lois")
my_function("Stewie")
```

Default Parameter Value

If no arguments are passed the default value will be passed

```
my_function <- function(country = "Norway") {
  paste("I am from", country)
}

my_function("Sweden")
my_function("India")
```

```
my_function() # will get the default value, which is Norway
my_function("USA")
```

Return Values

Use the `return()` function to get a value

```
my_function <- function(x) {
  return (5 * x)
}

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Nested Function

You can create nested functions the following ways:

- Call a function within another function.
- Write a function within a function.

```
Nested_function <- function(x, y) {
  a <- x + y
  return(a)
}

Nested_function(Nested_function(2,2), Nested_function(3,3))
```

Or write a function within a function

```
Outer_func <- function(x) {
  Inner_func <- function(y) {
    a <- x + y
    return(a)
  }
  return (Inner_func)
}
output <- Outer_func(3) # To call the Outer_func
output(5)
```

Recursion

recursion means that a function can call itself

In the following example an recursion occurs till the argument k is not 0

```
tri_recursion <- function(k) {
  if (k > 0) {
    result <- k + tri_recursion(k - 1)
    print(result)
  } else {
    result = 0
    return(result)
  }
}
tri_recursion(6)
```

Global Variables
Variables not decelerated within a function are called global variables
They can be accessed everywhere

```
txt <- "awesome"
my_function <- function() {
  paste("R is", txt)
}

my_function()
```

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

```
txt <- "global variable"
my_function <- function() {
  txt = "fantastic"
  paste("R is", txt)
}

my_function()

txt # print txt
```

If you want to create a global variable within a function use `<<-`

```r
my_function <- function() {
txt <<- "fantastic"
  paste("R is", txt)
}

my_function()

print(txt)
```

if you want to change a global Variable within a function use `<<-`

```r
txt <- "awesome"
my_function <- function() {
  txt <<- "fantastic"
  paste("R is", txt)
}

my_function()
```

R Data-Structures

Vectors

A vector is a list of items of the same type
To make a vector we use the `c()` function

```r
# Vector of strings
fruits <- c("banana", "apple", "orange")

# Print fruits
fruits
```

A vector with numerical values in a sequence is created with the `:` operator:

```r
# Vector with numerical values in a sequence
numbers <- 1:10

numbers
```

This can be used with other Datatypes like boolean, decimals etc

```r
# Vector of logical values
log_values <- c(TRUE, FALSE, TRUE, FALSE)

log_values
```

To get the amount of items in a vector use `length()`

```r
fruits <- c("banana", "apple", "orange")

length(fruits)
```

To sort items in a vector alphabetically or numerically, use the `sort()` function:

```r
fruits <- c("banana", "apple", "orange", "mango", "lemon")
numbers <- c(13, 3, 5, 7, 20, 2)

sort(fruits)  # Sort a string
sort(numbers) # Sort numbers
```

To access an item from a vector use the index in Brackets [index]

```r
fruits <- c("banana", "apple", "orange")

# Access the first item (banana)
fruits[1]
```

To access multiple elements referre to different index positions with the `c()` function:

```r
fruits <- c("banana", "apple", "orange", "mango", "lemon")

# Access the first and third item (banana and orange)
fruits[c(1, 3)]
```

To access all items except one item simply use negative items

```r
fruits <- c("banana", "apple", "orange", "mango", "lemon")

# Access all items except for the first item
fruits[c(-1)]
```

you can change the value of a specific item by referring to the index number:

```r
fruits <- c("banana", "apple", "orange", "mango", "lemon")

# Change "banana" to "pear"
fruits[1] <- "pear"

# Print fruits
fruits
```

repeat Vector

To repeat use the `rep()` function

```r
repeat_each <- rep(c(1,2,3), each = 3)

repeat_each
```

Sequenced Vectors

```r
numbers <- 1:10

numbers
```

To make the steps bigger or smaller use `seq()` function

```r
numbers <- seq(from = 0, to = 100, by = 20)

numbers
```

Note: The `seq()` function has three parameters: `from` is where the sequence starts, `to` is where the sequence stops, and `by` is the interval of the sequence.

