R Vorkurs

Teil 2

Martin Arnold & Jens Klenke

08.04.2021

Übersicht

1. Logische Operatoren
2. Dataframes
3. Listen
4. Bedingte Anweisungen
5. Schleifen
6. Funktionen
7. R Pakete

Logische Operatoren

Einführung

Die Klasse `logical` habt ihr im ersten Teil schon mal gesehen (`is.na()`)

Logische Vergleiche werden für das Programmieren bedingter Anweisungen benötigt. Sie kommen aber auch häufig bei Schleifen und beim Subsetten von Daten zum Einsatz.

Das Ergebnis logischer Vergleiche ist immer ein *boolscher Wert*, also `TRUE` / `FALSE` bzw. `T` / `F`.

Die wichtigsten Operatoren im Überblick:

```
==  # "ist gleich"
!=  # "ist ungleich"
<   # "ist kleiner"
<=  # "ist kleiner oder gleich"
>   # "ist größer"
>=  # "ist größer gleich"
&   # "logisches 'Und'"
|   # "logisches 'Oder'"
!   # einen boolschen Wert "negieren"
```

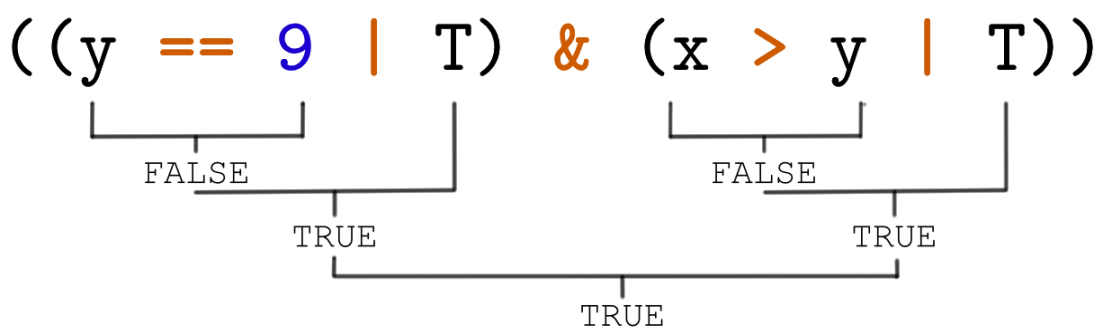
Logische Operatoren

Beispiele

Hier ein paar Beispiele:

```
x <- 5
y <- 10

x == y
x > y | y == 10
x > y | x == 10
!(x == y | y < x)
((y == 9 | T) & (x > y | T))
!T & !F
```



Logische Operatoren

Übungsaufgaben

1. Sind die folgenden Ausdrücke `TRUE` oder `FALSE`?

- $5 \geq 5$
- $5 > 5$
- $T = 5$
- $T \wedge F \ \vee \ F \wedge T$
- $F \wedge F \wedge F \ \vee \ T$
- $(\neg(5 > 3) \vee A = B)$
- $\neg(((T > F) > T) \wedge \ \neg T$

Logische Operatoren

Übungsaufgaben

2. Es sei `z <- c(1, 2, NA, 4)`. Überprüfen Sie die folgenden Aussagen mittels einer Logikabfrage in R.

- Die Länge des Vektors z ist ungleich 2.
- Die Länge der logischen Überprüfungen, ob die einzenlen Elementen gleich 2 sind, ist 4.
- Der Vektor z hat die Klasse `numeric`.
- Einige Elemente des Vektors z sind `NA`.
- Das Zweite Element des Vektors z ist `numeric`.
- Das Minimum und das Maximum sind ungleich.

Logische Operatoren

Übungsaufgaben

3. Es sei `M <- matrix(1:9, ncol = 3)`. Was ergeben folgende Ausdrücke?

- sum((M[, 1]) == 6)
- max((M[, 2]) <= 5)
- (M[2, 2] != 4 & M[2, 2] > 6)

Dataframes

Über Dataframes

Ein dataframe ist eine Sammlung von Variablen, ähnlich einer Matrix.

Am Beispiel des Datensatzes `iris`:

```
iris[1:10, ]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
```

Dataframes

Über Dataframes

Die Funktion `str()` liefert Informationen über die Struktur eines Objekts.

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Wir sehen:

- Das Objekt `iris` hat die Klasse `data.frame`
- `iris` hat 150 Zeilen (Beobachtungen) und 5 Spalten (Variablen).
- Vier der Variablen gehören zur Klasse `numeric`, 1 Variable zur Klasse `Factor`

Dataframes

Erstellen von Dataframes

Ein dataframe wird mit der Funktion `data.frame()` erstellt. Hierzu übergeben wir einfach Vektoren, welche als Spalten gruppiert werden sollen. Die Spalten können benannt werden.

