Lua 5.1
A Short Reference

Acknowledgments

Lua 5.1 Short Reference is a reformatted and updated version of Enrico Colombini's "Lua 5.0 Short Reference (draft 2)" in which he acknowledged others "I am grateful to all people that contributed with notes and suggestions, including John Belmonte, Albert-Jan Brouwer, Tiago Dionizio, Marius Gheorghe, Asko Kauppi, Philippe Lhoste, Virgil Smith, Ando Sonenblick, Nick Trout and of course Roberto Ierusalimschy, whose 'Lua 5.0 Reference Manual' and 'Programming in Lua' have been my main sources of Lua lore". This Lua 5.1 update further acknowledges and thanks Enrico Colombini's for his Lua 5.0 Short Reference (draft 2) and Roberto Ierusalimschy's 'Lua 5.1 Reference Manual' and 'Programming in Lua, 2nd Edition'.

This Short Reference update was done as a means of becoming familiar with Lua, so it has been edited and extended from the perspective of a new-comer to Lua.

Graham Henstridge

Lua 5.1 Short Reference

Introduction

Lua is a free modern and evolving scripting language designed by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes at Tecgraf, PUC-Rio, Brazil. Note: depreciated functions have been omitted.

Reserved words

and	**break**	**do**	**else**	**elseif**	**end**	**false**	**for**
function		**if**	**in**	**local**	**nil**	**not**	**or**
repeat	**return**	**then**	**true**	**until**	**while**		

_A... A system variable, **A** = any uppercase letter.

Other reserved strings

\+ - * / % ^ # == ~= <= >= < > = ( )
{ } [ ] ; : , ...

Comments

--	Comment to end of line.
--[[... **--]]**	Multi-line comment.
#!	At start of first line for Linux executable.

Strings and escape sequences

' ' **" "** **[[ ]]** **[=[ ]=]** string delimiters; **[[ ]]** can be multi-line, escape sequences are ignored. If **[=[ ]=]** number of **=**'s must balance

\a (bell)	**\b** (backspace)	**\f** (form feed)
\nnewline)	**\r** (return)	**\t** (tab)
\v (vert. tab)	**** (backslash)	**\"** (double quote)
\' (single quote)	**\[** (square bracket)	**\]** (square bracket)

\ddd (character represented decimal number).

Types

boolean	**number**	**string**	**table**
function	**thread**	**userdata**	**nil**

For booleans, **nil** and **false** count as **false**, all the rest is **true** including **0** and "" (null string). The type belongs to the value, NOT the variable to which it is assigned.

Operators, decreasing precedence

^	right-associative, math lib required				
not	**#**	**-** (unary negative sign)			
*****	**/**	**%**			
+	**-**				
..	(string concatenation, right-associative)				
<	**>**	**<=**	**>=**	**~=**	**==**
and	(stops on false/nil)				
or	(stops on true (not false/nil))				

Assignment and coercion examples

a = 5	Simple assignment.
a = "hi"	Variables are not typed, they can hold different types.
a, b, c = 1, 2, 3	Multiple assignment.
a, b = b, a	Swap values, because right side values evaluated before assignment.
a, b = 4, 5, 6	Too many values, **6** is discarded.
a, b = "there"	Too few values, **nil** is assigned to **b**.
a = nil	**a**'s prior value will be garbage collected if unreferenced elsewhere.
a = #b	Size of **b.** If table, then first index that is followed by a **nil** value.
a = z	If z is not defined **a** = **nil**.
a = "3" + "2"	Strings converted to numbers: **a = 5**.
a = 3 .. 2	Numbers are converted to strings: **a = "32"**.

Relational and boolean examples

"abc" < "abe"	True: based first different character
"ab" < "abc"	True: missing character is less than any

Scope, blocks and chunks

scope	By default all variables global.
local	Reduces scope from point of definition to end of block.
block	Is the body of a control structure, body of a function or a chunk.
chunk	A file or string of script.

Control structures

do block **end**
while exp **do** block **end**
repeat block **until** exp
if exp **then** block {**elseif** exp **then** block} [**else** block] **end**
for var **=** start, **end** [, step] **do** block **end**
for k, v **in** iterator **do** block **end**
break (exits loop, but must be last statement in block).

