

# EinfuehrungR

Christof Zlabinger

2023-09-26

## R Tuotorial

### Syntax

#### Syntax

- Text:
  - "Hello, World!"
- Nummern:
  - 5
- Simple Rechnungen:
  - 5+5

#### Print

- Output Text:
  - "Hello, World"
- Print auf die Console:
  - `print("Hello, World!")`

### Comments

Kommentare fangen mit `#` an und führen dazu das eine Zeile Code vom Compiler ignoriert wird.

### Variables

#### Variables

Es gibt keinen Befehl um Variabeln zu erstellen, sie werden erstellt sobaldt ihnen das erste mal ein Wert zugewiesen wird. Einer Variable wir ein Wert zugewiesen mit `<-` . Eine Variable wird geprinted indem man ihren namen schreibt. Beispiel:

```
name <- "Christof" # Zuweisen einer Variable
```

```
name # Printen einer Variable
```

```
## [1] "Christof"
```

#### Concatenate Elements

Zwei Variabeln können kombiniert werden mittels `paste()` Beispiel:

```
text <- "Christof"
```

```
paste("My name is", name)
```

```
## [1] "My name is Christof"
```

Es ist auch möglich zwei Variablen miteinander zu kombinieren.

```
text1 <- "My name is"
```

```
text2 <- "Christof"
```

```
paste(text1, text2)
```

```
## [1] "My name is Christof"
```

Für Zahlen kann ein '+' verwendet werden

```
num1 <- 5
```

```
num2 <- 5
```

```
num1 + num2
```

```
## [1] 10
```

Wenn versucht wird einen String und eine Zahl zu kombinieren, wird R einen Error werfen.

```
text <- "Hello"
```

```
num <- 5
```

```
num + text
```

```
#Error in num + text : non-numeric argument to binary operator
```

## Multiple Variables

Es ist möglich einen Wert mehreren Variablen zu zuweisen

```
var1 <- var2 <- var3 <- "Hello, World!"
```

## Variable Names

Regeln: \* Eine Variable muss mit einem Buchstaben starten \* Sie können aus Buchstaben, Zahlen, und Punkten (.) sein. \* Eine Variable kann nicht mit einem \_ oder einer Zahl starten \* Variablenamen sind case sensitive \* Bestimmte Wörter können nicht verwendet werden (Bsp.: TRUE, FALSE, NULL, if, ...)

## Data Types

### Data Types

Variablen haben keinen fixen Typ. Der Typ wird automatisch festgelegt und kann sogar geändert werden.

- 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)

Die `class()` Methode überprüft den Datentyp der übergebenen Variable

## Numbers

Es gibt drei Arten von Nummern:

1. numeric # 10.5
2. integer # 10L
3. complex # 1i

## Numeric

Meist genutzter Typ. Kann Nummern mit oder ohne Dezimalzahlstelle sein: 10.5, 55, 123

## Integer

numerics ohne Dezimalstelle. Um einen Integer zu erstellen L an die Zahl anhängen: 10L, 55L

## Complex

Beinhaltet einen imaginären teil  $i$

## Type Conversion

Ein Typ kann zu einem anderen konvertiert werden.

```
as.numeric()
```

```
## numeric(0)
```

```
as.integer()
```

```
## integer(0)
```

```
as.complex()
```

```
## complex(0)
```

## Math

### Simple Math

Es können einfache mathematische Rechnungen mittels Operatoren durchgeführt werden.

- + Um Zahlen zu addieren
- - Um Zahlen zu dividieren

### Built-in Math functions

`min(x, y, z, ...)` um das Minimum eines Sets zu finden `max(x, y, z, ...)` um das Maximum eines Sets zu finden

`sqrt(x)` um die Wurzel einer Zahl zu finden

`abs(x)` um den Absoluten Wert einer Zahl zu finden

`ceiling(x)` rundet die Zahl auf die nächst höchste ganze Zahl auf

`floor(x)` rundet die Zahl auf die nächst niedrigere ganze Zahl ab

## String

### String

Strings werden verwendet um Text zu speichern. Sie werden mit ' oder mit " gekennzeichnet. (Bsp.: "Hello, World!", 'Hello, World!')