Anders als bei Matrizen müssen die einzelnen Spalten (wie oben gesehen) nicht derselben Klasse angehören!

```
df <- data.frame(Letters = c("A", "B", "C", "D"),
                 Numbers = 1:4,
                 Logicals = c(T, F, FALSE, TRUE),
                 z)
df
```

```
##   Letters Numbers Logicals  z
## 1       A       1     TRUE  1
## 2       B       2    FALSE  2
## 3       C       3    FALSE NA
## 4       D       4     TRUE  4
```

Dataframes

Zugriff auf Spalten und Elemente

Auf die einzelnen Zellen in einem dataframe kann man wie bei Matrizen durch Indexierung mit `[ ]` zugreifen. Zugriff auf einzelne Variablen/Spalten erhält man mit `$`:

```
df[ , 1]       # Spalte 1
```

```
## [1] "A" "B" "C" "D"
```

```
df[1, ]        # Zeile 1
```

```
##   Letters Numbers Logicals z
## 1       A       1     TRUE 1
```

```
df[1, 1]       # Zeile 1, Spalte 1
```

```
## [1] "A"
```

```
df[ , 1:2]     # Spalten 1 und 2
```

```
##   Letters Numbers
## 1       A       1
## 2       B       2
## 3       C       3
## 4       D       4
```

Dataframes

Zugriff auf Spalten und Elemente

```
df$Numbers     # Spalte/Variable "Numbers"
```

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

Subsetting:

```
df[df$Letters == "B", ]
```

```
##   Letters Numbers Logicals z
## 2       B       2    FALSE 2
```

```
df[df$Numbers > 2, ]
```

```
##   Letters Numbers Logicals  z
## 3       C       3    FALSE NA
## 4       D       4     TRUE  4
```

Dataframes

Bearbeiten von Dataframes

Ein dataframe ist nach seiner Erstellung nicht unveränderbar. Man kann Spalten und Zeilen hinzufügen oder entfernen. Das gilt auch für einzelne Beobachtungen.

```
df$Greeks <- c("alpha", "beta", "gamma", "delta")   # Hinzufügen der Spalte "Greeks"
df        <- df[-2, ]                               # Entfernen der zweiten Zeile
df[2, 2]  <- NA                                     # Beobachtung an Stelle 3x2 auf NA setzen
df
```

```
##   Letters Numbers Logicals  z Greeks
## 1       A       1     TRUE  1  alpha
## 3       C      NA    FALSE NA  gamma
## 4       D       4     TRUE  4  delta
```

Dataframes

Übungsaufgaben

4. Verschaffen Sie sich einen Überblick über den Datensatz `mtcars` (dieser ist in base R bereits geladen)
    - Aus wie vielen Variablen besteht er? Welche Klasse haben die einzelnen Variablen?
5. Lassen Sie sich folgende Subsets von `mtcars` ausgeben:
    - nur die Variable `mpg`
    - nur die ersten drei Zeilen
    - nur die ersten drei Variablen
    - nur die ersten beiden Beobachtungen der Variablen `cyl` und `hp`
    - alle Beobachtungen deren Ausprägung der Variablen `hp` größer ist als 200

Dataframes

Übungsaufgaben

6. Erstellen Sie einen dataframe $persons$ mit den Variablen `Name` (`character`), `Height` (cm, `numeric`) und `Weight` (kg, `numeric`) von 5 fiktiven Personen.
    - Lassen Sie sich das Körpergewicht der 3. Person anzeigen.
    - Lassen Sie sich nun die Körpergröße aller Personen anzeigen.
    - Fügen Sie die Variable "Augenfarbe" hinzu. Die Ausprägungen sollten vom Typ `character` sein. Schauen Sie sich den veränderten dataframe an.

Listen

Listen erzeugen

Listen werden mit der Funktion `list()` erzeugt. Ein **Vorteil** von Listen ist, dass die einzelnen Elemente von unterschiedlicher Größe und Typ sein können.

Der Zugriff auf Listenelemente erfolgt ebenfalls mit `$`.

```
my.list <- list(A = 1:5, B = mtcars, C = list(letters, LETTERS))
```

Viele Funktionen in R geben Ergebnisse als Listen zurück.

```
# Regressionsmodell
model <- lm(mpg ~ hp, data = mtcars)
str(model)
model$coefficients
model$residuals
```

(Mehr dazu in Teil 3)

Bedingte Anweisungen

If-Anweisung

Bedingte Anweisungen helfen uns den Programmablauf zu steuern. Die einfachste Anweisung ist die `if`-Anweisung:

```
if (x == 5) print("Hallo Welt!")
```

Dies kann man lesen wie eine natürliche Sprache:

FALLS `x` gleich 5 ist, *DANN* gebe "Hallo Welt!" aus.

