Lua Dokumentation für StdInOut

VulpesSoft / Manfred Fuchs
*Manfred Fuchs**
16.03.2016
Version 1

Dieses Dokument beschreibt die Funktionen, Konstanten und Variablen der **Lua** API für das Programm **StdInOut**.
Zudem enthält es eine Zusammenfassung der Syntax, Typen, Operatoren und allgemeinen Bibliotheksfunktionen von **Lua**.

Eine detailliertere Ausführung der Programmiersprache **Lua** finden Sie unter folgenden Adressen:

http://www.lua.org	*Lua Website*
http://www.lua.org/docs.html	*Lua Dokumentation*
http://lua.coders-online.net	*Lua Doku. auf Deutsch (für Version 5.1)*

* utilities@vulpessoft.de

Inhaltsverzeichnis

1 Die Syntax der Lua Programmiersprache 8

1.1 Reservierte Schlüsselwörter und Kommentare 8

1.1.1 and 8

1.1.2 break 8

1.1.3 do 8

1.1.4 else 8

1.1.5 elseif 8

1.1.6 end 9

1.1.7 false 9

1.1.8 for 9

1.1.9 function 9

1.1.10 if 9

1.1.11 in 9

1.1.12 local 10

1.1.13 nil 10

1.1.14 not 10

1.1.15 or 10

1.1.16 repeat 10

1.1.17 return 10

1.1.18 then 10

1.1.19 true 10

1.1.20 until 10

1.1.21 while 11

1.1.22 – 11

1.1.23 –[=[...]=] 11

1.1.24 _X 11

1.1.25 #! 11

1.2 Typen (die Stringwerte sind die möglichen Ergebnisse der Basisbibliotheksfunktion type()) 12

1.2.1 “nil“ 12

1.2.2 “boolean“ 12

1.2.3 “number“ 12

1.2.4 “string“ 12

1.2.5 “table“ 12

1.2.6 “function“ 12

1.2.7 “thread“ 13

1.2.8 “userdata“ 13
1.3 Strings und Escape Sequenzen 14
1.3.1 ’...’ und “...“ 14
1.3.2 [=[...]=] 14
1.3.3 \a 14
1.3.4 \b 14
1.3.5 \f 14
1.3.6 \n 14
1.3.7 \r 14
1.3.8 \t 14
1.3.9 \v 14
1.3.10 \\ 14
1.3.11 \“ 15
1.3.12 \’ 15
1.3.13 \[. 15
1.3.14 \] 15
1.3.15 \ddd 15
1.4 Operatoren (absteigende Priorität) 16
1.4.1 ^ 16
1.4.2 # 16
1.4.3 - (negativ) 16
1.4.4 * 16
1.4.5 / 16
1.4.6 % 16
1.4.7 + ... 17
1.4.8 - 17
1.4.9 17
1.4.10 < 17
1.4.11 > 17
1.4.12 <= 17
1.4.13 >= 18
1.4.14 ~= 18
1.4.15 == 18
1.4.16 and 18
1.4.17 or 18
1.4.18 not 18
1.5 Zuweisungen 19

1.5.1 a = 5, b = 'hi' . . . 19
1.5.2 local a = a . . . 19
1.5.3 a, b, c = 1, 2, 3 . . . 19
1.5.4 a, b = b, a . . . 19
1.5.5 a, b = 4, 5, '6' . . . 19
1.5.6 a, b = 'there' . . . 19
1.5.7 a = nil . . . 19
1.5.8 a = z . . . 19
1.5.9 a = '3' + '2' . . . 20
1.5.10 a = 3 .. 2 . . . 20
1.6 Kontrollstrukturen . . . 21
1.6.1 do *block* end . . . 21
1.6.2 if *exp* then *block* {elseif *exp* then *block*} {else *block*} end . . . 21
1.6.3 while *exp* do *block* end . . . 21
1.6.4 repeat *block* until *exp* . . . 21
1.6.5 for *var = start, end* [, *step*] do *block* end . . . 21
1.6.6 for *vars* in *iterator* do *block* end . . . 21
1.6.7 break . . . 21
1.7 Tabellenkonstruktoren . . . 22
1.7.1 t = {} . . . 22
1.7.2 t = {'yes', 'no', '?'} . . . 22
1.7.3 t = {[1] = 'yes', [2] = 'no', [3] = '?'} . . . 22
1.7.4 t = {[-900] = 3, [900] = 4} . . . 22
1.7.5 t = {x = 5, y = 10} . . . 22
1.7.6 t = {x = 5, y = 10, 'yes', 'no'} . . . 22
1.7.7 t = {msg = 'choice', {'yes', 'no', '?'}} . . . 22
1.8 Funktionsdefinition . . . 23
1.8.1 function *name* (*args*) *body* [return *values*] end . . . 23
1.8.2 local function *name* (*args*) *body* [return *values*] end 23
1.8.3 f = function (*args*) *body* [return *values*] end . . . 23
1.8.4 function ([*args,*] ...) *body* [return *values*] end . . . 23
1.8.5 function *t.name* (*args*) *body* [return *values*] end . 23
1.8.6 function *obj:name* (*args*) *body* [return *values*] end 23
1.9 Funktionsaufruf . . . 24
1.9.1 f(x) . . . 24
1.9.2 f 'hello' . . . 24
1.9.3 f 'goodby' . . . 24

1.9.4 f [[see you soon]] . . . 24
1.9.5 f {x = 3, y = 4} . . . 24
1.9.6 t.f(x) . . . 24
1.9.7 x:move(2, -3) . . . 24
1.10 Metatabellen Prozeduren . . . 25
1.11 Metatabellen Felder (für Tabellen und Userdaten) . . . 25
1.12 Muster . . . 26
1.12.1 Zeichenklasse . . . 26
1.12.2 Musterelement . . . 27
1.12.3 Muster . . . 27
1.12.4 Treffer . . . 27

2 Bibliotheken 29

2.1 Die StdInOut Bibliothek [SIO] . . . 29
2.1.1 Funktionen . . . 29
2.1.1.1 CharCount() . . . 29
2.1.1.2 CreateBak() . . . 29
2.1.1.3 ForEachChar() . . . 29
2.1.1.4 ForEachLine() . . . 30
2.1.1.5 GetChar() . . . 30
2.1.1.6 GetLine() . . . 30
2.1.1.7 InPlace() . . . 30
2.1.1.8 InputFile() . . . 31
2.1.1.9 LineCount() . . . 31
2.1.1.10 OutputFile() . . . 31
2.1.1.11 ScriptFile() . . . 31
2.1.1.12 SubDirs() . . . 31
2.2 Die Basisbibliothek [kein Prefix] . . . 32
2.2.1 Variablen . . . 32
2.2.1.1 _VERSION . . . 32
2.2.2 Funktionen . . . 32
2.3 Die I/O Bibliothek [io] . . . 33
2.3.1 Variablen . . . 33
2.3.2 Funktionen . . . 33
2.4 Die Mathematik Bibliothek [math] . . . 34
2.4.1 Variablen . . . 34
2.4.2 Funktionen . . . 34
2.5 Die Betriebssystem Bibliothek [os] . . . 35