Lists

A List a can contain many different data types. To create a list use the `list` function

```r
# List of strings
thislist <- list("apple", "banana", "cherry")

# Print the list
thislist
```

access List

You can access the list items by referring to its index number, inside brackets. The first item has index 1, the second item has index 2, and so on:

```
thislist <- list("apple", "banana", "cherry")

thislist[1]
```

change Item Value

to change a value use the index

```
thislist <- list("apple", "banana", "cherry")
thislist[1] <- "blackcurrant"

# Print the updated list
thislist
```

length

to check the length of a list use `length()` function

```
thislist <- list("apple", "banana", "cherry")

length(thislist)
```

check if

to check if a specific item is inside a List use the `%in%` operator

```
thislist <- list("apple", "banana", "cherry")

"apple" %in% thislist
```

Add to list

End:

To add an item to the end of an list use `append()` function

```
thislist <- list("apple", "banana", "cherry")

append(thislist, "orange")
```

Specific:
To add an item to the right of a specified index, add " `after=_index number_` " in the `append()` function:

```
thislist <- list("apple", "banana", "cherry")

append(thislist, "orange", after = 2)
```

Remove Item
You can also remove list items. The following example creates a new, updated list without an "apple" item:

```
thislist <- list("apple", "banana", "cherry")

newlist <- thislist[-1]

# Print the new list
newlist
```

Range of indexes
You can specify a range by telling the start point and the end

```
thislist <- list("apple", "banana", "cherry", "orange", "kiwi", "melon",
"mango")

(thislist)[2:5]
```

Loop
to loop through a list use for loops

```
thislist <- list("apple", "banana", "cherry")

for (x in thislist) {
  print(x)
}
```

Join Lists
With the `c()` function you can combine two elements together

```
list1 <- list("a", "b", "c")
list2 <- list(1,2,3)
list3 <- c(list1,list2)
```

```
list3
```

Matrices

A matrix is a set with columns and rows. See it as 2 Dimensional
It is created with the `matrix()` function. To set amount of rows and columns use the `nrow` and `ncol` parameters

```
# Create a matrix
thismatrix <- matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)

# Print the matrix
thismatrix
```

You can create matrix by string

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2,
ncol = 2)

thismatrix
```

Accessing Matrix

You can access the items by using `[ ]` brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position:

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2,
ncol = 2)

**thismatrix[1, 2]**
```

With the comma after the number in the bracket you can specify that the whole row will be accessed

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2,
ncol = 2)

**thismatrix[2,]**
```

With the comma before the number in the bracket you can specify that the whole column will be accessed

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2,
ncol = 2)

**thismatrix[,2]**
```

Access more than one Row

You can access more than one row be using the `c()` function

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape",
"pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)

thismatrix[c(1,2),]
```

Access more than one column

You can access more than one column be using the `c()` function

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape",
"pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)

thismatrix[, c(1,2)]
```

Add Rows and Columns

To add columns use the `cbind()` function

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape",
"pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)

newmatrix <- cbind(thismatrix, c("strawberry", "blueberry", "raspberry"))

# Print the new matrix
newmatrix
```

To add rows use the `rbind()` function

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange","grape",
"pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)

newmatrix <- rbind(thismatrix, c("strawberry", "blueberry", "raspberry"))

# Print the new matrix
newmatrix
```

Remove Rows and Columns

Use the `c()` function to remove rows and columns in a Matrix:

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange", "mango",
"pineapple"), nrow = 3, ncol =2)

#Remove the first row and the first column
thismatrix <- thismatrix[-c(1), -c(1)]

thismatrix
```

check if item exists

to check if an item exists use the `%in%` function

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2,
ncol = 2)

"apple" %in% thismatrix
```

Number of Rows and Columns

`dim()` is used find the number of rows and columns in a Matrix:

