

# The Lua language (v5.1)

## Reserved identifiers and comments

<b>and</b>	<b>break</b>	<b>do</b>	<b>else</b>	<b>elseif</b>	<b>end</b>	<b>false</b>	<b>for</b>	<b>function</b>	<b>if</b>	<b>in</b>
<b>local</b>	<b>nil</b>	<b>not</b>	<b>or</b>	<b>repeat</b>	<b>return</b>	<b>then</b>	<b>true</b>	<b>until</b>	<b>while</b>	
-- ...	comment to end of line				--[= [ ]=]	multi line comment (zero or multiple '=' are valid)				
_X is "reserved" (by convention) for constants (with X being any sequence of uppercase letters)					#!	usual Unix shebang; Lua ignores whole first line if this starts the line.				

## Types (the string values are the possible results of base library function type())

'nil'	'boolean'	'number'	'string'	'table'	'function'	'thread'	'userdata'
-------	-----------	----------	----------	---------	------------	----------	------------

Note: for type boolean, **nil** and **false** count as false; everything else is true (including 0 and "").

## Strings and escape sequences

'...' and '...'	string delimiters; interpret escapes.	[=...]=]	multi line string; escape sequences are ignored.
\a bell	\b backspace	\f form feed	\n newline
\r return	\t horiz. tab	\v vert. tab	
\\ backslash	\" d. quote	\' quote	[\ [ sq. bracket
			\] sq. bracket
			\ddd decimal (up to 3 digits)

## Operators, decreasing precedence

^ (right associative, math library required)			
<b>not</b>	# (length of strings and tables)	- (unary)	
*	/	%	
+		-	
.. (string concatenation, right associative)			
<	>	<=	>=
		~=	==
<b>and</b> (stops on <b>false</b> or <b>nil</b> , returns last evaluated value)			
<b>or</b> (stops on <b>true</b> (not <b>false</b> or <b>nil</b> ), returns last evaluated value)			

## Assignment and coercion

<b>a = 5 b= "hi"</b>	simple assignment; variables are not typed and can hold different types. Local variables are lexically scoped; their scope begins after the full declaration (so that local <b>a = 5</b> ).
<b>local a = a</b>	multiple assignments are supported
<b>a, b, c = 1, 2, 3</b>	swap values: right hand side is evaluated before assignment takes place
<b>a, b = b, a</b>	excess values on right hand side ("6") are evaluated but discarded
<b>a, b = 4, 5, "6"</b>	for missing values on right hand side <b>nil</b> is assumed
<b>a, b = "there"</b>	destroys <b>a</b> ; its contents are eligible for garbage collection if unreferenced.
<b>a = nil</b>	if <b>z</b> is not defined it is <b>nil</b> , so <b>nil</b> is assigned to <b>a</b> (destroying it)
<b>a = z</b>	numbers expected, strings are converted to numbers (a = 5)
<b>a = "3" + "2"</b>	strings expected, numbers are converted to strings (a = "32")
<b>a = 3 .. 2</b>	

## Control structures

<b>do block end</b>	block; introduces local scope.
<b>if exp then block [elseif exp then block] [else block] end</b>	conditional execution
<b>while exp do block end</b>	loop as long as <i>exp</i> is true
<b>repeat block until exp</b>	exits when <i>exp</i> becomes true; <i>exp</i> is in loop scope.
<b>for var = start, end [, step] do block end</b>	numerical for loop; <i>var</i> is local to loop.
<b>for vars in iterator do block end</b>	iterator based for loop; <i>vars</i> are local to loop.
<b>break</b>	exits loop; must be last statement in block.

## Table constructors

<b>t = {}</b>	creates an empty table and assigns it to <b>t</b>
<b>t = {"yes", "no", "?"}</b>	simple array; elements are <b>t[1]</b> , <b>t[2]</b> , <b>t[3]</b> .
<b>t = { [1] = "yes", [2] = "no", [3] = "?" }</b>	same as above, but with explicit fields
<b>t = {[ -900 ] = 3, [900] = 4}</b>	sparse array with just two elements (no space wasted)
<b>t = {x=5, y=10}</b>	hash table, fields are <b>t["x"]</b> , <b>t["y"]</b> (or <b>t.x</b> , <b>t.y</b> )
<b>t = {x=5, y=10; "yes", "no"}</b>	mixed, fields/elements are <b>t.x</b> , <b>t.y</b> , <b>t[1]</b> , <b>t[2]</b>
<b>t = {msg = "choice", {"yes", "no", "?"}}</b>	tables can contain others tables as fields