2.5.1 Variablen . . . 35
2.5.2 Funktionen . . . 35
2.6 Die Pfad Bibliothek [os.path] . . . 36
2.6.1 Variablen . . . 36
2.6.2 Funktionen . . . 36
2.7 Die String Bibliothek [string] . . . 37
2.7.1 Funktionen . . . 37
2.7.1.1 byte() . . . 37
2.7.1.2 char() . . . 37
2.7.1.3 find() . . . 37
2.7.1.4 format() . . . 38
2.7.1.5 len() . . . 38
2.7.1.6 lower() . . . 38
2.7.1.7 rep() . . . 39
2.7.1.8 reverse() . . . 39
2.7.1.9 split() . . . 39
2.7.1.10 splitfirst() . . . 39
2.7.1.11 splitlast() . . . 40
2.7.1.12 sub() . . . 40
2.7.1.13 trim() . . . 40
2.7.1.14 trimleft() . . . 40
2.7.1.15 trimright() . . . 41
2.7.1.16 upper() . . . 41
2.8 Die Tabellen Bibliothek [table] . . . 42
2.8.1 Funktionen . . . 42
2.9 Die UTF-8 Bibliothek [utf8] . . . 43
2.9.1 Variablen . . . 43
2.9.2 Funktionen . . . 43
2.10 Die Zip Bibliothek [zip] . . . 44
2.10.1 Funktionen . . . 44
2.10.1.1 create() . . . 44
2.10.1.2 zip() . . . 44
2.10.2 Funktionen des Zip Objektes . . . 44
2.10.2.1 add() . . . 44
2.10.2.2 close() . . . 44
2.10.2.3 extract() . . . 45
2.10.2.4 extractall() . . . 45

2.10.2.5 filecount() 45
2.10.2.6 fileinfo() 45
2.10.2.7 filename() 46
2.10.2.8 filenames() 46
2.10.2.9 getcomment() 46
2.10.2.10 getfilecomment() 46
2.10.2.11 indexof() 47
2.10.2.12 open() 47
2.10.2.13 setcomment() 47
2.10.2.14 setfilecomment() 47

1 Die Syntax der Lua Programmiersprache

1.1 Reservierte Schlüsselwörter und Kommentare

1.1.1 and

and

1.1.2 break

break

Die **break**-Anweisung wird benutzt, um die Ausführung einer **while**-, **repeat**- oder **for**-Schleife zu beenden und zur nächsten Anweisung nach der Schleife zu springen. Ein **break** beendet die innerste verschachtelte Schleife.
Die **break**-Anweisung kann nur als letzte Anweisung eines Blocks geschrieben werden.

1.1.3 do

do

Beginn eines Blocks. Ein Block ist eine Liste von Anweisungen und kann explizit begrenzt werden, um eine einzelne Anweisung zu erzeugen. Explizite Blöcke sind nützlich, um den Gültigkeitsbereich von Variablen zu steuern. Explizite Blöcke werden auch manchmal benutzt, um eine **return**- oder **break**-Anweisung innerhalb eines anderen Blocks hinzuzufügen.

1.1.4 else

else

Mit **else** kann in einer **if**-Anweisung ein optionaler Anweisungsblock definiert werden. Dieser wird ausgeführt wenn die Bedingung von **if** nicht zutrifft.

1.1.5 elseif

elseif

Mit **elseif** kann in einer **if**-Anweisung ein optionaler Anweisungsblock definiert werden. Dieser wird ausgeführt wenn die Bedingung von **if** nicht zutrifft und beinhaltet zugleich eine weitere Abfrage zur bedingten Ausführung.

Da es in Lua keine switch-Anweisung gibt, dient **elseif** als syntaktische Erleichterung um verschachtelte **if**-Anweisungen zu vereinfachen.

1.1.6 end

`end`

Definiert das Ende eines Blocks welcher mit **do** oder **then** eingeleitet wurde.

1.1.7 false

`false`

false repräsentiert den logischen Ausdruck für Falsch.

1.1.8 for

`for`

Die **for**-Anweisung hat zwei Formen: Eine numerische und eine generische. Die numerische **for**-Schleife wiederholt einen Code-Block, während eine Laufvariable eine arithmetische Folge durchläuft. Sie hat folgende Syntax:

```
for var = from, to [, step] do block end
```

block wird für *var* beginnend beim Wert von *from* wiederholt, bis es mit der Schrittweite von *step to* erreicht.
Alle drei Ausdrücke zur Steuerung werden lediglich eimnal ausgewertet, bevor die Schleife beginnt. Sie müssen alle in einer Zahl resultieren. Falls *step* (die Schrittweite) nicht angegeben wird, wird eine Schrittweite von 1 benutzt.

Die generische **for**-Anweisung arbeitet mit Funktionen, genannt Iteratoren. Nach jeder Iteration wird der Iterator aufgerufen, um einen neuen Wert zu erzeugen, bis der neue Wert **nil** ist. Die generische **for**-Schleife hat folgende Syntax:

```
for var [, var [, ...]] in iterator do block end
```

Sie können **break** benutzen, um eine **for**-Schleife zu verlassen. Die Schleifenvariablen *var* sind lokal für die Schleife; Sie können deren Wert nach Beendigung oder Abbruch von **for** nicht verwenden. Falls Sie diesen Wert benötigen, weisen Sie ihn vor dem Abbruch oder der Beendigung der Schleife einer anderen Variable zu.

1.1.9 function

`function`

1.1.10 if

`if`

Die **if**-Anweisung dient zur bedingten Ausführung eines Anweisungsblocks. Wenn die Bedingung **true** (wahr) oder einen gültigen Wert enthält, dann wird der Anweisungsblock ausgeführt. Wenn die Bedingung **false** (falsch) oder **nil**(nicht existent) enthält, dann wird der Anweisungsblock nicht ausgeführt. Bei der **if**-Anweisung wird der Block nicht mit dem sonst üblichen **do** sondern mit **then** eingeleitet.

1.1.11 in

`in`

1.1.12 local

`local`

Variablen sind per Definition global. Mit **local** kann eine lokale Variable definiert werden welche dann nur innerhalb des aktuellen Blocks gültig ist. Lokale Variablen können überall innerhalb eines Blocks deklariert werden. Die Deklaration kann eine initiale Zuweisung beinhalten.

1.1.13 nil

`nil`

1.1.14 not

`not`

1.1.15 or

`or`

1.1.16 repeat

`repeat`

Die **repeat**-Anweisung repräsentiert eine Schleife mit Austrittsbedingung. Die Schleife läuft solange, wie die Bedingung nach dem Schlüsselwort **until** wahr ist. Sie können **break** benutzen, um eine **repeat**-Schleife zu verlassen.

1.1.17 return

`return`

Die **return**-Anweisung wird benutzt, um Werte aus einer Funktion zurückzugeben. Funktionen können mehr als einen Wert zurückgeben.
Die **return**-Anweisung kann nur als letzte Anweisung eines Blocks geschrieben werden.

1.1.18 then

`then`

Beginn eines Blocks im Falle einer **if**-Anweisung.

1.1.19 true

`true`

true repräsentiert den logischen Ausdruck für Wahr.

1.1.20 until

`until`

Die **until**-Anweisung leitet die Abfrage der Austrittsbedingung einer **repeat**-Schleife ein.

1.1.21 while

`while`

Die **while**-Anweisung repräsentiert eine Schleife mit Eintrittsbedingung. Die Schleife läuft solange, wie die Bedingung nach dem Schlüsselwort **while** wahr ist. Sie können **break** benutzen, um eine **while**-Schleife zu verlassen.

1.1.22 – ...

`-- ...`

Kommentar *(einzeilig)*, alles ab hier bis zum Ende der Zeile wird vom Interpreter ignoriert.