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2,
ncol = 2)

dim(thismatrix)
```

Length

`length()` is used to find the dimension of a Matrix:

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2,
ncol = 2)

length(thismatrix)
```

Loop Through a Matrix

With a a `for` loop you can go through a matrix. The loop will start at the first row, moving right:

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2,
ncol = 2)
```

```
for (rows in 1:nrow(thismatrix)) {
  for (columns in 1:ncol(thismatrix)) {
    print(thismatrix[rows, columns])
  }
}
```

Combine two Matrices

You can use the `rbind()` or `cbind()` function to combine two or more matrices together:

```
# Combine matrices
Matrix1 <- matrix(c("apple", "banana", "cherry", "grape"), nrow = 2, ncol =
2)
Matrix2 <- matrix(c("orange", "mango", "pineapple", "watermelon"), nrow = 2,
ncol = 2)

# Adding it as a rows
Matrix_Combined <- rbind(Matrix1, Matrix2)
Matrix_Combined

# Adding it as a columns
Matrix_Combined <- cbind(Matrix1, Matrix2)
Matrix_Combined
```

Arrays

Arrays are 3 or more Dimensional

With the `array()` function we can create an array and with the `dim` parameter we can specify the dimensions:

```
# An array with one dimension with values ranging from 1 to 24
thisarray <- c(1:24)
thisarray

# An array with more than one dimension
multiarray <- array(thisarray, dim = c(4, 3, 2))
multiarray
```

By referring to an index position you can access an array.

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))
```

```
multiarray[2, 3, 2]
```

With the `c()` function you can access a whole row or column

```
thisarray <- c(1:24)

# Access all the items from the first row from matrix one
multiarray <- array(thisarray, dim = c(4, 3, 2))
multiarray[c(1),,1]

# Access all the items from the first column from matrix one
multiarray <- array(thisarray, dim = c(4, 3, 2))
multiarray[,c(1),1]
```

Check if

If you want to find an element inside an array use `%in%`

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

2 %in% multiarray
```

amount of rows

With `dim()` you can get the amount of rows and columns

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

dim(multiarray)
```

Loop

With an `for` loop through an array

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

for(x in multiarray){
  print(x)
}
```

Data Frames

Frames can have different types of data inside it.
However each columns can only have the same data-type
With `data.frame()` we can create a Data Frame

```
# Create a data frame
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Print the data frame
Data_Frame
```

With the `summary()` function we summarize data from frames

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame

summary(Data_Frame)
```

Access
We can access frames through brackets `[]` or `$`

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame[1]

Data_Frame[["Training"]]

Data_Frame$Training
```

Add rows
With the `rbind()` function we add rows

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Add a new row
New_row_DF <- rbind(Data_Frame, c("Strength", 110, 110))

# Print the new row
New_row_DF
```

Add columns

With the `cbind()` function we can add new columns

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Add a new column
New_col_DF <- cbind(Data_Frame, Steps = c(1000, 6000, 2000))

# Print the new column
New_col_DF
```

Remove rows and columns

The `c()` function is used to remove rows and columns

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Remove the first row and column
Data_Frame_New <- Data_Frame[-c(1), -c(1)]

# Print the new data frame
Data_Frame_New
```

Amount of

With the `dim()` function we can get the amount of rows and columns in a Frame

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

dim(Data_Frame)
```

The `ncol()` function is used to find the number of columns
The `nrow()` to find the number of rows

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

ncol(Data_Frame)
nrow(Data_Frame)R
```

Length

With the `length()` function we can get the number of columns inside a Frame

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

length(Data_Frame)
```

combining

To combine vertically two or more frames use the `rbind()` function

```
Data_Frame1 <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
```

```
)

Data_Frame2 <- data.frame (
  Training = c("Stamina", "Stamina", "Strength"),
  Pulse = c(140, 150, 160),
  Duration = c(30, 30, 20)
)

New_Data_Frame <- rbind(Data_Frame1, Data_Frame2)
New_Data_Frame
```

To combine horizontally two or more frames use the `cbind()` function