Table constructors

t = {} A new empty table.

t = {"yes", "no"}
A new simple array, elements are t[1] = **yes**, t[2] = **no**.

t = {[1] = "yes", [2] = "no"}
Same as line above.

t = {[-900] = 3, [+900] = 4}
Sparse array, two elements.

t = {x=5, y=10}
Hash table or dictionary, fields t["x"], t["y"] or t.x, t.y.

t = {x=5, y=10; "yes", "no"}
Mixed fields: elements t.x, t.y, t[1], t[2].

t = {msg = "choice", {"yes", "no"}}
Table containing a table as field.

Function definition

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

local function name **(** **args** **)** body [**return** values] **end**
Function local to chunk.

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

function (**...** **)** body [**return** values] **end**
(**...**) indicates variable args and {**...**} places them in a table where they processed in standard way.

function t.name **(** **args** **)** body [**return** values] **end**
Shortcut for t.name = **function** [...]

function obj:name **(** **args** **)** body [**return** values] **end** Object function getting extra **arg** **self**.

Functions can return multiple results.

Function call

f (**args** **)**	Simple call, returning zero or more values.
t.f (args)	Calling function stored in field **f** of table **t**.
f "string"	Calling with a single string argument
f {table}	Calling with a single table argument

Function examples

local function f (**mode**, **value** **)** body **end**
a local function with two arguments.

r = f {value = 3.14, mode = "auto"}
By passing a table, args can be passed by name where function is defined: **function f (** **t** **)** body **end**, and arg accessed as **t.value**, **t.mode**.

Metatable operations

setmetatable (**t, mt** **)**
Sets mt as metatable for **t**, unless **t**'s metatable has a **__metatable** field.

getmetatable (**t** **)**
Returns **__metatable** field of **t**'s metatable, or **t**'s metatable, or **nil**.

rawget (**t, i** **)**
Gets **t[i]** of a table without invoking metamethods.

rawset (**t, i, v** **)**
Sets **t[i]** = **v** on a table without invoking metamethods.

rawequal (**t1, t2** **)**
Returns boolean (**t1** == **t2**) without invoking metamethods.

Metatable fields (for tables and userdata)

__add	Sets handler **h (a, b)** for '**+**'.
__sub	Sets handler **h (a, b)** for binary '**-**'.
__mul	Sets handler **h (a, b)** for '*****'.
__div	Sets handler **h (a, b)** for '**/**'.
__pow	Sets handler **h (a, b)** for '**^**'.
__unm	Sets handler **h (a)** for unary '**-**'.
__concat	Sets handler **h (a, b)** for '**..**'.
__eq	Sets handler **h (a, b)** for '**==**', '**~=**'.
__lt	Sets handler **h (a, b)** for '**<**', '**>**' and '**<=**', '**>=**' if no **__le**
__le	Sets handler **h (a, b)** for '**<=**', '**>=**'.
__index	Sets handler **h (t, k)** for non-existing field access.
__newindex	Sets handler **h (t, k)** for new field assignment.
__call	Sets handler **h (f, ...)** for function call (using the object as a function).
__tostring	Sets handler **h (a)** to convert to string, e.g. for **print ( )**.
__gc	Set finalizer **h (ud)** for userdata (can be set from the C side only).
__mode	Table mode: '**k**' = weak keys, '**v**' = weak values, '**kv**' = both.
__metatable	Set value returned by **getmetatable ( )**.

The Basic Library

Environment and global variables

getfenv ([f])
If **f** a function, returns its environment; if **f** a number, returns the environment of function at level **f** (1 = current [default], 0 = global); if the environment has a field **__fenv**, that is returned.

setfenv (f, t)
Sets environment for function **f** or function at level **f** (0 = current thread); if the original environment has a field **__fenv**, raises an error.

_G	Variable whose value = global environment.
_VERSION	Variable with interpreter's version.