## Function definition

<b>function name ( args ) body [return values] end</b>	defines function and assigns to global variable <b>name</b>
<b>local function name ( args ) body [return values] end</b>	defines function as local to chunk
<b>f = function ( args ) body [return values] end</b>	anonymous function assigned to variable <b>f</b>
<b>function ( [args, ] ... ) body [return values] end</b>	variable argument list, in <i>body</i> accessed as ...
<b>function t.name ( args ) body [return values] end</b>	shortcut for <b>t.name = function ...</b>
<b>function obj:name ( args ) body [return values] end</b>	object function, gets <i>obj</i> as additional first argument <b>self</b>

## Function call

<b>f (x)</b>	simple call, possibly returning one or more values
<b>f "hello"</b>	shortcut for <b>f("hello")</b>
<b>f 'goodbye'</b>	shortcut for <b>f('goodbye')</b>

<b>f [[see you soon]]</b>	shortcut for <b>f([[see you soon]])</b>
<b>f {x = 3, y = 4}</b>	shortcut for <b>f({x = 3, y = 4})</b>
<b>t.f (x)</b>	calling a function assigned to field <b>f</b> of table <b>t</b>
<b>x:move (2, -3)</b>	object call: shortcut for <b>x.move(x, 2, -3)</b>

## Metatable operations (base library required)

<b>setmetatable (t, mt)</b>	sets <b>mt</b> as metatable for <b>t</b> , unless <b>t</b> 's metatable has a <b>__metatable</b> field, and returns <b>t</b>
<b>getmetatable (t)</b>	returns <b>__metatable</b> field of <b>t</b> 's metatable or <b>t</b> 's metatable or <b>nil</b>
<b>rawget (t, i)</b>	gets <b>t[i]</b> of a table without invoking metamethods
<b>rawset (t, i, v)</b>	sets <b>t[i] = v</b> on a table without invoking metamethods
<b>rawequal (t1, t2)</b>	returns boolean ( <b>t1 == t2</b> ) without invoking metamethods

## Metatable fields (for tables and userdata)

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

# The base library [no prefix]

## Environment and global variables

<b>getfenv (f)</b>	if <b>f</b> is a function, returns its environment; if <b>f</b> is a number, returns the environment of function at level <b>f</b> (1 = current [default], 0 = global); if the environment has a field <b>__fenv</b> , returns that instead.
<b>setfenv (f, t)</b>	sets environment for function <b>f</b> (or function at level <b>f</b> , 0 = current thread); if the original environment has a field <b>__fenv</b> , raises an error. Returns function <b>f</b> if <b>f</b> ~= 0.
<b>_G</b>	global variable whose value is the global environment (that is, <b>_G._G == _G</b> )
<b>_VERSION</b>	global variable containing the interpreter's version (e.g. "Lua 5.1")

## Loading and executing

<b>require (pkgname)</b>	loads a package, raises error if it can't be loaded
<b>dofile (filename)</b>	loads and executes the contents of <b>filename</b> [default: standard input]; returns its returned values.
<b>load (func [, chunkname])</b>	loads a chunk (with chunk name set to <b>name</b> ) using function <b>func</b> to get its pieces; returns compiled chunk as function (or <b>nil</b> and error message).
<b>loadfile (filename)</b>	loads file <b>filename</b> ; return values like <b>load()</b> .
<b>loadstring (s [, name])</b>	loads string <b>s</b> (with chunk name set to <b>name</b> ); return values like <b>load()</b> .
<b>pcall (f [, args])</b>	calls <b>f()</b> in protected mode; returns <b>true</b> and function results or <b>false</b> and error message.
<b>xpcall (f, h)</b>	as <b>pcall()</b> but passes error handler <b>h</b> instead of extra args; returns as <b>pcall()</b> but with the result of <b>h()</b> as error message, if any.

## Simple output and error feedback

<b>print (args)</b>	prints each of the passed <i>args</i> to stdout using <b>tostring()</b> (see below)
<b>error (msg [, n])</b>	terminates the program or the last protected call (e.g. <b>pcall()</b> ) with error message <b>msg</b> quoting level <b>n</b> [default: 1, current function]
<b>assert (v [, msg])</b>	calls <b>error(msg)</b> if <b>v</b> is <b>nil</b> or <b>false</b> [default <b>msg</b> : "assertion failed!"]