1.1.23 –[=[...]=]

`--[=[...]=]`

Kommentar *(mehrzeilig)*, alles innerhalb wird vom Interpreter ignoriert. Null oder auch mehrfache '=' sind gültig.

1.1.24 _X

`_X`

Alle Wörter in Großbuchstaben mit führendem Unterstrich sind für Konstanten reserviert.

1.1.25 #!

`#!`

Übliches Unix shebang Symbol. Lua ignoriert die komplette erste Zeile wenn sie damit beginnt.

1.2 Typen (die Stringwerte sind die möglichen Ergebnisse der Basisbibliotheksfunktion type())

1.2.1 “nil“

“nil“

Nil ist der Typ des Wertes **nil**, dessen Haupteigenschaft es ist, verschieden von allen anderen Werten zu sein; für gewöhnlich zeigt er die Abwesenheit eines sinnvollen Wertes an.

1.2.2 “boolean“

“boolean“

Boolean ist der Typ der Werte **false** und **true**. Sowohl **nil** als auch **false** machen eine Bedingung falsch; jeder andere Wert macht sie wahr.

1.2.3 “number“

“number“

Number repräsentiert reelle (doppelt genaue Gleitpunkt-)Zahlen.

1.2.4 “string“

“string“

String repräsentiert ein Feld von Zeichen. Lua benutzt 8 Bit: Zeichenketten können beliebige 8-bit-Zeichen enthalten, inklusive Nullen (’\0’).

1.2.5 “table“

“table“

Der Typ table implementiert assoziative Felder; dies sind Felder, welche nicht nur mit Zahlen, sondern mit beliebigen Werten (außer **nil**) indiziert werden können. Tabellen können heterogen sein, d. h. diese können Werte aller Typen (außer **nil**) enthalten. Tabellen sind der grundlegende Mechanismus in Lua für Datenstrukturen; diese können benutzt werden, um gewöhnliche Felder, Symboltabellen, Mengen, Strukturen, Graphen, Bäume etc. zu repräsentieren. Um Strukturen darzustellen benutzt Lua den Feldbezeichner als Index. Die Sprache unterstützt diese Repräsentation durch a.name als eine syntaktische Vereinfachung für a[’name’]. Es gibt einige bequeme Wege, um Tabellen in Lua zu erzeugen.

Genauso wie Indizes können die Werte eines Tabellenfeldes von beliebigem Typ (außer **nil**) sein. Im Besonderen können Tabellenfelder Funktionen enthalten, da diese Werte erster Ordnung sind. Derartige Tabellen können ebenso Methoden beinhalten.

1.2.6 “function“

“function“

Dieser Datentyp repräsentiert eine Funktion (entweder des Host-Programmes oder im Script definiert).

1.2.7 “thread“

“thread“

Der Typ thread repräsentiert unabhängige Threads zur Ausführung und wird zur Implementierung von Koroutinen benutzt. Verwechseln Sie nicht Lua-Threads mit denen des Betriebssystems. Lua unterstützt unter allen Systemen Koroutinen; auch auf Systemen, die keine Threads unterstützen.

1.2.8 “userdata“

“userdata“

Der Typ userdata wird angeboten, um beliebige Daten in Lua-Variablen zu speichern. Dieser Typ entspricht einem Block rohen Speichers und hat außer der Zuweisung und einem Gleichheitstest keine vordefinierten Operationen in Lua. Durch die Verwendung von Metatabellen kann der Programmierer jedoch Operationen für “userdata“-Werte definieren.
“Userdata“-Werte können in Lua nicht erstellt oder bearbeitet werden, sondern nur über das Host-Programm.

1.3 Strings und Escape Sequenzen

1.3.1 '...' und "..."

'...' und "..."

Textbegrenzer. Escapesequenzen werden berücksichtigt.

1.3.2 [=[...]=]

[=[...]=]

Mehrzeiliger Text. Escapesequenzen werden ignoriert.

1.3.3 \a

\a

Klingel.

1.3.4 \b

\b

Backspace.

1.3.5 \f

\f

Formfeed.

1.3.6 \n

\n

Newline.

1.3.7 \r

\r

Return.

1.3.8 \t

\t

Horizontaler Tabulator.

1.3.9 \v

\v

Vertikaler Tabulator.

1.3.10 \\

****

Backslash.

1.3.11 \“

\“

Doppeltes Anführungszeichen.

1.3.12 \’

\’

Einfaches Anführungszeichen.

1.3.13 \[

\[

Eckige Klammer auf.

1.3.14 \]

\]

Eckige Klammer zu.

1.3.15 \ddd

\ddd

Zeichen mit Dezimalcode *ddd*.

1.4 Operatoren (absteigende Priorität)

1.4.1 ^

^

Exponential-Operator.

1.4.2

#

Der Längenoperator wird als unärer Operator # geschrieben. Die Länge einer Zeichenkette entspricht ihrer Anzahl an Bytes (dies ist die gewöhnliche Bedeutung der Länge einer Zeichenkette, wenn jedes Zeichen ein Byte belegt).

Die Länge einer Tabelle t ist als ganzzahliger Index n definiert, so dass $t[n]$ nicht **nil** ist und $t[n+1]$ **nil** ist; darüber hinaus kann n 0 sein, wenn $t[1]$ **nil** ist. Bei einem regulären Feld mit nicht-nil Werten von 1 bis zu einem gegebenen n ist dessen Länge eben dieses n - der Index des letzten Wertes. Falls das Feld "Löcher" hat (d. h. **nil**-Werte zwischen anderen nicht-nil Werten), kann $\#t$ irgendeiner der Indizes sein, welcher direkt einem **nil**-Wert vorangeht (d. h. er nimmt einen solchen **nil**-Wert als Ende des Feldes an).

1.4.3 - (negativ)

- (negativ)

Negiert eine Zahl.

1.4.4 *

*

Arithmetischer Operator welcher zwei Zahlen multipliziert. Wenn einer oder beide Operanden Zeichenketten sind, so werden sie, sofern möglich, zu Zahlen konvertiert.

1.4.5 /

/

Arithmetischer Operator welcher die rechte Zahl von der linken dividiert. Wenn einer oder beide Operanden Zeichenketten sind, so werden sie, sofern möglich, zu Zahlen konvertiert.

1.4.6 %

%

Der Modulo-Operator liefert den Rest einer ganzzahligen Division. Wenn einer oder beide Operanden Zeichenketten sind, so werden sie, sofern möglich, zu Zahlen konvertiert.

1.4.7 +

+

Arithmetischer Operator welcher zwei Zahlen addiert. Wenn einer oder beide Operanden Zeichenketten sind, so werden sie, sofern möglich, zu Zahlen konvertiert.

1.4.8 -

-

Arithmetischer Operator welcher die rechte Zahl von der linken subtrahiert. Wenn einer oder beide Operanden Zeichenketten sind, so werden sie, sofern möglich, zu Zahlen konvertiert.

1.4.9 ..

..

Verbindet zwei Zeichenketten (**string**) miteinander (Konkatenierung). Wenn einer oder beide Operanden Zahlen sind, so werden diese zu Zeichenketten konvertiert.

1.4.10 <

<

Relationaler Operator. Die Bedingung ist **true** (wahr) wenn der linke Operand kleiner als der rechte ist, ansonsten liefert der Vergleich **false** (falsch).
Wenn beide Argumente Zahlen sind, werden sie als solche verglichen. Andernfalls, wenn beide Argumente Zeichenketten sind, werden deren Werte entsprechend der aktuellen Sprache verglichen.