Loading and executing

require (module)
Loads **module** and returns final value of **package.loaded[module]** or raises error. In order, checks if already loaded, for Lua module, for C library.

module (name [, ...])
Creates a module. If a table in package.loaded[**name**] this is the module, else if a global table **t** of **name**, that table is the module, else creates new table **t** assigned to **name**. Initializes **t._NAME** to **name**, **t._M** to **t** and **t._PACKAGE** with package **name**.
Optional functions passed to be applied over module

dofile ([filename])
Loads and executes the contents of **filename** [default: standard input]. Returns file's returned values.

load (f [, n])
Loads a chunk using function **f** to get its pieces. Each **f** call to return a string (last = **nil**) that is concatenated. Returns compiled chunk as a function or **nil** and error message. Optional chunk name = **n** for debugging.

loadfile (n)
Loads contents of file **n**, without executing. Returns compiled chunk as function, or **nil** and error message.

loadstring (s [, n])
Returns compiled string **s** chunk as function, or **nil** and error message. Sets chunk name = **n** for debugging.

loadlib (lib, func)
Links to dynamic library named **lib** (**.so** or **.dll**). Returns function named **func**, or **nil** and error message.

pcall (f [, args])
Calls function **f** in protected mode; returns **true** and results if OK, else **false** and error message.

xpcall (f, h)
As **pcall ()** but passes error handler **h** instead of extra args; returns as **pcall ()** but with the result of **h ()** as error message, if any (use **debug.traceback ()** from the debug library for extended error info).

Simple output and error feedback

print (args)
Prints each of passed **args** to **stdout** using **tostring**.

error (msg [, n])
Terminates the program or the last protected call (e.g. **pcall ()**) with error message **msg** quoting level **n** [default: 1, current function].

assert (v [, msg])
Calls error (**msg**) if **v** is **nil** or false [default **msg**: "**assertion failed!**"].

Information and conversion

select (i, ...)
For numeric index **i**, returns the **i**'th argument from the **...** argument list. For **i** = string **"#"** (including quotes) returns total number of arguments including **nil**'s.

type (x)
Returns the type of **x** as a string (e.g. "**nil**", "**string**").

tostring (x)
Converts **x** to a string, using table's metatable's **__tostring** if available.

tonumber (x [, b])
Converts string **x** representing a number in base **b** [2..36, default: 10] to a number, or **nil** if invalid; for base 10 accepts full format (e.g. "1.5e6").

unpack (t)
Returns **t[1]..t[n]** as separate values, where **n** = **#t**.

Iterators

ipairs (t)
Returns an iterator getting index, value pairs of array **t** in numeric order.

pairs (t)
Returns an iterator getting key, value pairs of table **t** in no particular order.

next (t [, index])
Returns next index-value pair (**nil** when finished) from **index** (default **nil**, i.e. beginning) of table **t**.

Garbage collection

collectgarbage (opt [, v])
where **opt** can be:

"**stop**"	Stops garbage collection.
"**restart**"	Restart garbage collection.
"**collect**"	Initiates a full garbage collection.
"**count**"	Returns total memory used.
"**step**"	Perform garbage collection step size **v**, returns true if it finished a cycle.
"**setpause**"	Set **pause** to **v**/100.
"**setstepmul**"	Sets **multiplier** to **v**/100.

The garbage collector can be tuned using the **pause** and **multiplier** values:

pause	Determines how long waits, larger value is less aggressive. Default = 2.
multiplier	Controls speed of collection relative to memory allocation. Default = 2.

Coroutines

coroutine.create (f)
Creates a new coroutine with function **f**, returns it.

coroutine.resume (co, args)
Starts or continues running coroutine **co**, passing **args** to it. Returns **true** (and possibly values) if **co** calls **coroutine.yield ( )** or terminates, returns **false** and a message in case of error.

coroutines.running ( )
Returns current running coroutine or **nil** if main thread.

coroutine.yield (args)

Suspends execution of the calling coroutine (not from within C functions, metamethods or iterators), any **args** become extra return values of **coroutine.resume ( )**.

coroutine.status (co)
Returns the status of coroutine **co** as a string: either "**running**", "**suspended**" or "**dead**".

coroutine.wrap (f)
Creates coroutine with function **f** as body and returns a function that acts as **coroutine.resume ( )** without first arg and first return value, propagating errors.

The Package Library

package.cpath
A variable used by **require ( )** for a C loader. Set at startup to environment variable LUA_CPATH. (see Path Formats below).

package.loaded
Table of packages already loaded. Used by **require ( )**

package.loadlib (lib, fun)
Dynamically links to library **lib**, which must include path. Looks for function **fun** (same name exported by **lib**).

package.path
Variable used by **require ( )**. Set at startup to environment variable **LUA_PATH**. (see Path Formats).

package.preload
A table to store loaders for specific modules.

package.seeall (m)
Sets a metatable for module **m** with **_index** field referring to global environment.