## Assign a String to a Variable

Ein String wird einer Variabel mittels `<-` zugewiesen.

```
str <- "Hello, World!"
```

## Multiline Strings

Ein String kann auch mehrere Zeilen lang sein.

```
str <- "Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."
```

Es wird ein `\n` am ende jeder Zeile hinzugefügt

Wenn die Zeilenumbrüche genau übernommen werden sollen dann muss `cat()` verwendet werden.

## String length

Mittels `nchar()` kann die Länge eines Strings ermittelt werden.

## Check a String

Um zu überprüfen ob ein String einen Character oder einen anderen String beinhaltet wird `grepl("Search", str)` verwendet.

## Combine two Strings

Um zwei Strings zu vereinen wird `paste()` verwendet,

## Escape Characters

### Escape Characters

Um Character die nicht in einem String verwendet werden dürfen müssen sie escaped werden. Ein Character kann escaped werden indem ein `\` davor hinzugefügt wird. Bsp.:

```
str <- "Hello, "Christof"!"
```

```
Error: unexpected symbol in "str <- "Hello, "Christof"
```

Um diesen Fehler zu verhindern kann `\` verwendet werden.

```
str <- "Hello, \"Christof\"!"
```

Andere Escape Character sind:

Code	Result
\\	Backslash
\n	New line
\r	Carriage Return
\t	Tab
\b	Backspace

## Booleans

Es können Werte verglichen werden und diese können entweder `TRUE` oder `FALSE` sein.

```
10 > 9      # TRUE
```

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

```
10 == 9     # FALSE
```

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

```
10 < 9      # FALSE
```

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

Dies ist auch mit Variablen möglich.

```
a <- 10
```

```
b <- 9
```

```
a > b
```

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

## Operators

### Operators

Operatoren werden verwendet um Operationen durchzuführen.

Zum Beispiel können zwei Zahlen mittels eines `+` addiert werden.

```
10 + 5
```

```
## [1] 15
```

In R werden Operatoren in folgende Kategorien eingeteilt:

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

### Arithmetic Operators

Operator	Name	Example
<code>+</code>	Addition	<code>x + y</code>
<code>-</code>	Subtraction	<code>x - y</code>
<code>*</code>	Multiplication	<code>x * y</code>
<code>^</code>	Exponent	<code>x^y</code>
<code>%%</code>	Modulus	<code>x %% y</code>
<code>%/%</code>	Integer Division	<code>x %/% y</code>

### Assignment Operators

Zuweisungs Operatoren werden verwendet um einer Variabel einen Wert zuzuweisen.

```
x <- 3
```

```
y <- "Hello"
```

## Comparison Operators

Vergleichs Operatoren werden verwendet um Werte miteinander zu vergleichen.

Operator	Name	Example
==	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

Logische Operatoren werden verwendet um zwei konditionale statements miteinander zu verbinden.

Operator	Description
&	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

Operator	Description	Example
:	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

### Conditions and If Statements

Operator	Name	Example
==	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

### The if statement

Der Code innerhalb eines if wird nur dann ausgeführt wenn der wert der if bedingung == TRUE ist

```
a <- 33
b <- 200

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

```
## [1] "b is greater than a"
```

### Else if

Wird verwendet falls die Bedingungen der vorherigen if nicht TRUE waren.

```
a <- 33
b <- 33

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

```
## [1] "a and b are equal"
```

### If else

Wird ausgeführt falls keine der Bedingungen der vorherigen if TRUE waren. Ein else kann auch ohne ein if else verwendet werden es muss jedoch ein if geben.

```
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")
}
```

```
## [1] "a is greater than b"
```

### Nested if statements

Wenn ein if in einem anderen if steht ist das ein 'nested if'

```
x <- 41

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

```
print("below 10.")
}
```

```
## [1] "Above ten"
## [1] "and also above 20!"
```

## AND OR

### AND

Das & Symbol wird verwendet um Bedingungen miteinander zu verbinden. Die Bedingung ist nur dann TRUE wenn beide alle Bedingungen TRUE sind.

```
a <- 200
b <- 33
c <- 500

if (a > b & c > a) {
  print("Both conditions are true")
}
```

```
## [1] "Both conditions are true"
```

### OR

Das | Symbol wird verwendet um Bedingungen zu verbinden. Die Bedingung ist nur dann TRUE wenn numindest eine der Bedingungen TRUE ist.