## Information and conversion

<b>select (index, ...)</b>	returns the arguments after argument number <b>index</b> or (if <b>index</b> is "#") the total number of arguments it received after <b>index</b>
<b>type (x)</b>	returns the type of <b>x</b> as a string (e.g. "nil", "string"); see <i>Types</i> above.
<b>tostring (x)</b>	converts <b>x</b> to a string, using <b>t</b> 's metatable's <b>__tostring</b> if available
<b>tonumber (x [, b])</b>	converts string <b>x</b> representing a number in base <b>b</b> [2..36, default: 10] to a number, or <b>nil</b> if invalid; for base 10 accepts full format (e.g. "1.5e6").
<b>unpack (t)</b>	returns <b>t[1]..t[n]</b> (n = #t) as separate values

## Iterators

<b>ipairs (t)</b>	returns an iterator getting <b>index, value</b> pairs of array <b>t</b> in numerical order
<b>pairs (t)</b>	returns an iterator getting <b>key, value</b> pairs of table <b>t</b> in an unspecified order
<b>next (t [, inx])</b>	if <b>inx</b> is <b>nil</b> [default] returns first <b>index, value</b> pair of table <b>t</b> ; if <b>inx</b> is the previous index returns next <b>index, value</b> pair or <b>nil</b> when finished.

## Garbage collection

<b>collectgarbage</b> (opt [, arg])	generic interface to the garbage collector; <b>opt</b> defines function performed.
-------------------------------------	--

## Modules and the package library [package]

<b>module</b> (name, ...)	creates module <b>name</b> . If there is a table in <b>package.loaded[name]</b> , this table is the module. Otherwise, if there is a global table <b>name</b> , this table is the module. Otherwise creates a new table and sets it as the value of the global <b>name</b> and the value of <b>package.loaded[name]</b> . Optional arguments are functions to be applied over the module.
<b>package.loadlib</b> (lib, func)	loads dynamic library <b>lib</b> (e.g. .so or .dll) and returns function <b>func</b> (or <b>nil</b> and error message)
<b>package.path</b> , <b>package.cpath</b>	contains the paths used by <b>require()</b> to search for a Lua or C loader, respectively
<b>package.loaded</b>	a table used by <b>require</b> to control which modules are already loaded (see module)
<b>package.preload</b>	a table to store loaders for specific modules (see <b>require</b> )
<b>package.seecall</b> (module)	sets a metatable for <b>module</b> with its <b>__index</b> field referring to the global environment

## The coroutine library [coroutine]

<b>coroutine.create</b> (f)	creates a new coroutine with Lua function <b>f()</b> as body and returns it
<b>coroutine.resume</b> (co, args)	starts or continues running coroutine <b>co</b> , passing <b>args</b> to it; returns <b>true</b> (and possibly values) if <b>co</b> calls <b>coroutine.yield()</b> or terminates or <b>false</b> and an error message.
<b>coroutine.yield</b> (args)	suspends execution of the calling coroutine (not from within C functions, metamethods or iterators); any <b>args</b> become extra return values of <b>coroutine.resume()</b> .
<b>coroutine.status</b> (co)	returns the status of coroutine <b>co</b> : either <b>"running"</b> , <b>"suspended"</b> or <b>"dead"</b>
<b>coroutine.running</b> ()	returns the running coroutine or <b>nil</b> when called by the main thread
<b>coroutine.wrap</b> (f)	creates a new coroutine with Lua function <b>f</b> as body and returns a function; this function will act as <b>coroutine.resume()</b> without the first argument and the first return value, propagating any errors.

## The table library [table]

<b>table.insert</b> (t, [i,] v)	inserts <b>v</b> at numerical index <b>i</b> [default: after the end] in table <b>t</b>
<b>table.remove</b> (t [, i])	removes element at numerical index <b>i</b> [default: last element] from table <b>t</b> ; returns the removed element or <b>nil</b> on empty table.
<b>table.maxn</b> (t)	returns the largest positive numerical index of table <b>t</b> or zero if <b>t</b> has no positive indices
<b>table.sort</b> (t [, cf])	sorts (in place) elements from <b>t[1]</b> to <b>#t</b> , using compare function <b>cf(e1, e2)</b> [default: '<']
<b>table.concat</b> (t [, s [, i [, j]]])	returns a single string made by concatenating table elements <b>t[i]</b> to <b>t[j]</b> [default: <b>i=1, j=#t</b> ] separated by string <b>s</b> ; returns empty string if no elements exist or <b>i &gt; j</b> .