Path Formats

A path is a sequence of path templates separated by semicolons. For each template, **require (filename)** will substitute each "?" by **filename**, in which each dot replaced by a "directory separator" ("/" in Linux); then it will try to load the resulting file name. Example:
require (dog.cat) with path **/usr/share/lua/?.lua;lua/?.lua** will attempt to load **cat.lua** from **/usr/share/lua/dog/** or **lua/dog/**

The Table Library

Tables as arrays (lists)

table.insert (t, [i,] v)
Inserts **v** at numerical index **i** [default: after the end] in table **t**, increments table size.

table.remove (t [, i])
Removes element at numerical index **i** [default: last element] from table **t**, decrements table size, returns the removed element or no value on empty table.

table.maxn (t)
Returns largest numeric index of table **t**. Slow.

table.sort (t [, cf])
Sorts (in-place) elements from **t**[1] to **t**[#**t**], using compare function **cf** (**e1**, **e2**) [default: '<'].

table.concat (t [, s [, i [, j]]])
Returns a single string made by concatenating table elements **t**[**i**] to **t**[**j**] (default: **i** =1, **j** = **table.getn ( )**) separated by string **s**; returns empty string if no given elements or **i** > **j**

Iterating on table contents

Use the **pairs** or **ipairs** iterators in a **for** loop. Example:
for k, v in pairs(table) do print (k, v) end
will print the key (**k**) and value (**v**) of all the **table**'s content.

The Math Library

Basic operations

math.abs (x) Returns the absolute value of **x**.
math.fmod (x, y) Returns the remainder of **x / y** as a rounded-down integer, for **y ~= 0**.
math.floor (x) Returns **x** rounded down to integer.
math.ceil (x) Returns **x** rounded up to the nearest integer.
math.min(args) Returns minimum value from **args**.
math.max(args) Returns maximum value from **args**.
math.huge ( ) Returns largest represented number
math.modf (x) Returns integer and fractional parts of **x**

Exponential and logarithmic

math.sqrt (x) Returns square root of **x**, for **x** >= 0.
math.pow (x, y) Returns x raised to the power of y, i.e. **x^y**; if **x** < 0, **y** must be integer.
math.exp (x) Returns **e** to the power of **x**, i.e. **e^x**.
math.log (x) Returns natural logarithm of **x**, for **x** >= 0.
math.log10 (x) Returns base-10 logarithm of **x**, for **x >= 0**.
math.frexp (x) If **x** = m**2^e**, returns **m** (normalized) and **e**.
math.ldexp (x, y) Returns **x2^y** with **y** integer.

Trigonometrical

math.deg (a) Converts angle **a** from radians to degrees.
math.rad (a) Converts angle **a** from degrees to radians.
math.pi Constant containing the value of **Pi**.
math.sin (a) Sine of angle **a** in radians.
math.cos (a) Cosine of angle **a** in radians.
math.tan (a) Tangent of angle **a** in radians.
math.asin (x) Arc sine of **x** in radians, for **x** in [-1, 1].
math.acos (x) Arc cosine of **x** in radians, for **x** in [-1, 1].
math.atan (x) Arc tangent of **x** in radians.

Pseudo-random numbers

**math.random ([n [, m]) **
Pseudo-random number in range [0, 1], [1, **n**] or [**n, m**].

math.randomseed (n)
Sets a seed **n** for random sequence.

The String Library

Basic operations

string.len (s)
Returns length of string **s**, including embedded zeros.

string.sub (s, i [, j])
Returns substring of **s** from position **i** to **j** [default: -1].

string.rep (s, n)
Returns a string of **n** concatenated copies of string **s**.

string.upper (s)
Returns a copy of **s** converted to uppercase.

string.lower (s)
Returns a copy of **s** converted to lowercase.

Character codes

string.byte (s [, i])
Ascii code of character at position **i** [default: **1**] in string **s**, or **nil** if invalid **i**.

string.char (args)
Returns a string made of the characters whose ascii codes are passed as **args**.