```
Data_Frame3 <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame4 <- data.frame (
  Steps = c(3000, 6000, 2000),
  Calories = c(300, 400, 300)
)

New_Data_Frame1 <- cbind(Data_Frame3, Data_Frame4)
New_Data_Frame1
```

Factors

Factors are used to categorize data
Use the `factor()` function to create a Factor

```
# Create a factor
music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz",
"Rock", "Jazz"))

# Print the factor
music_genre
```

You can see a level like a category
The `levels()` function is used to print the selected level

```r
music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz",
"Rock", "Jazz"))

levels(music_genre)
```

You can also set the levels, by adding the `levels` argument inside the `factor()` function:

```r
music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz",
"Rock", "Jazz"), levels = c("Classic", "Jazz", "Pop", "Rock", "Other"))

levels(music_genre)
```

length()

With the `length()` function we get the amount of items there are in the factor

```r
music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz",
"Rock", "Jazz"))

length(music_genre)
```

Access Factors

To access an item use brackets and an index number

```r
music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz",
"Rock", "Jazz"))

music_genre[3]
```

change item value

You can change an item value by referring to the index value

```r
music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz",
"Rock", "Jazz"))

music_genre[3] <- "Pop"

music_genre[3]
```

Einführung in R

Kapitel 2

Most of the Themes from "Kapitel 2" are already covered in the "w3school" section. That is why we skipped rewriting some functions in detail

Arithmetische Operatoren

```
|Operator|Name|Example|
|+|Addition|x + y|
|-|Subtraction|x - y|
|*|Multiplication|x * y|
|/|Division|x / y|
|^|Exponent|x ^ y|
|%%|Modulus (Remainder from division)|x %% y|
|%/%|Integer Division|x%/%y|
```

Logische Operatoren und Funktionen

```
|==|Equal|x == y|
|!=|Not equal|x != y|
|>|Greater than|x > y|
|<|Less than|x < y|
|>=|Greater than or equal to|x >= y|
|<=|Less than or equal to|x <= y|
```

```
abs(x) Betrag
sqrt(x) Quadratwurzel
ceiling(x) Aufrunden: ceiling(3.475) wir zu 4
floor(x) Abrudnen: floor(3.475) wird zu 3
round(x, digits = n) Runden round(3.475, digits = 2) ist 3
log(x) Natürlicher Logarithmus
log(x , base = n) LOgarithmus zur Basis n
log2(x) Logarithmus zur Basisi 2
log10(x) Logarithmus zur BAsis 10
exp(x) Exponentialfunktion: e^x
```

Statistische Funktionen

```
mean(x, na.rm = FALSE) Mittelwert
sd(x) Standardabweichung
```

```
var(x) Varianz
median(x) Median
quantile(x, probs) Quantile von X.
sum(x) Summe
min(x) Minimalwert x_min
max(x) Maximalwert x_max
range(x) x_min und x_max
```

nützliche Funktionen

```
c() Combine: kreiert einen Vektor
seq(from, to, by) Generiert eine Sequenz
rep(x, times, each) Wiederholt x
times: die Sequenz wird n-mal wiederholt
each: jedes Element wird n-mal wiederholt
head(x, n = 6) Zeigt die n ersten Elemente von x an
tail(x, n = 6) Zeigt die n letzten Elemente von x an
```

Variablennammen

Variable names must start with a lowercase letter and are not allowed to have spaces inside. Words are separated by underscores `_`

```
# gute Variablennamen [](https://methodenlehre.github.io/einfuehrung-in-
R/chapters/02-R-language.html#cb148-2)x_mean [](
(https://methodenlehre.github.io/einfuehrung-in-R/chapters/02-R-
language.html#cb148-3)x_sd
```

Funktionen aufruffen

To call a function simply call it by there name followed by `()` Inside the paramethers we can give arguements