## The mathematical library [math]

### Basic operations

<b>math.abs</b> (x)	returns the absolute value of <b>x</b>
<b>math.mod</b> (x, y)	returns the remainder of <b>x / y</b> as a rounded-down integer, for <b>y</b> $\neq$ 0
<b>math.floor</b> (x)	returns <b>x</b> rounded down to the nearest integer
<b>math.ceil</b> (x)	returns <b>x</b> rounded up to the nearest integer
<b>math.min</b> (args)	returns the minimum value from the <b>args</b> received
<b>math.max</b> (args)	returns the maximum value from the <b>args</b> received

### Exponential and logarithmic

<b>math.sqrt</b> (x)	returns the square root of <b>x</b> , for <b>x</b> $\geq$ 0
<b>math.pow</b> (x, y)	returns <b>x</b> raised to the power of <b>y</b> , i.e. <b>x<sup>y</sup></b> ; if <b>x</b> < 0, <b>y</b> must be integer.
<b>__pow</b> (x, y)	global function added by the math library to make operator '^' work
<b>math.exp</b> (x)	returns e (base of natural logs) raised to the power of <b>x</b> , i.e. <b>e<sup>x</sup></b>
<b>math.log</b> (x)	returns the natural logarithm of <b>x</b> , for <b>x</b> $\geq$ 0
<b>math.log10</b> (x)	returns the base-10 logarithm of <b>x</b> , for <b>x</b> $\geq$ 0

### Trigonometrical

<b>math.deg</b> (a)	converts angle <b>a</b> from radians to degrees
<b>math.rad</b> (a)	converts angle <b>a</b> from degrees to radians
<b>math.pi</b>	constant containing the value of pi
<b>math.sin</b> (a)	returns the sine of angle <b>a</b> (measured in radians)
<b>math.cos</b> (a)	returns the cosine of angle <b>a</b> (measured in radians)
<b>math.tan</b> (a)	returns the tangent of angle <b>a</b> (measured in radians)
<b>math.asin</b> (x)	returns the arc sine of <b>x</b> in radians, for <b>x</b> in [-1, 1]
<b>math.acos</b> (x)	returns the arc cosine of <b>x</b> in radians, for <b>x</b> in [-1, 1]
<b>math.atan</b> (x)	returns the arc tangent of <b>x</b> in radians
<b>math.atan2</b> (y, x)	similar to <b>math.atan(y / x)</b> but with quadrant and allowing <b>x = 0</b>

### Splitting on powers of 2

<b>math.frexp</b> (x)	splits <b>x</b> into normalized fraction and exponent of 2 and returns both
<b>math.ldexp</b> (x, y)	returns <b>x * (2 ^ y)</b> with <b>x</b> = normalized fraction, <b>y</b> = exponent of 2

## Pseudo-random numbers

<b>math.random</b> ((n [, m])	returns a pseudo-random number in range [0, 1] if no arguments given; in range [1, <b>n</b> ] if <b>n</b> is given, in range [ <b>n</b> , <b>m</b> ] if both <b>n</b> and <b>m</b> are passed.
<b>math.randomseed</b> (n)	sets a seed <b>n</b> for random sequence (same seed = same sequence)

## The string library [string]

Note: string indexes extend from 1 to #string, or from end of string if negative (index -1 refers to the last character).  
Note: the string library sets a metatable for strings where the **\_\_index** field points to the string table. String functions can be used in object-oriented style, e.g. string.len(s) can be written s:len(); literals have to be enclosed in parentheses, e.g. ("xyz"):len().

### Basic operations

<b>string.len</b> (s)	returns the length of string <b>s</b> , including embedded zeros (see also # operator)
<b>string.sub</b> (s, i [, j])	returns the substring of <b>s</b> from position <b>i</b> to <b>j</b> [default: -1] inclusive
<b>string.rep</b> (s, n)	returns a string made of <b>n</b> concatenated copies of string <b>s</b>
<b>string.upper</b> (s)	returns a copy of <b>s</b> converted to uppercase according to locale
<b>string.lower</b> (s)	returns a copy of <b>s</b> converted to lowercase according to locale

### Character codes

<b>string.byte</b> (s [, i [, j]])	returns the platform-dependent numerical code (e.g. ASCII) of characters <b>s[i]</b> , <b>s[i+1]</b> , ..., <b>s[j]</b> . The default value for <b>i</b> is 1; the default value for <b>j</b> is <b>i</b> .
<b>string.char</b> (args)	returns a string made of the characters whose platform-dependent numerical codes are passed as <b>args</b>

### Function storage

<b>string.dump</b> (f)	returns a binary representation of function <b>f()</b> , for later use with <b>loadstring()</b> ( <b>f()</b> must be a Lua function with no upvalues)
------------------------	---

### Formatting

<b>string.format</b> (s [, args])	returns a copy of <b>s</b> where formatting directives beginning with '%' are replaced by the value of arguments <b>args</b> , in the given order (see <i>Formatting directives</i> below)
-----------------------------------	--