```
a <- 200
b <- 33
c <- 500

if (a > b | a > c) {
  print("At least one of the conditions is true")
}
```

```
## [1] "At least one of the conditions is true"
```

## While loop

### Loops

Eine Schleife führt einen Code so lange aus wie die Bedingung TRUE ist. Es gibt zwei Arten von Schleifen:

- Die while loop
- Die for loop

### While loops

Der code einer while Schleifen wird solange ausgeführt wie die Bedingung TRUE ist.

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

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



```
## [1] 3
## [1] 4
## [1] 5
```

## Break

Ein `break` kann verwendet werden um aus einer Schleife auszubrechen.

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

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

## Next

Mit `next` kann eine Iteration der Schleife übersprungen werden ohne die Schleife abbrechen zu müssen.

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

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

## For loop

### For loops

Eine `for` Schleife wird verwendet um über eine Sequenz zu itterieren.

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

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

```
## [1] 9
## [1] 10
```

Um jeden Teil einer Liste zu printen kann eine `for` Schleife verwendet werden:

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

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

```
## [1] "apple"
## [1] "banana"
## [1] "cherry"
```

Das gleiche ist auch mit Zahlen möglich:

```
dice <- c(1, 2, 3, 4, 5, 6)

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

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

## Break

Das `break` funktioniert wie in einer `while` loop.

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

for (x in fruits) {
  if (x == "cherry") {
    break
  }
  print(x)
}
```

```
## [1] "apple"
## [1] "banana"
```

## Next

Das `next` funktioniert wie in einer `while` loop.

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

for (x in fruits) {
  if (x == "banana") {
    next
  }
  print(x)
}
```

```
## [1] "apple"
## [1] "cherry"
```

## Nested loops

Es ist auch möglich eine Schleife in einer anderen Schleife auszuführen.

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

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

```
## [1] "red apple"
## [1] "red banana"
## [1] "red cherry"
## [1] "big apple"
## [1] "big banana"
## [1] "big cherry"
## [1] "tasty apple"
## [1] "tasty banana"
## [1] "tasty cherry"
```

## Functions

Eine function ist ein Stück an Code welches nur ausgeführt wird wenn es aufgerufen wird.

### Creating functions

Um eine Funktion zu erstellen muss das `function()` keyword verwendet werden.

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

### Call a function

Um eine Funktion aufzurufen muss der name der Funktion gefolgt von Klammern geschrieben werden:  
`function()`

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

my_function()
```

```
## [1] "Hello World!"
```

## Arguments

Informationen können Funktionen als Argumente übergeben werden. Argumente werden in den Klammern nach `function` angegeben. Es können so viele hinzugefügt werden wie man will, sie müssen nur mit einem `,` getrennt werden. Ein Argument ist der Wert welcher der Funktion gesendet wird. Ein Parameter ist die Variable welche in den Klammern der Funktion steht.

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

```
my_function("Peter")
```

```
## [1] "Peter Griffin"
```

```
my_function("Lois")
```

```
## [1] "Lois Griffin"
```

```
my_function("Stewie")
```

```
## [1] "Stewie Griffin"
```

### Number of arguments

Eine Funktion muss in der Regel mit der richtigen Anzahl an Argumenten aufgerufen werden ansonsten wird ein Error geworfen.

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

```
my_function("Peter", "Griffin")
```

```
## [1] "Peter Griffin"
```

### Default parameter value

Ein default Parameter Wert kann verwendet werden damit kein Error geworfen wird falls kein oder zu wenige Argumente gegeben werden. Der default Parameter wird nur verwendet falls kein Wert übergeben wird.

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

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

```
## [1] "I am from Sweden"
```

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

```
## [1] "I am from India"
```

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

```
## [1] "I am from Norway"
```

```
my_function("USA")
```

```
## [1] "I am from USA"
```

### Return values

Damit eine Funktion einen Wert zurück gibt wird `return()` genutzt.

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

```
}

print(my_function(3))
```

```
## [1] 15

print(my_function(5))
```

```
## [1] 25

print(my_function(9))
```

```
## [1] 45
```

## Nested functions

Es gibt zwei Arten von nested functions: 1. Wenn eine Funktion in einer anderen Funktion aufgerufen wird 2. Wenn eine Funktion in einer anderen Funktion geschrieben wird.

Bsp. für 1. Fall:

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

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

```
## [1] 10
```

Bsp.: für den 2. Fall:

```
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)
```

```
## [1] 8
```

## Recursion

R unterstützt auch rekursive Aufrufe von Funktionen. Also eine Methode kann sich selbst aufrufen. Es muss aufgepasst werden dass eine Funktion die rekursiv funktioniert auch irgendwann aufhört.

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

tri_recursion(6)
```

```
## [1] 1
## [1] 3
## [1] 6
## [1] 10
## [1] 15
## [1] 21
```

## Global variables

Variablen welche ausserhalb einer Funktion erstellt werden können global verwendet werden.