1.4.11 >

>

Relationaler Operator. Die Bedingung ist **true** (wahr) wenn der linke Operand größer als der rechte ist, ansonsten liefert der Vergleich **false** (falsch).
Wenn beide Argumente Zahlen sind, werden sie als solche verglichen. Andernfalls, wenn beide Argumente Zeichenketten sind, werden deren Werte entsprechend der aktuellen Sprache verglichen.

1.4.12 <=

<=

Relationaler Operator. Die Bedingung ist **true** (wahr) wenn der linke Operand kleiner oder gleich dem rechten ist, ansonsten liefert der Vergleich **false** (falsch).
Wenn beide Argumente Zahlen sind, werden sie als solche verglichen. Andernfalls, wenn beide Argumente Zeichenketten sind, werden deren Werte entsprechend der aktuellen Sprache verglichen.

1.4.13 >=

>=

Relationaler Operator. Die Bedingung ist **true** (wahr) wenn der linke Operand größer oder gleich dem rechten ist, ansonsten liefert der Vergleich **false** (falsch).
Wenn beide Argumente Zahlen sind, werden sie als solche verglichen. Andernfalls, wenn beide Argumente Zeichenketten sind, werden deren Werte entsprechend der aktuellen Sprache verglichen.

1.4.14 ~=

~=

Relationaler Operator, negation von ==.

1.4.15 ==

==

Relationaler Operator. Es werden zuerst die Typen der Operanden verglichen. Falls die Typen unterschiedlich sind, ist das Ergebnis **false** (falsch). Andernfalls werden die Werte der Operanden verglichen. Zahlen und Zeichenketten werden auf gewöhnliche Weise verglichen. Objekte (Tabellen, Benutzerdaten, Threads und Funktionen) werden per Referenz verglichen: Zwei Objekte werden nur als gleich angenommen, wenn sie das selbe Objekt sind. Jedes mal, wenn Sie ein neues Objekt (eine Tabelle, Benutzerdaten, einen Thread oder eine Funktion) erzeugen, ist dieses Objekt von bisher existierenden Objekten verschieden.

1.4.16 and

and

Logischer Operator. Der Konjunktionsoperator **and** gibt sein erstes Argument zurück, wenn dieser **false** oder **nil** ist; andernfalls gibt **and** sein zweites Argument zurück. Es wird eine Kurzschlussauswertung verwendet, was bedeutet, dass das zweite Argument nur falls nötig ausgewertet wird. Dabei werden **false** sowie **nil** als falsch und alles andere als wahr betrachtet.

1.4.17 or

or

Logischer Operator. Der Disjunktionsoperator **or** gibt sein erstes Argument zurück, wenn dieser von **nil** und **false** verschieden ist; andernfalls gibt **or** sein zweites Argument zurück. Es wird eine Kurzschlussauswertung verwendet, was bedeutet, dass das zweite Argument nur falls nötig ausgewertet wird. Dabei werden **false** sowie **nil** als falsch und alles andere als wahr betrachtet.

1.4.18 not

not

Logischer Operator welcher das Ergebnis des Ausdrucks negiert. Wie die Kontrollstrukturen werden **false** sowie **nil** als falsch und alles andere als wahr betrachtet.

1.5 Zuweisungen

1.5.1 a = 5, b = 'hi'

```
a = 5, b = 'hi'
```

Einfache Zuweisung. Variablen besitzen keinen Typ und können unterschiedliche Typen aufnehmen.

1.5.2 local a = a

```
local a = a
```

Einfache Zuweisung. Lokale Variablen sind auf ihren Bereich beschränkt, ihr Bereich beginnt nach ihrer Deklarierung.

1.5.3 a, b, c = 1, 2, 3

```
a, b, c = 1, 2, 3
```

Mehrfache Zuweisungen sind möglich.

1.5.4 a, b = b, a

```
a, b = b, a
```

Wertetausch. Die rechte Seite wird komplett ausgewertet bevor die Zuweisung erfolgt.

1.5.5 a, b = 4, 5, '6'

```
a, b = 4, 5, '6'
```

Überzählige Werte auf der rechten Seite ('6') werden ausgewertet aber nicht zugewiesen.

1.5.6 a, b = 'there'

```
a, b = 'there'
```

Für fehlende Werte auf der rechten Seite wird **nil** zugewiesen.

1.5.7 a = nil

```
a = nil
```

Entfernt **a**. Der Inhalt wird demnächst vom **garbage collector** freigegeben, falls nicht anderweitig referenziert.

1.5.8 a = z

```
a = z
```

Wenn **z** nicht definiert ist ist es **nil**. Somit wird **a nil** zugewiesen (und dadurch entfernt).

1.5.9 a = ’3’ + ’2’

```
a = ’3’ + ’2’
```

Zahlen erwartet, die Texte werden in Zahlen umgewandelt (a = 5).

1.5.10 a = 3 .. 2

```
a = 3 .. 2
```

Text erwartet, die Zahlen werden in Texte umgewandelt (a = ’32’).

1.6 Kontrollstrukturen

1.6.1 do *block* end

```
do block end
```

Block. Erstellt einen lokalen Bereich.

1.6.2 if *exp* then *block* {elseif *exp* then *block*} {else *block*} end

```
if exp then block {elseif exp then block} {else block} end
```

Bedingte Ausführung.

1.6.3 while *exp* do *block* end

```
while exp do block end
```

Schleife. Wird ausgeführt solange *exp* wahr ist.

1.6.4 repeat *block* until *exp*

```
repeat block until exp
```

Schleife. Wird ausgeführt bis *exp* wahr ist. *exp* befindet sich im lokalen Bereich.

1.6.5 for *var* = *start*, *end* [, *step*] do *block* end

```
for var = start, end [, step] do block end
```

Zählschleife. *var* ist im lokalen Bereich der Schleife.

1.6.6 for *vars* in *iterator* do *block* end

```
for vars in iterator do block end
```

Iterierende Schleife. *vars* ist im lokalen Bereich der Schleife.

1.6.7 break

```
break
```

Beendet die Schleife. Muss der letzte Befehl in einem Block sein.

1.7 Tabellenkonstruktoren

1.7.1 t = {}

```
t = {}
```

Erstellt eine leere Tabelle und weist sie **t** zu.

1.7.2 t = {'yes', 'no', '?'}

```
t = {'yes', 'no', '?'}
```

Einfaches Array, die Elemente sind **t[1]**, **t[2]**, **t[3]**.

1.7.3 t = {[1] = 'yes', [2] = 'no', [3] = '?'}

```
t = {[1] = 'yes', [2] = 'no', [3] = '?'}
```

Selbiges wie zuvor, aber mit expliziter Indexzuweisung.

1.7.4 t = {[-900] = 3, [900] = 4}

```
t = {[-900] = 3, [900] = 4}
```

Verstreutes Array mit nur zwei Elementen (keine Platzverschwendung).

1.7.5 t = {x = 5, y = 10}

```
t = {x = 5, y = 10}
```

Hashtabelle. Die Felder sind **t['x']**, **t['y']** (oder **t.x**, **t.y**).

1.7.6 t = {x = 5, y = 10, 'yes', 'no'}

```
t = {x = 5, y = 10, 'yes', 'no'}
```

Gemischte Tabelle. Die Felder sind **t['x']**, **t['y']**, **t[1]**, **t[2]**.

1.7.7 t = {msg = 'choice', {'yes', 'no', '?'}}

```
t = {msg = 'choice', {'yes', 'no', '?'}}
```

Tabellen können andere Tabellen als Felder beinhalten.