Die Bedingung muss immer zu `TRUE` oder `FALSE` evaluieren. Nur wenn der Ausdruck in den Klammern nach *if* wahr ist, wird der nachfolgende Code ausgeführt.

If-Anweisung

Wenn der auszuführende Code länger als eine Zeile ist, nutzt man *geschweifte Klammern* um einen Code-Block zu erzeugen:

```
if (x > 0) {
  # Code
  # ...
}
```

Ein paar Beispiele:

```
if ( class(x) == "numeric" ) print("x ist eine Zahl!")
# FALLS die Klasse von x "numeric" ist, DANN gebe "x ist eine Zahl!" aus.

if (2 * x >= y) print("Die Hälfte von y ist x!")
# FALLS 2*x größer oder gleich y ist, DANN gebe "Die Hälfte von y ist x!" aus.

z  <- 1:10
op <- "add"

if (length(z) > 1 & op == "add") {
  sum(z)
}
# FALLS die Länge des Vektors z größer als 1 ist UND op GLEICH "add" ist,
# DANN  summiere die Elemente von z.
```

Bedingte Anweisungen

If-Else Anweisung

Wenn der Ausdruck innerhalb der Klammern nicht `TRUE` ist, wird der Codeblock nicht ausgeführt.

Aber was wäre wenn wir in diesem Fall einen anderen Code ausführen wollen?

Lösung: Die `If-Else`-Anweisung

```
if (expr) {     # Wenn "expr" TRUE ist, ...
  # BLOCK 1     # dann führe BLOCK 1 aus.
} else {        # Wenn "expr" FALSE ist, ...
  # BLOCK 2     # dann führe BLOCK 2 aus.
}
```

Bedingte Anweisungen

If-Else Anweisung

Auch hier Beispiele:

```
if (is.numeric(x) & x >= 0) {
  x^(-0.5)
} else {
  print("x ist keine Zahl oder negativ!")
}

if(length(z) > 0 & op == "add") {
  sum(z)
} else if(length(z) > 0 & op == "mult") {
  prod(z)
} else {
  print("z ist kein Vektor!")
}
```

Bedingte Anweisunge

If-Else(-If) Anweisung

Wie im letzten Beispiel gesehen kann man beliebig viele Bedingungen mit `If` und `Else` verknüpfen.

```
if (expression) {
  # ...
} else if(expression) {
  # ...
} else if(expression) {
  # ...
} else {
  # ...
}
```

Bedingte Anweisunge

Übungsaufgaben

7. Schreibe Code, der die Wurzel ($sqrt()$) eines Vektors x der Länge 1 berechnet, wenn der Wert in x nicht negativ ist.
8. Erstelle Code, welcher die Wurzel der Elemente eines Vektors x berechnet, wenn alle Werte in x nicht negativ sind.
    - *Hinweis:* nutze eine Funktion wie $\min()$ oder $\text{sum}()$
9. Schreibe Code, der die Struktur ($\text{str}()$) eines Objekts df wiedergibt, sofern df zur Klasse data.frame gehört. Andernfalls soll die Länge des Objekts wiedergegeben werden.
    - *Überprüft eure Codes in dem ihre verschiedene Werte für* x *bzw.* df *ausprobiert!*

Schleifen

for-Schleife

Es gibt drei Schleifentypen in R: `for`, `while` (und `repeat`)

Die `for`-Schleife hat folgenden Aufbau:

```
for(var in enumeration) {
  # Schleifenkörper
}
```

Für jeden Wert in `enumeration` wird der Schleifenkörper einmal ausgeführt. Bei jedem Durchgang ist der aktuelle Wert aus `enumeration` in `var` zwischengespeichert.

```
for(i in 1:5) {
  cat("Number: ", i, " ", "\n")
}
```

Schleifen

while-Schleife

Die `while`-Schleife kann genutzt werden, wenn nicht klar ist, wie oft ein Codeabschnitt ausgeführt werden soll:

```
while(condition) {
  # Schleifenkörper
}
```

Solange die `condition` wahr ist, wird der Schleifenkörper immer wieder ausgeführt. Vor dem ersten und nach jedem weiteren Durchlauf wird die `condition` erneut evaluiert.

```
x <- 0
while(x < 4) {
  x <- runif(n = 1, min = 1, max = 5)
  cat(x, " ", "\n")
}
```

Schleifen

Übungsaufgaben

10. Schreiben Sie eine Schleife, welche die Zahlen von 1 bis 15 aufaddiert.
11. Erstellen Sie eine Matrix M mit folgender Gestalt:

$$M = \begin{pmatrix} 1 & 4 & 7 & 10 & 13 \\ 2 & 5 & 8 & 11 & 14 \\ 3 & 6 & 9 & 12 & 15 \end{pmatrix}$$

`

    - Schreiben Sie eine Schleife, welche für jede Spalte die Spaltensumme berechnet und ausgibt.