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

my_function()
```

```
## [1] "R is awesome"
```

Wenn eine Variabel mit dem selben Namen innerhalb einer Funktion erstellt wird, wird die lokale Variabel verwendet.

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

my_function()
```

```
## [1] "R is fantastic"
```

```
txt # print txt
```

```
## [1] "global variable"
```

## The Global Assignment Operator

Um eine globale Variabel in einer Funktion zu erstellen kann der <<- Operator verwendet werden.

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

my_function()
```

```
## [1] "R is fantastic"
```

```
print(txt)
```

```
## [1] "fantastic"
```

<<- wird auch verwendet um den Wert einer globalen Variabel in einer Methode zu verändern

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

```

}

my_function()

## [1] "R is fantastic"
paste("R is", txt)

## [1] "R is fantastic"

```

## R Data Structures

### Vectors

Ein Vector ist eine Liste in der alle Elemente den gleichen Typen haben. Um eine Liste in einen Vector zu verwandeln wird die `c()` Funktion verwendet.

```

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

# Print fruits
fruits

```

```
## [1] "banana" "apple" "orange"
```

Es kann auch eine Sequenz in einen Vector umgewandelt werden.

```

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

numbers

```

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

### Vector Length

Die `length()` Funktion wird verwendet um die Länge eines Vektors zu bekommen.

```

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

length(fruits)

```

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

### Sort a Vector

Um einen Vector zu sortieren kann die `sort()` Funktion verwendet werden.

```

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

sort(fruits) # Sort a string

```

```
## [1] "apple" "banana" "lemon" "mango" "orange"
```

```
sort(numbers) # Sort numbers
```

```
## [1] 2 3 5 7 13 20
```

## Access Vectors

Ein Element eines Vectors kann mittels seinem Index innerhalb von '[]' erhalten werden.

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

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

```
## [1] "banana"
```

Es können auch mehrere Elemente auf einmal erhalten werden.

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

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

```
## [1] "banana" "orange"
```

Ein negativer Index gibt alle Elemente bis auf das des Indexes zurück.

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

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

```
## [1] "apple" "orange" "mango" "lemon"
```

## Change an Item

Um einen Wert eines Vectors zu ändern wird ihr Index verwendet.

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

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

```
# Print fruits  
fruits
```

```
## [1] "pear" "apple" "orange" "mango" "lemon"
```

## Repeat Vectors

Um einen Vector wiederholend zu machen kann die rep() Funktion verwendet werden.

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

```
repeat_each
```

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

## Generating sequenced vectors

Ein Vector kann mit einer Sequenz verwendet werden.

```
numbers <- 1:10
```

```
numbers
```

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



Damit größere oder kleinere Schritte in einer Sequenz machen zu können muss `seq()` verwendet werden.

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

```
numbers
```

```
## [1] 0 20 40 60 80 100
```

## Lists

Eine Liste ist eine Sammlung an Daten, welche verschiedene Typen haben können und sortiert sind.

```
# List of strings
```

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

```
# Print the list
```

```
thislist
```

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

```
## [1] "apple"
```

```
##
```

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

```
## [1] "banana"
```

```
##
```

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

```
## [1] "cherry"
```

## Access Lists

Auf eine Liste wird über ihren Index zugegriffen.

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

```
thislist[1]
```

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

```
## [1] "apple"
```

## Change Item Value

Ein Element einer Liste kann über den Index verändert werden.

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

```
thislist[1] <- "blackcurrant"
```

```
# Print the updated list
```

```
thislist
```

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

```
## [1] "blackcurrant"
```

```
##
```

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

```
## [1] "banana"
```

```
##
```

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

```
## [1] "cherry"
```

## List Length

Um die Länge einer Liste zu erhalten kann die `length()` methode verwendet werden.

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

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

## Check if Item Exists

Um zu wissen ob ein Element in einer Liste existiert kann der `%in%` Operator verwendet werden.

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

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

## Add List Items

Um ein Element an eine Liste hinzuzufügen wir die `append()` Methode verwendet.

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

```
## [[1]]  
## [1] "apple"  
##  
## [[2]]  
## [1] "banana"  
##  
## [[3]]  
## [1] "cherry"  
##  
## [[4]]  
## [1] "orange"
```

Um ein Element an einer bestimmten Stelle einzufügen wird der `after = n` als parameter übergeben.