1.8 Funktionsdefinition

1.8.1 function *name* (*args*) *body* [return *values*] end

```
function name ( args ) body [return values] end
```

Definiert eine Funktion und weist sie der Variable **name** zu.

1.8.2 local function *name* (*args*) *body* [return *values*] end

```
local function name ( args ) body [return values] end
```

Definiert eine Funktion lokal zum aktuellen Abschnitt.

1.8.3 f = function (*args*) *body* [return *values*] end

```
f = function ( args ) body [return values] end
```

Anonyme Funktion welche der Variable **f** zugewiesen wird.

1.8.4 function ([*args,*] ...) *body* [return *values*] end

```
function ( [args, ] ... ) body [return values] end
```

Variable Anzahl an Argumenten. Im Funktionsrumpf *body* wird darauf über **...** zugegriffen.

1.8.5 function *t.name* (*args*) *body* [return *values*] end

```
function t.name ( args ) body [return values] end
```

Kurzversion für *t.name* = **function**.

1.8.6 function *obj:name* (*args*) *body* [return *values*] end

```
function obj:name ( args ) body [return values] end
```

Objektfunktion, liefert *obj* als zusätzliches erstes Argument **self**.

1.9 Funktionsaufruf

1.9.1 f(x)

```
f(x)
```

Einfacher Aufruf, mögliche Rückgabe von einem oder mehreren Werten.

1.9.2 f 'hello'

```
f 'hello'
```

Kurzform für **f('hello')**.

1.9.3 f 'goodby'

```
f 'goodby'
```

Kurzform für **f('goodby')**.

1.9.4 f [[see you soon]]

```
f [[see you soon]]
```

Kurzform für **f([[see you soon]])**.

1.9.5 f {x = 3, y = 4}

```
f {x = 3, y = 4}
```

Kurzform für **f({x = 3, y = 4})**.

1.9.6 t.f(x)

```
t.f(x)
```

Ruft eine Funktion auf welche dem Feld **f** von Tabelle **t** zugewiesen ist.

1.9.7 x:move(2, -3)

```
x:move(2, -3)
```

Objektaufruf: Kurzform für **x.move(x, 2, -3)**.

1.10 Metatabellen Prozeduren

1.11 Metatabellen Felder (für Tabellen und Userdaten)

1.12 Muster

1.12.1 Zeichenklasse

Eine Zeichenklasse repräsentiert eine Menge von Zeichen. Die folgenden Kombinationen sind zur Beschreibung einer Zeichenklasse erlaubt:

- x: (soweit x keines der *magischen Zeichen* ^$()%.[]* +-? ist) repräsentiert das Zeichen x selbst.
- .: (ein Punkt) repräsentiert alle Zeichen.
- %a: repräsentiert alle Buchstaben.
- %c: repräsentiert alle Steuerzeichen.
- %d: repräsentiert alle Ziffern.
- %g: repräsentiert alle druckbaren Zeichen ausgenommen Zwischenraumzeichen (Leerzeichen, Tab, ...).
- %l: repräsentiert alle Kleinbuchstaben.
- %p: repräsentiert alle Interpunktionszeichen.
- %s: repräsentiert alle Zwischenraumzeichen (Leerzeichen, Tab, ...).
- %u: repräsentiert alle Großbuchstaben.
- %w: repräsentiert alle alphanumerischen Zeichen.
- %x: repräsentiert alle hexadezimalen Zeichen.
- %z: repräsentiert das Zeichen der Repräsentation 0.
- %x: (wobei x ein beliebiges nicht-alphanumerische Zeichen ist) repräsentiert das Zeichen x. Dies ist das Standardvorgehen um *magische Zeichen* zu maskieren. Allen Punktionszeichen (auch die nichtmagischen) kann ein '%' vorangestellt werden, um diese sich in einem Muster selbst repräsentieren zu lassen.
- [set]: repräsentiert eine Klasse welche die Vereinigung aller Zeichen der Menge ist. Ein Bereich von Zeichen kann durch Trennung der Endzeichen mit einem '-' spezifiziert werden. Alle oben beschriebenen Klassen %x können auch als Elemente der Menge verwendet werden. Alle anderen Zeichen der Menge repräsentieren sich selbst. Zum Beispiel [%w_] (oder [_%w]) repräsentiert alle alphanumerischen Zeichen inkl. dem Unterstrich, [0-7] repräsentiert die oktalen Ziffern und [0-7%l%-] repräsentiert die oktalen Ziffern inkl. Kleinbuchstaben und dem '-' Zeichen.
  Die Interaktion zwischen Bereichen und Klassen ist nicht festgelegt. Somit haben Muster wie [%a-z] oder [a-%%] keine Bedeutung.
- [^set]: repräsentiert das Komplement von *set*, wobei *set* wie zuvor beschrieben interpretiert wird.

Für alle Klassen welche von einzelnen Buchstaben repräsentiert werden (%a, %c, etc.) steht der entsprechende Großbuchstabe für das Komplement der Klasse. Somit repräsentiert z.B. %S alle Nicht-Zwischenraumzeichen.

Die Definition von Buchstaben, Zwischenraum- und anderen Zeichen ist von den aktuellen Ländereinstellungen abhängig. Insbesonders die Klasse [a-z] muss nicht mit %l identisch sein.

1.12.2 Musterelement

Ein Musterelement kann sein

- eine einzelne Zeichenklasse, welche auf jedes einzelne Zeichen der Klasse passt.
- eine einzelne Zeichenklasse gefolgt von '∗', welche auf 0 oder mehr Wiederholungen der Zeichen aus dieser Klasse passt. Diese Wiederholung passt immer auf die längste mögliche Sequenz.
- eine einzelne Zeichenklasse gefolgt von '+', welche auf 1 oder mehr Wiederholungen der Zeichen aus dieser Klasse passt. Diese Wiederholung passt immer auf die längste mögliche Sequenz.
- eine einzelne Zeichenklasse gefolgt von '-', welche auch auf 0 oder mehr Wiederholungen der Zeichen aus dieser Klasse passt. Im Gegensatz zu '∗' passt diese Wiederholung immer auf die kürzeste mögliche Sequenz.
- eine einzelne Zeichenklasse gefolgt von '?', welche auf 0 oder 1 Vorkommen eines Zeichens dieser Klasse passt.
- %n, für n zwischen 1 und 9; dies passt auf eine Teil-Zeichenkette, welcher gleich der n-ten gefundenen Zeichenkette ist (siehe unten).
- %bxy, wobei x und y zwei verschiedene Zeichen sind; dies passt auf eine Zeichenkette welche mit x startet, mit y endet und x und y ausbalanziert sind. Damit ist gemeint, dass wenn die Zeichenkette von links nach rechts gelesen und für x +1 und für y -1 gezählt wird, das abschließende y das erste y ist bei dem der Zähler 0 erreicht. Beispielsweise entspricht %b() einem Ausdruck mit ausbalanzierten Klammern.
- %f[set], ein Grenzmuster; dies entspricht einer Position in der Zeichenkette bei der das nächste Zeichen in der Zeichenkette auf die Menge passt und das vorherige Zeichen nicht auf die Menge passt. Die Menge *set* wird wie zuvor beschrieben interpretiert.

1.12.3 Muster

Ein Muster ist eine Sequenz von Musterelementen. Ein Zirkumflex '^' zu Beginn des Musters verankert die Übereinstimmung am Beginn der zu untersuchenden Zeichenkette. Ein '$' am Ende des Musters verankert die Übereinstimmung am Ende der zu untersuchenden Zeichenkette. An ander Position haben '^' und '$' keine spezielle Bedeutung und repräsentieren sich selbst.