Formatting

string.format (s [, args])
Returns a copy of **s** where formatting directives beginning with '**%**' are replaced by the value of arguments **args**. (see Formatting directives below)

Finding, replacing, iterating

string.find (s, p [, i [, d]])
Returns first and last position of pattern **p** in string **s**, or **nil** if not found, starting search at position **i** [default: 1]; returns parenthesized 'captures' as extra results. If **d** is true, treat pattern as plain string. (see Patterns below)

string.gmatch (s, p)
Returns an iterator getting next occurrence of pattern **p** (or its captures) in string **s** as substring(s) matching the pattern. (see Patterns below)

string.gsub (s, p, r [, n])
Returns a copy of **s** with up to **n** [default: 1] occurrences of pattern **p** (or its captures) replaced by **r.** If **r** is a string (**r** can include references to captures of form **%n**). If **r** is table, first capture is key. If **r** is a function, it is passed all captured substrings, and should return replacement

string, alternatively with a **nil** or **false** return, original match is retained. Returns as second result, number of substitutions made (see Patterns below).

Function storage

string.dump (f)
Returns binary representation of function **f**. Use with **loadstring ( )**. **f** must be Lua function with no upvalues.

Note: String indexes go from 1 to **string.len (s)**, from end of string if negative (index -1 refers to the last character).

Formatting directives for string.format

% [flags] [field_width] [.precision] type

Formatting field types

%d Decimal integer.
%o Octal integer.
%x Hexadecimal integer, uppercase if **%X**.
%f Floating-point in the form [-]nnnn.nnnn.
%e Floating-point in exp. form [-]n.nnnn e [+|-]nnn, uppercase if **%E**.
%g Floating-point as **%e** if exp. < -4 or >= precision, else as **%f**; uppercase if **%G**.
%c Character having the code passed as integer.
%s String with no embedded zeros.
%q String between double quotes, with all special characters escaped.
%% The '**%**' character.

Formatting flags

- Left-justifies, default is right-justify.
+ Prepends sign (applies to numbers).
(space) Prepends sign if negative, else space.
Adds "**0x**" before **%x**, force decimal point; for **%e**, **%f**, leaves trailing zeros for **%g**.

Formatting field width

n Puts at least **n** characters, pad with blanks.
0n Puts at least **n** characters, left-pad with zeros

Formatting precision

.n Use at least **n** digits for integers, rounds to **n** decimals for floating-point or no more than **n** chars. for strings.

Formatting examples

string.format ("dog: %d, %d",7,27)	dog: 7, 27
string.format ("<%5d>", 13)	< 13>
string.format ("<%-5d>", 13)	<13 >
string.format ("<%05d>", 13)	<00013>
string.format ("<%06.3d>", 13)	< 013>
string.format ("<%f>", math.pi)	<3.141593>
string.format ("<%e>", math.pi)	<3.141593e+00>
string.format ("<%.4f>", math.pi)	<3.1416>
string.format ("<%9.4f>", math.pi)	< 3.1416>
string.format ("<%c>", 64)	<@>
string.format ("<%s.4>", "goodbye")	<good>
string.format("%q",[[she said "hi"]])	"she said "hi""

Patterns and pattern items

General pattern format: pattern_item [pattern_items]

cc Matches a single character in the class **cc** (see Pattern character classes below).
cc* Matches zero or more characters in the class **cc**; matches longest sequence.
cc- Matches zero or more characters in the class **cc**; matches shortest sequence.
cc+ Matches one or more characters in the class **cc**; matches longest sequence.
cc? Matches zero or one character in the class cc.
%n (n = 1..9) Matches **n**-th captured string.
%bxy Matches balanced string from character **x** to character **y** (e.g. nested parenthesis).
^ Anchor pattern to string start, must be first in pattern.
$ Anchor pattern to string end, must be last in pattern.

Pattern captures

(sub_pattern) Stores substring matching sub_pattern as capture **%1**..**%9**, in order.
() Stores current string position as capture **%1**..**%9**, in order.