```
function_name(arg1, arg2 = val2)
```

verschachtelung von Funktionen

We can use the output of another function inside another function. this is called "verschachtelung"

Datentypen

numeric vectors: integer and doubles

character vectors: Strings/text
logical vectors: Boolean TRUE/FALSE

- Typ: `typeof()` : What type is it?
- Länge: `length()` : How many elements has it?
- Attribute: `attributes()` : additional values atributes

Missing Values

Missing values are declared with `NA`. The Function `is.na` can test if something has an missing vlaue

Character vectors
The Function paste() merches two character vectors together

Factors
Think about factors like about arrays. Factors can save many values of the same datatype. To declare a factor we use the `factor()` function

`relevel()`, is used to determine the reference category.

List
List can hold many different datatypes inside
We can define a list with the function `list()`
There are also `named lists` witch are typicality an output of an function
With `$` and tabulator after the name all elements of the list are shown

Data Frames
Data Frames are the most importend object in R
A Data Frame is constructed from rows and columns
Data-Frames have a 2 Dimensional structure
Data-Frames are called `tibbles` or `tbl`
To define a Frame we use the `tibble()` function
The attributes of a Data-Frames are `colnames()` and `rownames()`
`ncol()` amount of columns
`nrow()` amount of rows
Frames can be identified as a List or a Frame
List is available through `$`
Matrix available through `[`
Available through columnnummer, rownummer]

Datensätze

`rbind()` or `bind_rows()` are in the `dplyr` package

Statistikprogramme:

- SPSS
- jamovi
- *wide* Format is not optimized for all Applications
- many require the long format
- long is used for ggplot2
  Datenkonvertierung is also called Reshaping
- *tidy* Data = clean = long > wide
- Manual reshaping takes long and should be avoided
- *Pipe* Operator `|>`
- we often import data from SPSS or EXCEL
- There are 2 options to import Data into R Studio
- With the GUI under File > Import Dataset
- Or with the functions `read_csv()`, `read_csv2()`, `read_sav()` and `read_excel()`.
- CSV Files
- 1. Import via GUI
- Convert Group-Variables into Factors
- use GUI with Dropdown menu
- To save a DataFrame as an CSV File we use the `write_csv()` function

SPSS Files

- 1 `read_sav()` inside the Package `haven`
- The Difference between CSV Files and SPSS is that Data Frame have extra atributes
- SPSS coded numericly
- The descriptions are saved as attributes
- Use `as_factor()` from the package `haven` to convert *labelled double* into factors
- Use Option `levels = "default"` so if labels exist they are used.
- Other options are `"both"`, `"labels"` and `"values"`
- To create an "ordinaler Faktor" we use the argument `ordered` wich can be TRUE or FALSE

Exel

- 1 Import
- `read_excel()` from the package `readxl`
- convert into Factors

RDATA

- not .txt FIle | it is a Binary File
- less Data gets lost
- RDATA only readebly in R language
- Function `save()` is used to save objects
- Function `load()` is used to load those saved objects

Transform Data

Wide has more columns
long has more rows
converting Data into tidy(long) Data is called "data wrangling"
Packages:
`tidyr` for transformation
`dplyr` for manipulation
`forcats` for Factors