### 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 $\geq$ precision, else as %f; uppercase if %G.
%c	character having the (system-dependent) code passed as integer
%s	string with no embedded zeros
%q	string between double quotes, with all special characters escaped
%%	'%' character

### Formatting flags

-	left-justifies within <b>field_width</b> [default: right-justify]
+	prepends sign (only applies to numbers)
(space)	prepends sign if negative, else blank space
#	adds "0x" before %x, force decimal point for %e, %f, leaves trailing zeros for %g

### Formatting field width and precision

<b>n</b>	puts at least <b>n</b> (<100) characters, pad with blanks
<b>0n</b>	puts at least <b>n</b> (<100) characters, left-pad with zeros
<b>.n</b>	puts at least <b>n</b> (<100) digits for integers; rounds to <b>n</b> decimals for floating-point; puts no more than <b>n</b> (<100) characters for strings.

### Formatting examples

<b>string.format("results: %d, %d", 13, 27)</b>	results: 13, 27
<b>string.format("&lt;-%5d&gt;", 13)</b>	< 13>
<b>string.format("&lt;-%-5d&gt;", 13)</b>	<13 >
<b>string.format("&lt;-%05d&gt;", 13)</b>	<00013>
<b>string.format("&lt;-%06.3d&gt;", 13)</b>	< 013>
<b>string.format("&lt;%f&gt;", math.pi)</b>	<3.141593>
<b>string.format("&lt;%e&gt;", math.pi)</b>	<3.141593e+00>
<b>string.format("&lt;%4f&gt;", math.pi)</b>	<3.1416>
<b>string.format("&lt;-%9.4f&gt;", math.pi)</b>	< 3.1416>
<b>string.format("&lt;%e&gt;", 64)</b>	<@>
<b>string.format("&lt;-%.4s&gt;", "goodbye")</b>	<good>
<b>string.format("&lt;%q&gt;", [[she said "hi"]])</b>	"she said \"hi\""

## Finding, replacing, iterating (for the Patterns see below)

<b>string.find</b> (s, p [, i [, d]])	returns first and last position of pattern <b>p</b> in string <b>s</b> , or <b>nil</b> if not found, starting search at position <b>i</b> [default: 1]; returns captures as extra results. If <b>d</b> is true, treat pattern as plain string.
<b>string.gmatch</b> (s, p)	returns an iterator getting next occurrence of pattern <b>p</b> (or its captures) in string <b>s</b> as substring(s) matching the pattern.
<b>string.gsub</b> (s, p, r [, n])	returns a copy of <b>s</b> with up to <b>n</b> [default: all] occurrences of pattern <b>p</b> (or its captures) replaced by <b>r</b> if <b>r</b> is a string ( <b>r</b> can include references to captures in the form <i>%n</i> ). If <b>r</b> is a function <b>r()</b> is called for each match and receives captured substrings; it should return the replacement string. If <b>r</b> is a table, the captures are used as fields into the table. The function returns the number of substitutions made as second result.
<b>string.match</b> (s, p [, i])	returns captures of pattern <b>p</b> in string <b>s</b> (or the whole match if <b>p</b> specifies no captures) or <b>nil</b> if <b>p</b> does not match <b>s</b> ; starts search at position <b>i</b> [default: 1].

## Patterns and pattern items

General pattern format: *pattern\_item* [ *pattern\_items* ]

<b>cc</b>	matches a single character in the class <i>cc</i> (see <i>Pattern character classes</i> below)
<b>cc*</b>	matches zero or more characters in the class <i>cc</i> ; matchest longest sequence (greedy).
<b>cc+</b>	matches zero or more characters in the class <i>cc</i> ; matchest shortest sequence (non-greedy).
<b>cc?</b>	matches one or more characters in the class <i>cc</i> ; matchest longest sequence (greedy).
<b>cc?</b>	matches zero or one character in the class <i>cc</i>
<b>%n</b>	matches the <i>n</i> -th captured string ( <i>n</i> = 1..9, see <i>Pattern captures</i> )
<b>%bxy</b>	matches the balanced string from character <i>x</i> to character <i>y</i> (e.g. <b>%b()</b> for nested parentheses)
<b>^</b>	anchors pattern to start of string, must be the first item in the pattern
<b>\$</b>	anchors pattern to end of string, must be the last item in the pattern

## Captures

<b>(pattern)</b>	stores substring matching <i>pattern</i> as capture <b>%1..%9</b> , in order of opening parentheses
<b>()</b>	stores current string position as capture