12. Mit $\mathrm{rnorm}(1)$ ziehen wir eine Zufallszahl aus der Standardnormalverteilung (in der Konsole ausprobieren!). Schreiben Sie eine Schleife, welche solange ausgeführt wird, bis ein Wert gezogen wird, der größer als 1 ist. Geben Sie in jedem Durchlauf die gezogene Zahl mit $cat(x, \text{" } \backslash\mathrm{n} \text{ "})$ aus. (Hinweis: $\backslash\mathrm{n}$ steht für einen Zeilenumbruch)

Funktionen

Funktionen definieren

Viele Funktionen habt ihr schon kennengelernt: `length()`, `sum()`, `min()`, `data.frame()`

Eigene Funktionen werden wie folgt geschrieben:

```
name.der.funktion <- function(arg1, arg2, ...) {
  # Funktionskörper
  return(obj)
}
```

Beispiel: Summe zweier Objekte

```
summe <- function(x, y) {
  return(x + y)
}
# Nachdem die Funktion definiert (und entsprechend ausgeführt) wurde, kann man die Funktion direkt nutzen:
summe(x = 1, y = 3)
```

```
## [1] 4
```

Funktionen

Standardwerte der Argumente

In der Definition einer Funktion können auch Standardwerte der Argumente festgelegt werden. Dadurch kommt es zu keiner Fehlermeldung wegen fehlender Argumente.

```
summe <- function(x = 1, y = 3) {
  return(x + y)
}
summe()
```

```
## [1] 4
```

Environments

Beachtet, dass alle Objekte, welche innerhalb einer Funktion definiert wurden, außerhalb dieser Funktion nicht verfügbar sind! Außer man gibt deren Inhalt zurück:

```
internal.ops <- function() {
  int.x <- 5
  int.y <- 10
}

internal.ops() # int.x un int.y sind nicht in Environment Fenster zu sehen
int.x          # Fehler!
```

Funktionen

Übungsaufgaben

13. Die Dichte der Standardnormalverteilung lautet

`

$$\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

`

   - Schreiben Sie eine Funktion `stdnv`, welche die Dichte von `x` berechnet und zurückgibt.
     - **Hinweis:** `?exp`, `?pi`
     - **Hinweis:** Wenn die Funktion korrekt ist, sollten `stdnv(x)` und `dnorm(x)` die gleichen Ergebnisse liefern.

Funktionen

Übungsaufgaben

14. Schreiben Sie eine Funktion, welche die Argumente `z` sowie `opt` erwartet. Im Funktionskörper soll mit einer If-Anweisung gesteuert werden, welche Operation auf `z` ausgeführt werden soll:
    - *WENN opt gleich "add" ist, DANN addiere die Elemente von `z`, WENN `opt` gleich "mult" ist, dann multipliziere die Elemente von `z`, andernfalls führe keine Operation aus.*
    - Am Ende soll die Funktion das jeweilige Ergebnis wiedergeben.
15. Schreiben Sie eine Funktion, die den MSE (mean squared error) von zwei Vektoren `y` und `yhat` (die Argumente) berechnet. Der MSE is definiert als $\frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$. Testen Sie Ihre Funktion anhand der beiden Vektoren $y = 2, 4, 2, 5, 7$ und $\hat{y} = 2.3, 3.5, 2.1, 5.5, 7.6$ (das Ergebnis sollte 0.192 lauten).

R Pakete

Pakete allgemein

Die Erweiterbarkeit von R ist eine seiner Stärken. Jeder kann eigene Pakete entwickeln und sie den anderen Usern zur Verfügung stellen.

CRAN (Comprehensive R Archive Network) ist ein verteiltes Archiv, in dem Pakete gesammelt und der breiten Öffentlichkeit zugängig gemacht werden.

Wer sich schon etwas mit R auskennt und sich für die Entwicklung von Paketen interessiert, dem ist das Buch R Packages von Hadley Wickham zu empfehlen.

R Pakete

Pakete installieren updaten und entfernen

Das wichtigste (und standardmäßige) *Repository* ist der CRAN Server von RStudio.

Weitere CRAN Server:

- https://cloud.r-project.org/
- https://cran.uni-muenster.de/

Dort findet man die neuesten, stabilen Versionen von Paketen. Pakete auf CRAN müssen zudem bestimmte Anforderungen erfüllen.

RStudio stellt alles zur Verfügung um die eigene Paket-Bibliothek zu verwalten.

Sollte man Pakete doch "per Hand" managen gibt es die Funktionen `install.packages()`, `update.packages()` und `remove.packages()`.

```
install.packages("quantmod")        # ein Paket mit Methoden zur Analyse von Finanzdaten
update.packages()
remove.packages(pkgs = "quantmod")
```