```
|`tidyr`|`pivot_longer()`|erhöht die Anzahl der Zeilen, verringert die
Anzahl der Spalten|
|`tidyr`|`pivot_wider()`|verringert die Anzahl der Zeilen, erhöht die Anzahl
der Spalten|
|`tidyr`|`drop_na()`|löscht alle Zeilen eines Datensatzes, die missing
values (NA) enthalten|
|`dplyr`|`rename()`|zum Umbenennen von Variablen|
|`dplyr`|`select()`|wählt Variablen (Spalten) aus|
|`dplyr`|`filter()`|wählt Beobachtungen (Zeilen) aus|
|`dplyr`|`arrange()`|sortiert einen Datensatz nach einer bestimmten
Variablen|
|`dplyr`|`mutate()`|erstellt neue Variablen und ändert bereits vorhandene
Variablen|
|`dplyr`|`recode()`|rekodiert numerische Variablen|
|`forcats`|`fct_recode()`|zum Rekodieren/Umbenennen von Faktorstufen|
|`dplyr`|`group_by()`|ermöglicht Operationen an Teilmengen der Daten|
|`dplyr`|`summarize()` / `summarise()`|fasst Daten zusammen|
```

`filter()`, `select()`, `summarize()` and `mutate()` have scoped Versions
that means they can us the same transformation for many variables at the same time
`*_all` uses the function on all

`*_at` uses the function on the variables that are determined by `vars()`
`*_if` uses the function on the variables which are designed by the Condition
We nest the functions
The output of one function is used as the input for the next function
Con we need to declare 2 extra variables that we don't need
`pipe` Operator is cleaner than nesting
`|>` pipe is used to give the output to the next function
pipe must not be the fist argument
`_` is used to determine the argument position
`pivot_longer()` and `pivot_wider()` are inside the package `tidyr`
`pivot_longer()` is used to convert *wide* into long
`pivot_wider()` is used to convert long into wide

Grafiken mit ggplot2

Package `ggplot2`

- intuitiven Syntax,
- need *long* format
- 1. ggplot() creates a Plot Object
- define “aesthetic mappings” with `aes()`
- with `geom_` and a word behind we can tell how it will be plotted
- with `+` we add geometrical objects together
- `fill`, `color`, `shape`, `linetype`, `group` can be added to the `aes()` function
- `geom_jitter()` draws points with “jittering" together
- `geom_bar()` is used to show bars
- `colors()` is used to show better visualy
- `position = "dodge"` is used to show bars next to each other
- `position = "identity"` is used to stack bars at the same place
- `geom_histogram()`
- `binwidth` is very importend
- `y = after_stat(density)` y-Achse is relative and not absolute
- `geom_point()` or `geom_line()`
- `linteype` can be `"blank", "solid", "dashed", "dotted", "dotdash", "longdash"` `oder "twodash"`
- `geom_count()` counts the frequency of Y and X
- `facet_wrap()` and `facet_grid()` are used to separate graphics
- `ggtitle()` gives the function a Title
- `labs()` changes the legend

- `ggsave()` saves graphics
- Graphics can be saved into the formats eps, ps , tex ,pdf, jpeg ,tiff, svg and wmf

Datacamp

> https://campus.datacamp.com/courses/free-introduction-to-r/chapter-2-vectors-2?ex=1

Lesson 1

Mathematical functions

The Language R allows us to use basic mathematical functions like

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `^` - The `^` operator raises the number to its left to the power of the number to its right: for example `3^2` is 9.
- Modulo: `%%` - The modulo returns the remainder of the division of the number to the left by the number on its right, for example 5 modulo 3 or `5 %% 3` is 2.

Lesson 2

Creating variables

Syntax

```
my_var <- 4
```

First you write the name you want the variable you have. Then you follow it up with an <- and a value you want to assign

you can also use mathematical functions together with variables

```
# Assign a value to the variables my_apples and my_oranges

my_apples <- 5

my_oranges <- 6

# Add these two variables together
```

```
my_apples + my_oranges

# Create the variable my_fruit

my_fruit <- my_apples + my_oranges
```

Lesson 3

- Decimal values like `4.5` are called **numerics**.
- Whole numbers like `4` are called **integers**. Integers are also numerics.
- Boolean values (`TRUE` or `FALSE`) are called **logical**.
- Text (or string) values are called **characters**.

```
# Change my_numeric to be 42

my_numeric <- 42

# Change my_character to be "universe"

my_character <- "universe"

# Change my_logical to be FALSE

my_logical <- FALSE
```

Lesson 4

you can print classes with the `class()` statement

```
# Declare variables of different types

my_numeric <- 42

my_character <- "universe"

my_logical <- FALSE

# Check class of my_numeric

class(my_numeric)

# Check class of my_character
```

```
class(my_character)

# Check class of my_logical

class(my_logical)
```