1.12.4 Treffer

Ein Muster kann in Klammem eingeschlossene Teilmuster beinhalten; diese beschreiben Treffer. Wenn eine Übereinstimmung gefunden wird, wird die Teil-Zeichenkette der Zeichenkette, welche einen Treffer erzeugt, für zukünftige Nutzung gespeichert. Treffer werden entsprechend ihrer linken Klammer nummeriert. Im Muster "(a∗(.)%w(%s∗))" wird beispielsweise der Teil der Zeichenkette, welcher auf "a∗(.)%w(%s∗)" passt, als

erster Treffer gespeichert (und hat somit die Nummer 1); die Zeichen, welche auf "." passen, werden als Nummer 2 abgelegt und der Teil, welcher auf "%s∗" passt, hat die Nummer 3. () ermittelt als Spezialfall die aktuelle Zeichenketten-Position (eine Nummer). Wenn wir beispielsweise das Muster "()aa()" auf die Zeichenkette "flaaap" anwenden, bekommen wir die zwei Treffer 3 und 5. Ein Muster kann keine Nullen enthalten. Benutzen Sie stattdessen %z.

2 Bibliotheken

2.1 Die StdInOut Bibliothek [SIO]

2.1.1 Funktionen

2.1.1.1 CharCount()

```
SIO.CharCount()
```

Liefert die Anzahl der Zeichen in der aktuellen Eingabedatei zurück.

Beispiel:

```
print(SIO.CharCount())   =>   127
```

2.1.1.2 CreateBak()

```
SIO.CreateBak()
```

Liefert **true** wenn das Flag **/b** (Sicherungskopie erstellen) angegeben wurde, ansonsten **false**.

Beispiel:

```
print(SIO.CreateBak())   =>   false
```

2.1.1.3 ForEachChar()

```
SIO.ForEachChar(func, flag)
```

Legt eine Funktion fest, welche für jedes einzelne Zeichen der Eingabedatei aufgerufen werden soll. Der Parameter **func** ist ein String und beinhaltet den Namen der Funktion. Der Parameter **flag** *(optional, Vorgabe: 0)* ist eine Integerzahl welche an die Funktion weitergereicht wird. Konnte der angegebene Funktionsname erfolgreich registriert werden, wird **true** zurückgeliefert, im Fehlerfall **false**.
Eine Funktion mit dem in **func** angegebenen Namen muss im Script vorhanden sein und folgende Syntax besitzen:

```
function charfunc(c, idx, flag, cprev, cnext)
```

c ist das jeweilige aktuelle Zeichen aus der Eingabedatei. **idx** ist die Position des aktuellen Zeichens (beginnend mit 1) in der Eingabedatei. **flag** ist der weitergeleitete Parameter.
cprev ist das Zeichen vor dem aktuellen Zeichen aus der Eingabedatei. **cnext** ist das Zeichen nach dem aktuellen Zeichen aus der Eingabedatei. Die Funktion muss ein Zeichen oder eine Zeichenkette zurückliefern welches anstelle des übergebenen Zeichens in die Ausgabedatei übernommen wird. Wird ein leerer Text oder **nil** zurückgeliefert, so wird das Zeichen gelöscht.

Beispiel:

```
function charfunc(c, idx, flag)
    local newchar = c == 'x' and 'u' or c
    return newchar
end
SIO.ForEachChar('charfunc', 0)   =>   true
```

2.1.1.4 ForEachLine()

SIO.ForEachLine(func, flag)

Legt eine Funktion fest, welche für jede einzelne Zeile der Eingabedatei aufgerufen werden soll. Der Parameter **func** ist ein String und beinhaltet den Namen der Funktion. Der Parameter **flag** *(optional, Vorgabe: 0)* ist eine Integerzahl welche an die Funktion weitergereicht wird. Konnte der angegebene Funktionsname erfolgreich registriert werden, wird **true** zurückgeliefert, im Fehlerfall **false**.
Eine Funktion mit dem in **func** angegebenen Namen muss im Script vorhanden sein und folgende Syntax besitzen:

function(line, idx, flag)

line ist die jeweilige Zeile aus der Eingabedatei. **idx** ist die Position der Zeile (beginnend mit 1) in der Eingabedatei. **flag** ist der weitergeleitete Parameter.
Die Funktion muss einen String zurückliefern welcher anstelle der übergebenen Zeile in die Ausgabedatei übernommen wird.

Beispiel:

```
function linefunc(line, idx, flag)
    local newline = line:gsub('Python', 'Lua')
    return newline
end
SIO.ForEachLine('linefunc', 0)   =>   true
```

2.1.1.5 GetChar()

SIO.GetChar(index)

Liefert das Zeichen an Position **index** *(beginnend mit 1)* aus der aktuellen Eingabedatei zurück.

Beispiel:

```
print(SIO.GetChar(1))   =>   'L'
```

2.1.1.6 GetLine()

SIO.GetLine(index)

Liefert die Zeile an Position **index** *(beginnend mit 1)* aus der aktuellen Eingabedatei zurück.

Beispiel:

```
print(SIO.GetLine(1))   =>   'Zeile 1'
```

2.1.1.7 InPlace()

SIO.InPlace()

Liefert **true** wenn das Flag **/p** (Original überschreiben) angegeben wurde, ansonsten **false**.

Beispiel:

```
print(SIO.InPlace())   =>   false
```

2.1.1.8 InputFile()

SIO.InputFile()

Liefert den Namen (incl. Pfad) der aktuellen Eingabedatei zurück. Geschieht die Eingabe über Standardinput, so wird ein leerer String zurückgeliefert.

Beispiel:

```
print(SIO.InputFile())   =>   "C:\Daten\TestIn.txt"
```

2.1.1.9 LineCount()

SIO.LineCount()

Liefert die Anzahl der Zeilen in der aktuellen Eingabedatei zurück.

Beispiel:

```
print(SIO.LineCount())   =>   5
```

2.1.1.10 OutputFile()

SIO.OutputFile()

Liefert den Namen (incl. Pfad) der aktuellen Ausgabedatei zurück. Geschieht die Ausgabe über Standardoutput, so wird ein leerer String zurückgeliefert.

Beispiel:

```
print(SIO.OutputFile())   =>   "C:\Daten\TestOut.txt"
```

2.1.1.11 ScriptFile()

SIO.ScriptFile()

Liefert den Namen (incl. Pfad) der aktuellen Lua Scriptdatei zurück.

Beispiel:

```
print(SIO.ScriptFile())   =>   "C:\Daten\Test.sio"
```

2.1.1.12 SubDirs()

SIO.SubDirs()

Liefert **true** wenn das Flag **/s** (Unterverzeichnisse durchsuchen) angegeben wurde, ansonsten **false**.

Beispiel:

```
print(SIO.SubDirs())   =>   false
```

2.2 Die Basisbibliothek [kein Prefix]

2.2.1 Variablen

2.2.1.1 _VERSION

_VERSION

Globale Variable welche die Version des verwendeten Lua Interpreters beinhaltet.

