EinfuehrungR

Christof Zlabinger

2023-09-25

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 soballt 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
```

Concatenate Elements

Zwei Variabeln können kombiniert werden mittels paste() Beispiel:

```
text \<- "Christof"

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

Es ist auch möglich zwei Variabeln miteinander zu kombinieren.

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

text2 \<- "Christof"

paste(text1, text2)
```

Für Zahlen kann ein '+' verwendet werden

```
num1 \<- 5

num2 \<- 5

num1 + num2
```

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

```
num \<- 5` `text \<- "Hello"

num + text

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

Multiple Variables

Es ist möglich einen Wert meherern Variabeln zu zuweisen

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

Variable Names

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

Data Types

Data Types

Variabeln 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 üvergebenen Variabel

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 Dezimazahlstelle 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 kovertiert werden.

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

Math

Simple Math

Es können einfache matthematische Rechnungen mittel opperatoren durchgeführt werden.

- `+` Um Zahlen zu addiren
- `-` Im 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)` rudet die Zahl auf die nächst höchste ganze Zahl auf `floor(x)` rundet die Zahl auf die nächt 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()` verwender,

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 Charactere 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
10 == 9   # FALSE
10 < 9    # FALSE
```

Dies ist auch mit Variabeln möglich.

```
a <- 10
b <- 9

a > b
```

Operators

Operators

Operatoren werden verwendet um operationen durchzuführen.

Zum Beispiel können zwei Zahlen mittels eines + addiert werden.

```
10 + 5
```

In R werden Operatoren in volgende Kategorien eingeteilt:

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

Arithmetic Operators

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

Assignment Operators

Zusweisungs 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 verwender 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 wir nur dann ausgeführt wenn der wert der if bedingung == `TRUE` ist

```
a <- 33
b <- 200

if (b > a) {
  print("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")
}
```

If else

Wird ausgeführt falls keine der Bedinungen 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")
}
```

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("below 10.")
}
```

AND OR

AND

Das `&` Symbol wird verwended 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")
}
```

OR

Das `|` Symbol wird verwended um Bedinungen zu verbinden. Die Bedinung ist nur dann `TRUE` wenn numindest eine der Bedinugungen `TRUE` ist.

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

```r
if (a > b | a > c) {
  print("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 Bedinung `TRUE` ist.

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

Break

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

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

Next

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

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

For loop

For loops

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

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

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

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

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

Das gleiche ist auch mit Zahlen möglich:

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

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

Break

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

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

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

Next

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

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

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

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))
  }
}
```

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()
```

Arguments

Informationen können Funktionen als Arguemnt 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 Variabel welche in den Klammern der Funktion steht.

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

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

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")
```

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)
}
```

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

Return values

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

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

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

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:

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

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

Bsp.: für den 2. Fall:

```r
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

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

```r
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

Variabeln welche auserhalb einer Funktion erstellt werden können global verwendet werden.

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

my_function()
```

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()

txt # print txt
```

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()

print(txt)
```

`<<-` 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()

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

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
```

Es kann auch eine Sequenz in einen Vector umgewandelt werden.

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

numbers
```

Vector Length

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

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

length(fruits)
```

Sort a Vector

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

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

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

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]
```

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)]
```

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)]
```

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
```

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
```

Generating sequenced vectors

Ein Vector kann mit einer Sequenz verwendet werden.

```
numbers <- 1:10

numbers
```

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

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

numbers
```

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
```

Access Lists

Auf eine Liste wird über ihren Index zugegriffen.

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

thislist[1]
```

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
```

List Length

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

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

length(thislist)
```

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
```

Add List Items

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

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

append(thislist, "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)
```

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
```

Range of Indexes

Es kann auch eine Reihe an Indexen angegeben werden.

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

(thislist)[2:5]
```

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)
}
```