```
thislist <- list("apple", "banana", "cherry")  
  
append(thislist, "orange", after = 2)
```

```
## [[1]]  
## [1] "apple"  
##  
## [[2]]  
## [1] "banana"  
##  
## [[3]]  
## [1] "orange"  
##  
## [[4]]  
## [1] "cherry"
```

## Remove List Items

Um das erste Element aus einer Liste zu entfernen wird der index [-1] angegeben.

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

newlist <- thislist[-1]

# Print the new list
newlist

## [[1]]
## [1] "banana"
##
## [[2]]
## [1] "cherry"
```

## Range of Indexes

Es kann auch eine Reihe an Indexen angegeben werden.

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

(thislist)[2:5]

## [[1]]
## [1] "banana"
##
## [[2]]
## [1] "cherry"
##
## [[3]]
## [1] "orange"
##
## [[4]]
## [1] "kiwi"
```

## Loop Through a List

Mit einer for loop kann durch eine Lite durch Itteriert werden.

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

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

## [1] "apple"
## [1] "banana"
## [1] "cherry"
```

## Join Two Lists

Listen können verbunden werden indem die c() Methode verwendet wird.

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

```
list3
```

```
## [[1]]  
## [1] "a"  
##  
## [[2]]  
## [1] "b"  
##  
## [[3]]  
## [1] "c"  
##  
## [[4]]  
## [1] 1  
##  
## [[5]]  
## [1] 2  
##  
## [[6]]  
## [1] 3
```

## Matrices

Eine Matrix ist ein 2 Dmensionales Datenset mit columns und rows. Eine Matrix kann mit der `matrix()` Funktion erstellt werden.

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

## Access Matrix Items

Ein Element einer Matrix kann mit dem Index des columns und der row erhalten werden.

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
  
thismatrix[1, 2]  
  
## [1] "cherry"
```

Es kann auch eine ganze row erhalten werden indem nur ein Index angegeben wird.

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

Das gleiche ist auch mit einem column möglich.

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

```
## [1] "cherry" "orange"
```

### Access More Than One Row

Mit der `c()` Funktion können mehrere rows erhalten werden.

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)
thismatrix[c(1,2),]
```

```
##      [,1]      [,2]      [,3]
## [1,] "apple"  "orange" "pear"
## [2,] "banana" "grape"  "melon"
```

### Access More Than One Column

Das gleiche ist auch mit mehreren columns möglich.

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)
thismatrix[, c(1,2)]
```

```
##      [,1]      [,2]
## [1,] "apple"  "orange"
## [2,] "banana" "grape"
## [3,] "cherry" "pineapple"
```

### Add Rows and Columns

Um columns in eine Matrix hinzuzufügen kann die `cbind()` Funktion verwendet werden.

```
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
```

```
##      [,1]      [,2]      [,3]      [,4]
## [1,] "apple"  "orange"  "pear"  "strawberry"
## [2,] "banana" "grape"   "melon" "blueberry"
## [3,] "cherry" "pineapple" "fig"   "raspberry"
```

Um rows hinzuzufügen wird die `rbind()` Funktion verwendet.

```
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
```

```
##      [,1]      [,2]      [,3]
## [1,] "apple"  "orange"  "pear"
## [2,] "banana" "grape"   "melon"
```

```
## [3,] "cherry"      "pineapple" "fig"
## [4,] "strawberry" "blueberry" "raspberry"
```

### Remove Rows and Columns

Mit der `c()` Funktion können rows und columns entfernt werden.

```
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
```

```
## [1] "mango"      "pineapple"
```

### Check if an Item Exists

Um herauszufinden ob ein Element in einer Matrix enthalten ist wird der `%in%` Operator verwendet.

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

"apple" %in% thismatrix

## [1] TRUE
```

### Number of Rows and Columns

Um die Anzahl an rows und columns zu erhalten wird die `dim()` Funktion verwendet.