Beispiel:

```
print(_VERSION)   =>   Lua 5.3
```

2.2.2 Funktionen

2.3 Die I/O Bibliothek [io]

2.3.1 Variablen

2.3.2 Funktionen

2.4 Die Mathematik Bibliothek [math]

2.4.1 Variablen

2.4.2 Funktionen

2.5 Die Betriebssystem Bibliothek [os]

2.5.1 Variablen

2.5.2 Funktionen

2.6 Die Pfad Bibliothek [os.path]

2.6.1 Variablen

2.6.2 Funktionen

2.7 Die String Bibliothek [string]

2.7.1 Funktionen

2.7.1.1 byte()

string.byte(s [, i [, j]])
s:byte([, i [, j]])

Liefert den numerischen Code des i-ten bis j-ten Zeichens des übergebenen String. Wird nur der Startindex i übergeben, so wird der numerische Code des einen Zeichens an Position i zurückgeliefert. Wird überhaupt kein Index übergeben, so wird der Code des ersten Zeichens des String zurückgeliefert.
Achtung: In Lua beginnt der Index eines String bei 1.

Beispiel:

```
string.byte('ABCDE')   =>   65
string.byte('ABCDE', 1)   =>   65
string.byte('ABCDE', 3, 4)   =>   67 68
s = 'ABCDE'
s:byte(3, 4)   =>   67 68
```

2.7.1.2 char()

string.char(i1, i2, ...)

Erstellt einen String aus den Zeichencodes welche als Argumente übergeben werden.

Beispiel:

```
string.char(65, 66, 67)   =>   'ABC'
string.char()   =>   (leerer String)
```

2.7.1.3 find()

string.find(s, pattern [, index [, plain]])
s:find(pattern [, index [, plain]])

Findet das erste Vorkommen von *pattern* im übergebenen String. Wurde das Suchmuster gefunden, so werden zwei Werte zurückgeliefert welche die Position des ersten und letzten Index des gefundenen String repräsentieren. Konnte keine Übereinstimmung gefunden werden, so wird **nil** zurückgeliefert.
Optional kann mit einem dritten Argument *index* der Startindex der Suche übergeben werden. Das Argument kann auch negativ sein; In diesem Fall beginnt die Zählung am Ende des String.
Das *pattern*-Argument erlaubt eine komplexe Suchmaske. Die Funktion des Maskenvergleichs kann ausgeschaltet werden indem als viertes Argument *plain* **true** übergeben wird.
Achtung: In Lua beginnt der Index eines String bei 1.

Beispiel:

```
string.find('Hello Lua user', 'Lua')   =>   7 9
string.find('Hello Lua user', 'banana')   =>   nil
string.find('Hello Lua user', 'Lua', 1)   =>   7 9
string.find('Hello Lua user', 'Lua', 8)   =>   nil
string.find('Hello Lua user', 'e', -5)   =>   13 13
string.find('Hello Lua user', '%su')   =>   10 11
string.find('Hello Lua user', '%su', 1, true)   =>   nil
```

2.7.1.4 format()

string.format(s, e1, e2, ...)
s:format(e1, e2, ...)

Erstellt einen formatierten String aus dem übergebenen Format und Argumenten. Dies ist ähnlich zur *printf("format", ...)* Anweisung in C. Die zusätzliche Option **%q** schließt den Wert eines Stringarguments in Anführungszeichen ein.

c, d, E, e, f, g, G, i, o, u, X und x erwarten eine Nummer als Argument. q und s erwarten einen String.

Beispiel:

```
string.format('%s %q', 'Hello', 'Lua user!')   =>   Hello
"Lua user!"
string.format('%c%c%c', 76, 117, 97)   =>   Lua
string.format('%e,   %E',   math.pi,   math.pi)         =>
3.141593e+000, 3.141593E+000
string.format('%f, %g', math.pi, math.pi)   =>   3.141593,
3.14159
string.format('%d, %i, %u', -100, -100, -100)   =>   -100,
-100, 4294967196
string.format('%o,  %x,  %X',  -100,  -100,  -100)      =>
37777777634, ffffff9c, FFFFFF9C
```

2.7.1.5 len()

string.len(s)
s:len()

Liefert die Länge des übergebenen String.

Beispiel:

```
string.len('Lua')   =>   3
string.len('')   =>   0
string.len('Lua\000user')   =>   8
```

2.7.1.6 lower()

string.lower(s)
s:lower()

Konvertiert den übergebenen String in Kleinbuchstaben.

Beispiel:

```
string.lower('Hello, Lua user!')   =>   'hello, lua user!'
```

2.7.1.7 rep()

string.rep(s, n)
s:rep(n)

Erstellt einen String aus *n* zusammengehängten Wiederholungen des übergebenen String.

Beispiel:

```
    string.rep('Lua ', 5)   =>   'Lua Lua Lua Lua Lua '
```

2.7.1.8 reverse()

string.reverse(s)
s:reverse()

Kehrt die Reihenfolge der Zeichen im String um.

Beispiel:

```
    string.reverse('lua')   =>   'aul'
```

2.7.1.9 split()

string.split(s, delim)
s:split(delim)

Trennt den String *s* an der Zeichenkette *delim* auf. Wird *delim* nicht angegeben, so wird der String an seinen Leerzeichen aufgetrennt. Wird für *delim* ein leerer String übergeben, so wird der String in seine einzelnen Buchstaben zerlegt.

Beispiel:

```
    string.split('Teil 1:Teil 2:Teil3', ':')    =>    'Teil 1'
'Teil 2' 'Teil 3'
    string.split('Das ist ein Test')   =>   'Das' 'ist' 'ein'
'Test'
    string.split('Ein Test', '')   =>   'E' 'i' 'n' ' ' 'T' 'e'
's' 't'
```

2.7.1.10 splitfirst()

string.splitfirst(s, delim [, ignorecase])
s:splitfirst(delim [, ignorecase])

Trennt den String *s* am ersten Vorkommen von *delim* in zwei Teile. Wird *delim* nicht angegeben, so wird der String am ersten Leerzeichen aufgetrennt. Wird das optionale Argument *ignorecase* angegeben, wird bei **true** die Groß-/Kleinschreibung ignoriert, bei **false** nicht (Vorgabe ist **true**).

Beispiel:

```
    string.splitfirst('Teil 1:Teil 2:Teil3', ':')   =>   'Teil
1' 'Teil 2:Teil 3'
    string.splitfirst('Das ist ein Test')   =>   'Das' 'ist ein
Test'
```

2.7.1.11 splitlast()

string.splitlast(s, delim [, ignorecase])
s:splitlast(delim [, ignorecase])

Trennt den String *s* am letzten Vorkommen von *delim* in zwei Teile. Wird *delim* nicht angegeben, so wird der String am letzten Leerzeichen aufgetrennt. Wird das optionale Argument *ignorecase* angegeben, wird bei **true** die Groß-/Kleinschreibung ignoriert, bei **false** nicht (Vorgabe ist **true**).

Beispiel:

```
    string.splitlast('Teil 1:Teil 2:Teil3', ':')   =>   'Teil
1:Teil 2' 'Teil 3'
    string.splitlast('Das ist ein Test')   =>   'Das ist ein'
'Test'
```

2.7.1.12 sub()

string.sub(s, i [, j])
s:sub(i [, j])

Liefert einen Teilstring des übergebenen String. Der Teilstring startet bei *i*. Wenn das dritte Argument *j* nicht angegeben wird, endet der Teilstring am Ende des String. Wird das dritte Argument angegeben, endet der Teilstring bei *j* (einschließlich).

