Core Functions Tutorial

This page covers section 5.1 of the Reference Manual. These functions provide access to the core functionality of Lua. We will not go into detail about all of the topics here, but links will be mentioned to other parts of the tutorial for more detail.

assert(test [, message])

Similar to the `assert()` function in C. If the test condition is false or nil an error is raised; otherwise test is returned. An optional message is returned instead of a system message. Also, see the `error()` function. E.g.,

```
> assert(1==1)          -- no error as test was true
> assert(1==0)
stdin:1: assertion failed!
stack traceback:
        [C]: in function `assert'
        stdin:1: in main chunk
        [C]: ?
> assert("green"=="blue", "Colours not equal")
stdin:1: Colours not equal
stack traceback:
        [C]: in function `assert'
        stdin:1: in main chunk
        [C]: ?
```

Many Lua functions, such as `io.open`, return a value on success, or return nil and an error message on failure. This works well with `assert`:

```
file = assert(io.open(filename))
```

This either opens `filename` for reading and assigns it to `file`, or it raises an error with the message in the second return value from `io.open`.

collectgarbage([limit])

Sets the memory allocation limit at which the garbage collector will be called. If the new limit is less than the current amount of Lua memory allocation, or no argument is given, the collector is called immediately. See the GarbageCollectionTutorial for more information on.

```
> = gcinfo()            -- find current memory stats, i.e. 21kb used
```

```
21      35
> bigalloc = string.rep('a', 100000) -- create a big string
> = gcinfo()            -- 164kb allocated now
164     327
> collectgarbage()      -- force collection
> = gcinfo()            -- we did free some memory on collection
139     278
> bigalloc = nil        -- release the string we created
> = gcinfo()            -- it's not deleted until its collected
140     278
> collectgarbage()      -- collect
> = gcinfo()            -- it's deleted
29      59
```

dofile(filename)

TO DO: Opens the named file and executes its contents as a Lua chunk. When called without arguments, dofile executes the contents of the standard input (stdin). Returns any value returned by the chunk. In case of errors, dofile propagates the error to its caller (that is, it does not run in protected mode).

error(message [, level])

TO DO: Terminates the last protected function called, and returns message as the error message. Function error never returns. The level argument specifies where the error message points the error. With level 1 (the default), the error position is where the error function was called. Level 2 points the error to where the function that called error was called; and so on.

_G

`_G` is a global variable which points to the global environment. For example to display all of the globals variables we might do the following:

```
> table.foreach(_G,print)
string  table: 00357098
xpcall  function: 00354E10
tostring        function: 00354708
gcinfo  function: 00354E90
loadlib function: 00358B40
os      table: 00355AE0
unpack  function: 003547C8
level   2
require function: 00354F90
getfenv function: 00354548
... -- etc.
```

`_G` is also recursive as `_G` lives in `_G`! I.e. the following all point to the same table (which holds the global variables):

```
> = _G
table: 00353710
> = _G._G
table: 00353710
> = _G._G._G
table: 00353710
```

Lua itself does not use this variable, so changing its value does not affect any environment. You should use `setfenv()` to change environments.

getfenv(f)

TODO: Returns the current environment in use by the function. f can be a Lua function or a number, which specifies the function at that stack level: Level 1 is the function calling getfenv. If the given function is not a Lua function, or if f is 0, getfenv returns the global environment. The default for f is 1. if the environment has a "__fenv" field, returns the associated value, instead of the environment.

getmetatable(object)

TO DO: If the object does not have a metatable, returns nil. Otherwise, if the object's metatable has a "__metatable" field, returns the associated value. Otherwise, returns the metatable of the given object.

gcinfo()

TO DO: Returns two results: the number of Kbytes of dynamic memory that Lua is using and the current garbage collector threshold (also in Kbytes).

ipairs(t)

TODO: Returns an iterator function, the table t, and 0, so that the construction `for i,v in ipairs(t) do ... end` will iterate over the pairs (1,t[1]), (2,t[2]), ..., up to the first integer key with a nil value in the table.

loadfile(filename)

TODO: Loads a file as a Lua chunk (without running it). If there are no errors, returns the compiled chunk as a function; otherwise, returns nil plus the error message. The environment of the returned function is the global environment.

loadlib(libname, funcname)

> *Links the program with the dynamic C library libname. Inside this library, looks for a function funcname and returns this function as a C function. libname must be the complete file name of the C library, including any eventual path and extension. This function is not supported by ANSI C. As such, it is only available on some platforms (Windows, Linux, Solaris, BSD, plus other Unix systems that support the dlfcn standard).*

`loadlib` gives you the ability to enhance your lua scripts with self written C functions. The following example should give you some hints if you want to implement your own stuff. This was tested in Linux but should work on other platforms too. You should know how to create .so/.dll. All files in the example should be in the same directory.

```
mylib.c has to look like this:
------------------------
#include "lua.h"
#include "stdio.h"

// This function will be exported to lua
static int lua_myfunc(lua_State* l) {
  printf("blabla");
  return 0;
}

// This function is our Initialization
int init(lua_State* l) {
  printf("Registering personal functions");

  lua_register(l, "myfunc", lua_myfunc);

  printf("Done registering");
  return 0;
}
```

Now compile it as library:

gcc -Wall -g -O2 -shared `lua-config --include` -c mylib.c -o mylib.o

gcc mylib.o -Wall -g -O2 -shared `lua-config --include` `lua-config --libs` -o mylib.a