## Pattern character classes

<b>.</b>	any character		
<b>%a</b>	any letter	<b>%A</b>	any non-letter
<b>%c</b>	any control character	<b>%C</b>	any non-control character
<b>%d</b>	any digit	<b>%D</b>	any non-digit
<b>%l</b>	any lowercase letter	<b>%L</b>	any non-(lowercase letter)
<b>%p</b>	any punctuation character	<b>%P</b>	any non-punctuation character
<b>%s</b>	any whitespace character	<b>%S</b>	any non-whitespace character
<b>%u</b>	any uppercase letter	<b>%U</b>	any non-(uppercase letter)
<b>%w</b>	any alphanumeric character	<b>%W</b>	any non-alphanumeric character
<b>%x</b>	any hexadecimal digit	<b>%X</b>	any non-(hexadecimal digit)
<b>%z</b>	the byte value zero	<b>%Z</b>	any non-zero character
<b>%x</b>	if <i>x</i> is a symbol the symbol itself	<i>x</i>	if <i>x</i> not in <b>^\$()%.*+? </b> the character itself
<b>[ set ]</b>	any character in any of the given classes; can also be a range [ <i>c1-c2</i> ], e.g. [a-z].	<b>[ ^set ]</b>	any character not in <i>set</i>

## Pattern examples

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

## The I/O library [io]

### Complete I/O

<b>io.open</b> (fn [, m])	opens file with name <b>fn</b> in mode <b>m</b> : "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 (a userdata with a C handle).
<b>file:close</b> ()	closes <b>file</b>
<b>file:read</b> ( <i>formats</i> )	returns a value from <b>file</b> for each of the passed <i>formats</i> : "*" = reads a number, "%a" = reads the whole <b>file</b> as a string from current position (returns "" at end of file), "*1" = reads a line ( <b>nil</b> at end of file) [default], <i>n</i> = reads a string of up to <i>n</i> characters ( <b>nil</b> at end of file)
<b>file:lines</b> ()	returns an iterator function for reading <b>file</b> line by line; the iterator does not close the file when finished.

<b>file:write</b> ( <i>values</i> )	writes each of the <i>values</i> (strings or numbers) to <b>file</b> , with no added separators. Numbers are written as text, strings can contain binary data (in this case, <b>file</b> may need to be opened in binary mode on some systems).
<b>file:seek</b> ([p] [, of])	sets the current position in <b>file</b> relative to <b>p</b> ("set" = start of file [default], "cur" = current, "end" = end of file) adding offset <b>of</b> [default: zero]; returns new current position in <b>file</b> .
<b>file:flush</b> ()	flushes any data still held in buffers to <b>file</b>

### Simple I/O

<b>io.input</b> ([file])	sets <b>file</b> as default input file; <b>file</b> 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 <b>file</b> given; raises error on failure.
<b>io.output</b> ([file])	sets <b>file</b> as default output file (the current output file is not closed); <b>file</b> can be either an open file object or a file name; in the latter case the file is opened for writing in text mode. Returns a file object, the current one if no <b>file</b> given; raises error on failure.
<b>io.close</b> ([file])	closes <b>file</b> (a file object) [default: closes the default output file]
<b>io.read</b> ( <i>formats</i> )	reads from the default input file, usage as <b>file:read()</b>
<b>io.lines</b> ([fn])	opens the file with name <b>fn</b> for reading and returns an iterator function to read line by line; the iterator closes the file when finished. If no <b>fn</b> is given, returns an iterator reading lines from the default input file.
<b>io.write</b> ( <i>values</i> )	writes to the default output file, usage as <b>file:write()</b>
<b>io.flush</b> ()	flushes any data still held in buffers to the default output file

### Standard files and utility functions

<b>io.stdin</b> , <b>io.stdout</b> , <b>io.stderr</b>	predefined file objects for stdin, stdout and stderr streams
<b>io.popen</b> ([prog [, mode]])	starts program <b>prog</b> in a separate process and returns a file handle that you can use to read data from (if <b>mode</b> is "r", default) or to write data to (if <b>mode</b> is "w")
<b>io.type</b> (x)	returns the string "file" if <b>x</b> is an open file, "closed file" if <b>x</b> is a closed file or <b>nil</b> if <b>x</b> is not a file object
<b>io.tmpfile</b> ()	returns a file object for a temporary file (deleted when program ends)