Beispiel:

```
    string.sub('Hello Lua user', 7)   =>   Lua user
    string.sub('Hello Lua user', 7, 9)   =>   Lua
    string.sub('Hello Lua user', -8)   =>   Lua user
    string.sub('Hello Lua user', -8, 9)   =>   Lua
    string.sub('Hello Lua user', -8, -6)   =>   Lua
```

2.7.1.13 trim()

string.trim(s)
s:trim()

Entfernt Leerzeichen und Steuerzeichen (Tab, LineFeed, ...) am Anfang und Ende des String.

Beispiel:

```
    string.trim(' Lua ')   =>   'Lua'
```

2.7.1.14 trimleft()

string.trimleft(s)
s:trimleft()

Entfernt Leerzeichen und Steuerzeichen (Tab, LineFeed, ...) am Anfang des String.

Beispiel:

```
    string.trimleft(' Lua ')   =>   'Lua '
```

2.7.1.15 trimright()

string.trimright(s)
s:trimright()

Entfernt Leerzeichen und Steuerzeichen (Tab, LineFeed, ...) am Ende des String.

Beispiel:

```
string.trimright(' Lua ')   =>   ' Lua'
```

2.7.1.16 upper()

string.upper(s)
s:upper()

Konvertiert den übergebenen String in Großbuchstaben.

Beispiel:

```
string.upper('Hello, Lua user!')   =>   'HELLO, LUA USER!'
```

2.8 Die Tabellen Bibliothek [table]

2.8.1 Funktionen

2.9 Die UTF-8 Bibliothek [utf8]

2.9.1 Variablen

2.9.2 Funktionen

2.10 Die Zip Bibliothek [zip]

2.10.1 Funktionen

2.10.1.1 create()

zip.create()

Erstellt ein neues Zip Objekt. Dies ist die einzige Funktion dieser Bibliothek. Alle weiteren Aktionen erfolgen mittels des Zip Objektes.

Beispiel:

```
z = zip.create()
```

2.10.1.2 zip()

zip()

Kurzform für **zip.create()**.

2.10.2 Funktionen des Zip Objektes

2.10.2.1 add()

zipobject:add(filename [, internalname [, storeonly]])

Fügt die Datei *filename* zu dem Zip Archiv hinzu. Wird *internalname* angegeben, so wird die Datei im Archiv unter diesem Namen gespeichert. Wird *storeonly* angegeben und ist es **true**, so wird die Datei unkomprimiert im Archiv gespeichert.
War die Aktion erfolgreich, so wird **true** zurückgeliefert. Im Fehlerfall wird **nil** und der Fehlertext zurückgeliefert.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
z:add('D:\\testfile.dat')
z:close()
```

2.10.2.2 close()

zipobject:close()

Schließt das Zip Archiv und vervollständigt dabei alle noch offenen Dateioperationen.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
z:add('D:\\testfile.dat')
z:close()
```

2.10.2.3 extract()

```
zipobject:extract(index [, path [, createsubs]])
zipobject:extract(filename [, path [, createsubs]])
```

Extrahiert die Datei an Position *index* bzw. mit dem Namen *filename* in den Pfad *path* des Dateisystems. Wird *path* nicht angegeben oder ist es ein leerer String, so wird die Datei in das aktuelle Verzeichnis extrahiert. Wird *createsubs* angegeben und ist es **false**, so werden die im Dateinamen enthaltenen Verzeichnisse nicht angelegt.
War die Aktion erfolgreich, so wird **true** zurückgeliefert. Im Fehlerfall wird **nil** und der Fehlertext zurückgeliefert.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
z:extract(1, 'D:\\Temp')
z:close()
```

2.10.2.4 extractall()

```
zipobject:extractall([path])
```

Extrahiert alle Dateien des Archivs in den Pfad *path* des Dateisystems. Wird *path* nicht angegeben oder ist es ein leerer String, so werden die Dateien in das aktuelle Verzeichnis extrahiert.
War die Aktion erfolgreich, so wird **true** zurückgeliefert. Im Fehlerfall wird **nil** und der Fehlertext zurückgeliefert.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
z:extractall('D:\\Temp')
z:close()
```

2.10.2.5 filecount()

```
zipobject:filecount()
```

Liefert die Anzahl der Dateien im Archiv.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
print(z:filecount())   =>   8
z:close()
```

2.10.2.6 fileinfo()

```
zipobject:fileinfo(index)
zipobject:fileinfo(filename)
```

Liefert Informationen über die Datei an Position *index* bzw. mit dem Namen *filename*. Die Dateiinformationen werden als Tabelle mit folgenden

Schlüsselwörtern zurückgeliefert:

MadeByVersion	Zip Version, mit der die Datei archiviert wurde
RequiredVersion	Zum entpacken benötigte Zip Version
Flag	Dateiattribute
CompressionMethod	Kompressionsmethode
ModifiedDateTime	Dateidatum der letzten Änderung
CRC32	Prüfsumme der Datei
CompressedSize	Komprimierte Dateigröße
UncompressedSize	Unkomprimierte Dateigröße

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
print(z:fileinfo(1))   =>   (Tabelle mit Dateiinfos)
z:close()
```

2.10.2.7 filename()

`zipobject:filename(index)`

Liefert den Namen der Datei welche an Position *index* im Archiv steht. Die Zählung beginnt bei 1. Ist die Indexposition ungültig, so wird ein leerer String zurückgeliefert.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
print(z:filename(1))   =>   'testfile.dat'
z:close()
```

2.10.2.8 filenames()

`zipobject:filenames()`

Liefert eine Liste mit den Namen aller Dateien im Archiv.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
z:filenames()   =>   (Tabelle mit Dateinamen)
z:close()
```

2.10.2.9 getcomment()

`zipobject:getcomment()`

Liefert den im Archiv eingetragenen Kommentar.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
print(z:getcomment())   =>   'kommentar'
z:close()
```

2.10.2.10 getfilecomment()

`zipobject:getfilecomment(index)`
`zipobject:getfilecomment(filename)`

Liefert den Kommentar der Datei an Position *index* bzw. mit dem Namen *filename*.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
print(z:getfilecomment(1))   =>   'kommentar'
z:close()
```

2.10.2.11 indexof()

zipobject:indexof(filename)

Liefert die Indexposition der Datei mit dem Namen *filename* im Archiv. Ist keine Datei mit diesem Namen vorhanden, so wird 0 zurückgeliefert.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
print(z:indexof('testfile.dat'))   =>   1
z:close()
```

2.10.2.12 open()

zipobject:open(zipfile [, readonly])

Öffnet das Zip Archiv mit dem *zipfile*. Wird *readonly* angegeben und ist es **true**, so wird das Archiv schreibgeschützt geöffnet.
War die Aktion erfolgreich, so wird **true** zurückgeliefert. Im Fehlerfall wird **nil** und der Fehlertext zurückgeliefert.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
z:add('D:\\testfile.dat')
z:close()
```

2.10.2.13 setcomment()

zipobject:setcomment([text])

Ändert den im Archiv eingetragenen Kommentar in *text*. Wird *text* nicht angegeben oder ist es ein leerer String, so wird der Kommentar gelöscht.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
z:setcomment('kommentar'))
z:close()
```

2.10.2.14 setfilecomment()

zipobject:setfilecomment(index [, text])
zipobject:setfilecomment(filename [, text])

Ändert den Kommentar der Datei an Position *index* bzw. mit dem Namen *filename* in *text*. Wird *text* nicht angegeben oder ist es ein leerer String, so wird der Kommentar gelöscht.

Beispiel:

```
z = zip()
z:open('D:\\test.zip')
print(z:setfilecomment(1, 'kommentar'))
z:close()
```