After this you should have a file called mylib.a - This is our library. Now lets write a simple test script in lua

```
------------------------
luainit = loadlib("./mylib.a", "init")

-- now call the initialization routine
```

```
luainit()

print("New registered function: " ..myfunc)

-- start the new function
myfunc()

print("well done.")
------------------------
```

That's it. If it doesn't work don't hesitate to write me a mail: reflex -2000 <aat> gmx < dottt> net

loadstring(string [, chunkname])

TODO: Loads a string as a Lua chunk (without running it). If there are no errors, returns the compiled chunk as a function; otherwise, returns nil plus the error message. The environment of the returned function is the global environment. The optional parameter chunkname is the name to be used in error messages and debug information.

next(table [, index])

TO DO: Allows a program to traverse all fields of a table. Its first argument is a table and its second argument is an index in this table. next returns the next index of the table and the value associated with the index. When called with nil as its second argument, next returns the first index of the table and its associated value. When called with the last index, or with nil in an empty table, next returns nil. If the second argument is absent, then it is interpreted as nil. Lua has no declaration of fields; There is no difference between a field not present in a table or a field with value nil. Therefore, next only considers fields with non-nil values. The order in which the indices are enumerated is not specified, even for numeric indices. (To traverse a table in numeric order, use a numerical for or the ipairs function.)

pairs(t)

TO DO: Returns the next function and the table t (plus a nil), so that the construction for k,v in pairs(t) do ... end

pcall(f, arg1, arg2, ...)

TO DO: Calls function f with the given arguments in protected mode. That means that any error inside f is not propagated; instead, pcall catches the error and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In such case, pcall also returns all results from the call, after this first result. In case of any error, pcall returns false plus the error message.

print(e1, e2, ...)

`print` can be passed any number of comma seperated arguments which it prints the values of to stdout. `print` uses `tostring` to convert the arguments into string form to be printed. E.g.

```
> print(1,2,"buckle my shoe", 22/7)
1       2       buckle my shoe  3.1428571428571
```

`print` is very simple and will not recurse into tables printing the content, it will just print the type and a unique id.

```
> print({1,2,3})
table: 002FE9C8
```

`print` does not format text. In order to do this you should use the `string.format()` function in conjunction with `print` (see StringLibraryTutorial) E.g.,

```
> print(string.format("Pi is approximately %.4f", 22/7))
Pi is approximately 3.1429
```

rawequal(v1, v2)

TO DO: Checks whether v1 is equal to v2, without invoking any metamethod. Returns a boolean.

rawget(table, index)

TO DO: Gets the real value of table[index], without invoking any metamethod. table must be a table; index is any value different from nil.

rawset(table, index, value)

TO DO: Sets the real value of table[index] to value, without invoking any metamethod. table must be a table, index is any value different from nil, and value is any Lua value.

require(packagename)

TO DO: Loads the given package. The function starts by looking into the table _LOADED to determine whether packagename is already loaded. If it is, then require returns the value that the package returned when it was first loaded. Otherwise, it searches a path looking for a file to load.

setfenv(f, table)

TO DO: Sets the current environment to be used by the given function. f can be a Lua function or a number, which specifies the function at that stack

level: Level 1 is the function calling setfenv.

setmetatable(table, metatable)

TO DO: Sets the metatable for the given table. (You cannot change the metatable of a userdata from Lua.) If metatable is nil, removes the metatable of the given table. If the original metatable has a "__metatable" field, raises an error.

tonumber(e [, base])

TO DO: Tries to convert its argument to a number. If the argument is already a number or a string convertible to a number, then tonumber returns that number; otherwise, it returns nil. An optional argument specifies the base to interpret the numeral. The base may be any integer between 2 and 36, inclusive. In bases above 10, the letter `A´ (in either upper or lower case) represents 10, `B´ represents 11, and so forth, with `Z´ representing 35. In base 10 (the default), the number may have a decimal part, as well as an optional exponent part (see 2.2.1). In other bases, only unsigned integers are accepted.

tostring(e)

TODO: Receives an argument of any type and converts it to a string in a reasonable format. For complete control of how numbers are converted, use format (see 5.3). If the metatable of e has a "__tostring" field, tostring calls the corresponding value with e as argument, and uses the result of the call as its result.

type(v)

`type` returns a string describing the type of the object passed to it.

```
> = type(1), type(true), type("hello"), type({}), type(function() end)
number  boolean string  table   function
```

Note, the type of `nil` is "nil".

```
> =type(nil)
nil
```

The possible types returned in Lua 5.0 are: "number", "string", "boolean, "table", "function", "thread", and "userdata". A "thread" is a coroutine, and "userdata" is a C data type with a reference to it in Lua.

unpack(list)

`unpack` takes the elements of a list and returns them, e.g.:

```
> = unpack( {1,2,3} )
1       2       3
> = unpack( {"one",2,6*7} )
one     2       42
```

The number of elements unpacked is defined by the size of the table, which is not necessarily the number of elements in the table! See the TableLibraryTutorial for more details on this.

```
> t = {"one",2,6*7}
> t.n = 2
> = table.getn(t)
2
> = unpack(t)
one     2
```

_VERSION

Straight from from the manual, `_VERSION` is "a global variable (not a function) that holds a string containing the current interpreter version."

```
> = _VERSION
Lua 5.0
```

xpcall(f, err)

TO DO: This function is similar to pcall, except that you can set a new error handler. xpcall calls function f in protected mode, using err as the error handler. Any error inside f is not propagated; instead, xpcall catches the error, calls the err function with the original error object, and returns a status code. Its first result is the status code (a boolean), which is true if the call succeeds without errors. In such case, xpcall also returns all results from the call, after this first result. In case of any error, xpcall returns false plus the result from err.