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

dim(thismatrix)

## [1] 2 2
```

### Matrix Length

Um die Dimension einer Matrix herauszufinden wird die `length()` Funktion verwendet.

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

length(thismatrix)

## [1] 4
```

### Loop Through a Matrix

Um über eine Matrix zu iterieren wird die `for` loop verwendet.

```
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])
  }
}

## [1] "apple"
## [1] "cherry"
```

```
## [1] "banana"
## [1] "orange"
```

## Combine two Matrices

Die `rbind()` und `cbind()` Funktionen können verwendet werden um Matrixen miteinander zu verbinden.

```
# 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
```

```
##      [,1]      [,2]
## [1,] "apple"  "cherry"
## [2,] "banana" "grape"
## [3,] "orange" "pineapple"
## [4,] "mango"  "watermelon"
```

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

```
##      [,1]      [,2]      [,3]      [,4]
## [1,] "apple"  "cherry" "orange" "pineapple"
## [2,] "banana" "grape"  "mango"  "watermelon"
```

## Arrays

Arrays Funktionieren wie Matrixen nur haben sie mehr Dimensionen. Um einen Array zu erstellen kann die `array()` Funktion verwendet werden.

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

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

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

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   13   17   21
## [2,]   14   18   22
## [3,]   15   19   23
```

```
## [4,] 16 20 24
```

### Access Array Items

Ein Element eines Arrays kann mit dem Index erhalten werden.

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

multiarray[2, 3, 2]
```

```
## [1] 22
```

Es ist auch möglich ein row oder column zu erhalten.

```
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]
```

```
## [1] 1 5 9
```

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

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

### Check if an Item Exists

Um herauszufinden ob ein Array ein Element enthält kann der `%in%` Operator verwendet werden.

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

2 %in% multiarray
```

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

### Amount of Rows and Columns

Um die Anzahl an rows und columns zu erhalten wird die `dim()` Funktion verwendet.

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

dim(multiarray)
```

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

### Array Length

Um die Dimension eines Arrays zu erhalten wird die `length()` Funktion verwendet.

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

length(multiarray)
```



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

## Loop Through an Array

Es kann über einen Array iteriert werden indem eine for loop verwendet wird.

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

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

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
```

## Data Frames

[https://methodenlehre.github.io/einfuehrung-in-R/chapters/03-data\\_frames.html](https://methodenlehre.github.io/einfuehrung-in-R/chapters/03-data_frames.html)

Ein Data Frame sind Daten die als Tabellen dargestellt werden. Sie können character, numbers, ... beinhalten solange sie nicht in einem column gemischt sind.

```
# 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
```

```
##   Training Pulse Duration
## 1 Strength   100       60
```

```
## 2 Stamina 150 30
## 3 Other 120 45
```

## Summarize the Data

Um die Daten zusammenzufassen wird die `summary()` Funktion verwendet.

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

Data\_Frame

```
## Training Pulse Duration
## 1 Strength 100 60
## 2 Stamina 150 30
## 3 Other 120 45
```

```
summary(Data_Frame)
```

```
## Training      Pulse      Duration
## Length:3      Min.   :100.0   Min.   :30.0
## Class :character 1st Qu.:110.0   1st Qu.:37.5
## Mode :character Median :120.0   Median :45.0
##              Mean  :123.3   Mean  :45.0
##              3rd Qu.:135.0   3rd Qu.:52.5
##              Max.   :150.0   Max.   :60.0
```

## Access Items

Es können `[]`, `[][]` oder `$` verwendet werden um columns von einem Data Frame zu erhalten.

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

Data\_Frame[1]

```
## Training
## 1 Strength
## 2 Stamina
## 3 Other
```

```
Data_Frame[["Training"]]
```

```
## [1] "Strength" "Stamina" "Other"
```

```
Data_Frame$Training
```

```
## [1] "Strength" "Stamina" "Other"
```

## Add Rows

Um eine row hinzuzufügen wird die `rbind()` Funktion verwendet.

```

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

##   Training Pulse Duration
## 1 Strength   100      60
## 2  Stamina   150      30
## 3   Other   120      45
## 4 Strength   110     110

```

## Add Columns

Um ein column hinzuzufügen wird die `cbind()` Funktion verwendet.

```

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

##   Training Pulse Duration Steps
## 1 Strength   100      60  1000
## 2  Stamina   150      30  6000
## 3   Other   120      45  2000

```

## Remove Rows and Columns

Um columns und rows zu entfernen wird die `c()` Funktion verwendet.

```

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

##   Pulse Duration

```

```
## 2    150      30
## 3    120      45
```

### Amount of Rows and Columns

Um die Anzahl an rows und columns zu erhalten wird die `dim()` Funktion verwendet.