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

## The operating system library [os]

### System interaction

<b>os.execute</b> (cmd)	calls a system shell to execute the string <b>cmd</b> as a command; returns a system-dependent status code.
<b>os.exit</b> ([code])	terminates the program returning <b>code</b> [default: success]
<b>os.getenv</b> (var)	returns a string with the value of the environment variable <b>var</b> or <b>nil</b> if no such variable exists
<b>os.setlocale</b> (s [, c])	sets the locale described by string <b>s</b> for category <b>c</b> : "all", "collate", "ctype", "monetary", "numeric" or "time" [default: "all"]; returns the name of the locale or <b>nil</b> if it can't be set.
<b>os.remove</b> (fn)	deletes the file <b>fn</b> ; in case of error returns <b>nil</b> and error description.
<b>os.rename</b> (of, nf)	renames file <b>of</b> to <b>nf</b> ; in case of error returns <b>nil</b> and error description.
<b>os.tmpname</b> ()	returns a string usable as name for a temporary file; subject to name conflicts, use <b>io.tmpfile()</b> instead.

### Date/time

<b>os.clock</b> ()	returns an approximation of the amount in seconds of CPU time used by the program
<b>os.time</b> ([tt])	returns a system-dependent number representing date/time described by table <b>tt</b> [default: current]. <b>tt</b> must have fields <b>year</b> , <b>month</b> , <b>day</b> ; can have fields <b>hour</b> , <b>min</b> , <b>sec</b> , <b>isdst</b> (daylight saving, boolean). On many systems the returned value is the number of seconds since a fixed point in time (the "epoch").
<b>os.date</b> ([fmt [, t]])	returns a table or a string describing date/time <b>t</b> (should be a value returned by <b>os.time()</b> [default: current date/time]), according to the format string <b>fmt</b> [default: date/time according to locale settings]; if <b>fmt</b> is "*" or "!*", returns a table with fields <b>year</b> (yyyy), <b>month</b> (1..12), <b>day</b> (1..31), <b>hour</b> (0..23), <b>min</b> (0..59), <b>sec</b> (0..61), <b>wday</b> (1..7, Sunday = 1), <b>yday</b> (1..366), <b>isdst</b> (true = daylight saving), else returns the <b>fmt</b> string with formatting directives beginning with '%' replaced according to <i>Time formatting directives</i> (see below). In either case a leading "!" requests UTC (Coordinated Universal Time).
<b>os.difftime</b> (t2, t1)	returns the difference between two values returned by <b>os.time()</b>

### Time formatting directives (most used, portable features):

<b>%c</b>	date/time (locale)		
<b>%x</b>	date only (locale)	<b>%X</b>	time only (locale)
<b>%y</b>	year (nn)	<b>%Y</b>	year (yyyy)
<b>%j</b>	day of year (001..366)		
<b>%m</b>	month (01..12)		
<b>%b</b>	abbreviated month name (locale)	<b>%B</b>	full name of month (locale)
<b>%d</b>	day of month (01..31)		
<b>%U</b>	week number (01..53), Sunday-based	<b>%W</b>	week number (01..53), Monday-based
<b>%w</b>	weekday (0..6), 0 is Sunday		
<b>%a</b>	abbreviated weekday name (locale)	<b>%A</b>	full weekday name (locale)
<b>%H</b>	hour (00..23)	<b>%I</b>	hour (01..12)
<b>%p</b>	either AM or PM		
<b>%M</b>	minute (00..59)		
<b>%S</b>	second (00..61)		
<b>%Z</b>	time zone name, if any		

## The debug library [debug]

### Basic functions

<b>debug.debug</b> ()	enters interactive debugging shell (type <b>cont</b> to exit); local variables cannot be accessed directly.
<b>debug.getinfo</b> (f [, w])	returns a table with information for function <b>f</b> or for function at level <b>f</b> [ <b>f</b> = caller], or <b>nil</b> if invalid level (see <i>Result fields for getinfo</i> below); characters in string <b>w</b> select one or more groups of fields [default: all] (see <i>Options for getinfo</i> below).
<b>debug.getlocal</b> (n, i)	returns name and value of local variable at index <b>i</b> (from 1, in order of appearance) of the function at stack level <b>n</b> (1= caller); returns <b>nil</b> if <b>i</b> is out of range, raises error if <b>n</b> is out of range.
<b>debug.getupvalue</b> (f, i)	returns name and value of upvalue at index <b>i</b> (from 1, in order of appearance) of function <b>f</b> ; returns <b>nil</b> if <b>i</b> is out of range.
<b>debug.traceback</b> (msg)	returns a string with traceback of call stack, prepended by <b>msg</b>
<b>debug.setlocal</b> (n, i, v)	assigns value <b>v</b> to the local variable at index <b>i</b> (from 1, in order of appearance) of the function at stack level <b>n</b> (1= caller); returns <b>nil</b> if <b>i</b> is out of range, raises error if <b>n</b> is out of range.
<b>debug.setupvalue</b> (f, i, v)	assigns value <b>v</b> to the upvalue at index <b>i</b> (from 1, in order of appearance) of function <b>f</b> ; returns <b>nil</b> if <b>i</b> is out of range.
<b>debug.sethook</b> ([h, m [, n]])	sets function <b>h</b> as hook, called for events given in string (mask) <b>m</b> : "c" = function call, "r" = function return, "l" = new code line; also, a number <b>n</b> will call <b>h()</b> every <b>n</b> instructions; <b>h()</b> will receive the event type as first argument: "call", "return", "tail return", "line" (line number as second argument) or "count"; use <b>debug.getinfo(2)</b> inside <b>h()</b> for info (not for "tail_return").
<b>debug.gethook</b> ()	returns current hook function, mask and count set with <b>debug.sethook()</b>