Pattern character classes (cc's)

. Any character.
%a Any letter.
%A Any non-letter.
%c Any control character.
%C Any non-control character.
%d Any digit.
%D Any non-digit.
%l Any lowercase letter.
%L Any non-(lowercase letter).
%p Any punctuation character
%P Any non-punctuation character
%s Any whitespace character.
%S Any non-whitespace character.
%u Any uppercase letter.
%U Any non-uppercase letter.
%w Any alphanumeric character.
%W Any non-alphanumeric character.
%x Any hexadecimal digit.
%X Any non-(hexadecimal digit).
%z The zero character.
%Z Any non-zero character.
%x (**x** = symbol) The symbol itself.
x If **x** not in **^$()%.[]*+-?** the character itself.
[set] Any character in any of the given classes, can also be a range [c1-c2].
[^set] Any character not in set.

examples

string.find("Lua is great!", "is")
> 5 6
string.find("Lua is great!", "%s")
> 4 4
string.gsub("Lua is great!", "%s", "-")
> Lua-is-great! 2
string.gsub("Lua is great!", "[%s%l]", "*")
> L**********! 11
string.gsub("Lua is great!", "%a+", "*")
> * * *! 3
string.gsub("Lua is great!", "(.)", "%1%1")
> LLuuaa iiss ggrreeaatt!! 13
string.gsub("Lua is great!", "%but", "")
> L! 1
string.gsub("Lua is great!", "^.-a", "LUA")
> LUA is great! 1
string.gsub("Lua is great!", "^.-a", function (s)
return string.upper(s) end)
> LUA is great! 1

The I/O Library

The I/O functions return **nil** and an error message on failure, unless otherwise stated; passing a closed file object raises an error instead.

Complete I/O

io.open (fn [, m])
Opens file with name **fn** in mode **m**: "**r**" = read [default], "**w**" = write, "**a**" = append, "**r+**" = update-preserve, "**w+**" = update-erase, "**a+**" = update-append (add trailing "**b**" for binary mode on some systems), returns a file object (an userdata with a C handle) usable with '**:**' syntax.

Note in the following that "**file:**" is an **io** meta-method call.

file:close ( )
Closes file.
file:read (formats)
Returns a value from file for each of the passed **formats**: "***n**" = reads a number, "***a**" = reads whole file as a string from current position ("" at end of file), "***l**" = reads a line (**nil** at end of file) [default], **n** = reads a string of up to **n** characters (**nil** at end of file).

file:lines ( )
Returns an iterator function reading line-by-line from file; the iterator does not close the file when finished.
file:write (values)
Write each of **values** (strings or numbers) to file, with no added separators. Numbers are written as text, strings can contain binary data (may need binary mode read).
file:seek ([p] [, of])
Sets current position in file relative to **p** ("**set**" = start of file [default], "**cur**" = current, "**end**" = end of file) adding offset **of** [default: zero]; returns new position in file.
file:flush ( )
Writes to file any data still held in memory buffers.

Simple I/O

io.input ([file])
Sets **file** as default input file; **file** can be either an open file object or a file name; in the latter case the file is opened for reading in text mode; returns a file object, the current one if no file given; raises error on failure.
io.output ([file])
Sets **file** as default output file (current output file is not closed); **file** can be either an open file object or a file name; in the latter case **file** is opened for writing in text mode. Returns a file object, the current one if no file given. Raises error on failure.
io.close ([file])
Closes file object **file**. Default: closes default output file.
io.read (formats)
Reads from default input file, same as **file:read ( )**.
io.lines ([fn])
Opens file name **fn** for reading. Returns an iterator function reading from it line-by-line. Iterator closes file when finished. If no **fn**, returns iterator reading lines from default input file.
io.write (values)
Writes to the default output file, same as **file:write ( )**.
io.flush ( )
Writes to default output file any data in buffers.

Standard files and utility functions

io.stdin Predefined input file object.
io.stdout Predefined output file object.
io.stderr Predefined error output file object.
io.type (x)
Returns string "**file**" if **x** is an open file, "**closed file**" if **x** is a closed file, **nil** if **x** is not a file object.
io.tmpfile ( )
Returns file object for temporary file (deleted when program ends).

The OS Library

Many characteristics of this library are determined by operating system support.