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

dim(Data_Frame)
```

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

Es kann auch die `ncol()` und `nrow()` Funktion verwendet werden um die Anzahl an columns und die Anzahl an rows zu erhalten.

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

ncol(Data_Frame)
```

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

```
nrow(Data_Frame)
```

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

### Data Frame Length

Um die Anzahl an columns zu erhalten wird die `length()` Funktion verwendet.

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

length(Data_Frame)
```

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

### Combining Data Frames

Um Zwei oder mehr Data Frames vertikal zu verbinden wird die `rbind()` Funktion verwendet.

```
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

```

```

##   Training Pulse Duration
## 1 Strength   100       60
## 2  Stamina   150       30
## 3   Other   120       45
## 4  Stamina   140       30
## 5  Stamina   150       30
## 6 Strength   160       20

```

Um Zwei oder mehr Data Frames horizontal zu verbinden wird die `cbind()` Funktion verwendet.

```

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

```

```

##   Training Pulse Duration Steps Calories
## 1 Strength   100       60  3000       300
## 2  Stamina   150       30  6000       400
## 3   Other   120       45  2000       300

```

## Factors

Factors werden verwendet um Daten zu Kategorisieren. Um einen Factor zu erstellen wird die `factor()` Funktion verwendet.

```

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

# Print the factor
music_genre

```

```

## [1] Jazz   Rock   Classic Classic Pop    Jazz   Rock   Jazz
## Levels: Classic Jazz Pop Rock

```

Um nur die Levels zu printen wird die `levels()` Funktion verwendet.

```

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

levels(music_genre)

```

```
## [1] "Classic" "Jazz"      "Pop"      "Rock"
```

Die Levels können auch gesetzt werden indem `levels` in der `factor()` Funktion verwendet wird.

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

```
## [1] "Classic" "Jazz"      "Pop"      "Rock"     "Other"
```

## Factor Length

Um herauszufinden wie viele Elemente in einem Factor sind wird die `length()` Funktion verwendet.

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

```
## [1] 8
```

## Access Factors

Um auf ein Element eines Factors zuzugreifen wird der Index verwendet.

```
music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "Jazz"))  
music_genre[3]
```

```
## [1] Classic  
## Levels: Classic Jazz Pop Rock
```

## Change Item Value

Um den Wert eines Elementes zu ändern wird der Index verwendet.

```
music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "Jazz"))  
music_genre[3] <- "Pop"  
music_genre[3]
```

```
## [1] Pop  
## Levels: Classic Jazz Pop Rock
```

Falls ein Element in einen Factor hinzugefügt werden soll, dass noch nicht beinhaltet/definiert ist tritt ein Fehler auf.

```
music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "Jazz"))  
music_genre[3] <- "Opera"
```

```
## Warning in `[<-factor`(`*tmp*`, 3, value = "Opera"): invalid factor level, NA  
## generated  
music_genre[3]
```

```
## [1] <NA>  
## Levels: Classic Jazz Pop Rock
```

Wenn das Element aber schon im `levels` definiert ist funktioniert es.

```
music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "Jazz"), levels = c("Jazz", "Rock", "Classic", "Pop", "Opera"))
music_genre[3] <- "Opera"
music_genre[3]
```

```
## [1] Opera
## Levels: Classic Jazz Pop Rock Opera
```

## Numerische Funktionen

abs(x)	Betrag
sqrt(x)	Quadratwurzel
ceiling(x)	Aufrunden: ceiling(3.475) ist 4
floor(x)	Abrunden: floor(3.475) ist 3
round(x, digits = n)	Runden: round(3.475, digits = 2) ist 3.48
log(x)	Natürlicher Logarithmus
log(x, base = n)	Logarithmus zur Basis n
log2(x)	Logarithmus zur Basis 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. probs: Vektor mit Wahrscheinlichkeiten
sum(x)	Summe
min(x)	Minimalwert x_min
max(x)	Maximalwert x_max
range(x)	x_min und x_max
scale(x, center = TRUE, scale = TRUE)	Zentrieren und Standardisieren
sample(x, size, replace = FALSE, prob)	Ziehen mit/ohne Zurücklegen.
prob:	Vektor mit Wahrscheinlichkeiten