Note: the debug library functions are not optimised for efficiency and should not be used in normal operation.

### Result fields for debug.getinfo

<b>source</b>	name of file (prefixed by '@') or string where the function was defined
<b>short_src</b>	short version of <b>source</b> , up to 60 characters
<b>linedefined</b>	line of source where the function was defined
<b>what</b>	"Lua" = Lua function, "C" = C function, "main" = part of main chunk
<b>name</b>	name of function, if available, or a reasonable guess if possible
<b>namewhat</b>	meaning of <b>name</b> : "global", "local", "method", "field" or ""
<b>nups</b>	number of upvalues of the function
<b>func</b>	the function itself

### Options for debug.getinfo (character codes for argument w)

<b>n</b>	returns fields <b>name</b> and <b>namewhat</b>	<b>l</b>	returns field <b>currentline</b>
<b>f</b>	returns field <b>func</b>	<b>u</b>	returns field <b>nup</b>
<b>S</b>	returns fields <b>source</b> , <b>short_src</b> , <b>what</b> and <b>linedefined</b>		

## The stand-alone interpreter

### Command line syntax

**lua** [*options*] [*script* [*arguments*]]

### Options

<b>-</b>	loads and executes <b>script</b> from standard input (no args allowed)
<b>-e stats</b>	executes the Lua statements in the literal string <i>stats</i> , can be used multiple times on the same line
<b>-l filename</b>	requires <i>filename</i> (loads and executes if not already done)
<b>-i</b>	enters interactive mode after loading and executing <i>script</i>
<b>-v</b>	prints version information

<b>--</b>	stops parsing options
-----------	-----------------------

### Recognized environment variables

<b>LUA_INIT</b>	if this holds a string in the form <i>@filename</i> loads and executes <i>filename</i> , else executes the string itself
<b>LUA_PATH</b>	defines search path for Lua modules, with "?" replaced by the module name
<b>LUA_CPATH</b>	defines search path for dynamic libraries (e.g. .so or .dll files), with "?" replaced by the module name
<b>_PROMPT[2]</b>	set the prompts for interactive mode

### Special Lua variables

<b>arg</b>	<b>nil</b> if no arguments on the command line, else a table containing command line <i>arguments</i> starting from <b>arg[1]</b> while <b>#arg</b> is the number of <i>arguments</i> ; <b>arg[0]</b> holds the script name as given on the command line; <b>arg[-1]</b> and lower indexes contain the fields of the command line preceding the script name.
<b>_PROMPT[2]</b>	contain the prompt for interactive mode; can be changed by assigning a new value.

## The compiler

### Command line syntax

**luac** [*options*] [*filenames*]

### Options

<b>-</b>	compiles from standard input
<b>-l</b>	produces a listing of the compiled bytecode
<b>-o filename</b>	sends output to <b>filename</b> [default: <b>luac.out</b> ]
<b>-p</b>	performs syntax and integrity checking only, does not output bytecode
<b>-s</b>	strips debug information; line numbers and local names are lost.
<b>-v</b>	prints version information
<b>--</b>	stops parsing options

Note: compiled chunks are portable between machines having the same word size.

*Lua is a language designed and implemented by Roberto Ierusalimsky, Luiz Henrique de Figueiredo and Waldemar Celes; for details see lua.org. Drafts of this reference card (for Lua 5.0) were produced by Enrico Colombini <erix@erix.it> in 2004 and updated by Thomas Lauer <thomas.lauer@gmail.com> in 2007, 2008 and 2009. Comments, praise or blame please to the lua-1 mailing list. This reference card can be used and distributed according to the terms of the Lua 5.1 license.*