Date/time

Time and date accessed via time-table tt = {**year** = 1970-2135 , **month** = 1-12, **day** = 1-31, **hour** = 0-23, **min** = 0-59, **sec** = 0-59, **isdst** = **true-false**,}
os.time ([tt])
Returns date/time, in seconds since epoch, described by table **tt** [default: current]. Requires **year**, **month**, **day**; while **hour**, **min**, **sec**, **isdst** fields optional.
os.difftime (t2, t1)
Returns difference between two **os.time ( )** values.
os.date ([fmt [, t]])
Returns a table or string describing date/time **t** (that should be a value returned by **os.time**), according to the format string **fmt**:
! A leading "**!**" requests UTC time
***t** Returns a table similar to time-table
while the following format a string representation:
%a Abbreviated weekday name.
%A Full weekday name.
%b Abbreviated month name.
%B Full month name.
%c Date/time (default)
%d Day of month (01..31).
%H Hour (00..23).
%I Hour (01..12).
%M Minute (00..59).
%m Month (01..12).
%p Either "am" or "pm".
%S Second (00..61).
%w Weekday (0..6), 0 is Sunday.
%x Date only.
%X Time only.
%y Year (nn).
%Y Year(nnnn).
os.clock ( )
Returns the approx. CPU seconds used by program.

System interaction

os.execute (cmd)
Calls system shell to execute string **cmd**, returning status code.
os.exit ([code])
Terminates script, returning **code** [default: success].
os.getenv (var)
Returns a string with the value of the environment variable named **var**, or **nil** if no such variable exists.
os.setlocale (s [, c])
Sets the locale described by string **s** for category **c**: "**all**" (default), "**collate**", "**ctype**", "**monetary**", "**numeric**" or "**time**". Returns name of new locale, or **nil** if not set.
os.remove (fn)
Deletes file **fn**, or returns **nil** and error description.
os.rename (of, nf)
Renames file **of** to **nf**, or returns **nil** and error message.
os.tmpname ( )
Returns a string usable as name for a temporary file.

The Debug Library

The debug library functions are inefficient and should not be used in normal operation. In addition to debugging they can be useful for profiling.

Basic functions

debug.debug ( )
Enters interactive debugging shell (type "**cont**" to exit); local variables cannot be accessed directly.
debug.getfenv (o)
Returns the environment of object **o**
debug.getinfo ([c,] f [, w])
Returns table with information for function **f** in coroutine **c** or for function at level **f** [1 = caller], or **nil** if invalid level. Table keys are:
source
Name of file (prefixed by '@') or string where **f** defined.
short_src
Short version of source, up to 60 characters.
linedefined
Line of source where the function was defined.
what
"**Lua**" = Lua function, "**C**" = C function, "**main**" = part of main chunk.
name
Name of function, if available, or a reasonable guess if possible.
namewhat
Meaning of name: "**global**", "**local**", "**method**", "**field**" or "".
nups
Number of upvalues of the function.
func
The function itself.

Characters in string **w** select one or more groups of fields (default is all):
- **n** Returns fields name and namewhat.
- **f** Returns field func.
- **S** Returns fields **source**, **short_src**, **what** and **linedefined**.
- **l** Returns field **currentline**.
- **u** Returns field **nup**.

debug.getlocal ([c,] n, i)
Returns name and value of local variable at index i (from 1, in order of appearance) of the function at stack level n (1= caller) in coroutine c; returns **nil** if i is out of range, raises error if n is out of range.

debug.gethook ([c])
Returns current hook function, mask and count set with **debug.sethook ( )** for coroutine c.

debug.getmetatable (o)
Returns metatable of object o or **nil** if none.

debug.getregistry ( )
Returns registry table that contains static library data.

debug.getupvalue (f, i)
Returns name and value of upvalue at index i (from 1, in order of appearance) of function f. If i is out of range, returns **nil**.

debug.traceback ([c,] [msg])
Returns a string with traceback of call stack, prepended by msg. Coroutine c may be specified.

debug.setfenv (o, t)
Sets environment of object o to table t. Returns o.

debug.sethook ([[c,] h, m [, n]])
For coroutine c, sets function h as hook, called for events given in mask string m: "**c**" = function call, "**r**" = function return, "**l**" = new code line, optionally call h **( )** every n instructions. Event type received by h **( )** as first argument: "**call**", "**return**", "**tail return**", "**line**" (line number as second argument) or "**count**". Use **debug.getinfo (2)** inside h **( )** for info (not for "**tail_return**").

debug.setlocal ([c,] n, i, v)
Assigns value v to the local variable at index i (from 1, in order of appearance) of the function at stack level n (1= caller); returns **nil** if i is out of range, raises error if n is out of range. Coroutine c may be specified.

debug.setmatatable (o, t)
Sets metatable of object o to table t, which can be **nil**.

debug.setupvalue (f, i, v)
Assigns value v to upvalue at index i (from 1, in order of appearance) of function f. Returns **nil** if i is out of range.

The Stand-alone Interpreter

Command line syntax

lua [options] [script [arguments]]

Options

-	Loads and executes script from standard input (no args allowed).
-e stats	Executes the Lua statements contained in the literal string **stats**, can be used multiple times on same line.
-l filename	Loads and executes **filename** if not already loaded.
-i	Enters interactive mode after loading and execution of script.
-v	Prints version information.
--	Stops parsing options.

Recognized environment variables

LUA_INIT	If it contains a string in the form @filename loads and executes filename, else executes the string itself.
_PROMPT	Sets the prompt for interactive mode.

Special Lua variables

arg **nil** if no command line arguments, else table containing command line arguments starting from **arg[1]**, **arg.n** is the number of arguments, **arg [0]** = script name as given on command line and **arg[-1]** and lower indexes contain fields of command line preceding script name.

The Compiler

Command line syntax

luac [options] [scripts]

Options

-	Compiles from standard input.
-l	Produces a listing of the compiled bytecode.
-o filename	Sends output to **filename** [default: **luac.out**].
-p	Performs syntax and integrity checking only, does not output bytecode.
-s	Strips debug information; line numbers and local names are lost.
-v	Prints version information.
--	Stops parsing options.

Compiled chunks portable on machines with same word size.

Independent Libraries

Lua core is designed to be a minimalist, portable and modern scripting language. As such it has only basic libraries. Independent libraries add functionality. Some useful libraries include:

bitlib library

The small elegantly written library by Reuben Thomas provides a useful set of bit-wise functions to Lua. All function arguments should be integers. Non integers can return unexpected results.

bit.bnot (a)	One's complement of a.
bit.band (w1, ...)	Bitwise "**and**" of the w's
bit.bor (w1, ...)	Bitwise "**or**" of the w's
bit.bxor (w1, ...)	Bitwise "**exclusive** or" of the w's
bit.lshift (a, b)	a shifted left b places
bit.rshift (a, b)	a shifted logically right b places
bit.arshift (a, b)	a shifted arithmetically right b places
bit.mod (a, b)	Integer remainder of a divided by b

lua file system library

The lua file system library was written by Roberto Ierusalimschy, André Carregal and Tomás Guisasolaprovides. It is a convenient set of machine dependent file access functions:

lfs.attributes (filepath [, aname])
Returns a table with the file attributes single attribute (aname) or nil and error message. Attributes include:

dev	the device that the inode resides on.
ino	the inode number.
mode	string representing associated protection mode (**file**, **directory**, **link**, **socket**, **named pipe**, **char device**, **block device** or **other**)
nlink	number of hard links to the file
uid	user-id of owner
gid	group-id of owner.
rdev	device type, for special file inodes.
access	time of last access
modification	time of last data modification
change	time of last file status change
size	file size, in bytes
blocks	block allocated for file.
blksize	optimal file system I/O block size.

lfs.chdir (path)
Change dir to path. Returns true or **nil** and error string.

lfs.currentdir ( )
Current working directory string or **nil** and error string.

lfs.dir (path)
Returns an iterator function that returns a string for each entry directory, **nil** no more entries. Raises error if **path** not a directory.

lfs.lock (filehandle, mode[, start[, length]])
Locks an open file or a part of it. **Mode** "**r**" for a read/shared lock or "**w**" for a write/exclusive lock. Starting point **start** and length **length** both numbers. Returns true or **nil** and error string.

lfs.mkdir (dirname)
Creates a new directory **dirname**. Returns **true** or **nil** and error string.

lfs.rmdir (dirname)
Removes directory **dirname**. Returns **true** or **nil** and error string.

lfs.touch (filepath [, atime [, mtime]])
Set access **atime** and modification **mtime** times of file **filepath**. Times in seconds as **os.date()**. Defaults to current time. Returns **true** or **nil** plus an error string.

lfs.unlock (filehandle[, start[, length]])
Unlocks an open file or a part of it. **Start** and **length** both numbers. Returns **true** or **nil** plus an error string.

Examples

lfs.attributes("/var/spool/mail/root", "size")
returns the size of "**root**" in bytes

for f in **lfs.dir ("/tmp")** do print (f) end
prints all files and directories in /tmp directory