Lua Style Guide

This style guides lists the coding conventions used in the LuaRocks project. It does not claim to be the best Lua coding style in the planet, but it is used successfully in a long-running project, and we do provide rationale for many of the design decisions listed below.

The list of recommendations in this document was based on the ones mentioned in the following style guides: (sometimes copying material verbatim, sometimes giving the opposite advice! :))

- https://github.com/Olivine-Labs/lua-style-guide/
- https://github.com/zaki/lua-style-guide
- http://lua-users.org/wiki/LuaStyleGuide
- http://sputnik.freewisdom.org/en/Coding_Standard
- https://gist.github.com/catwell/b3c01dbea413aa78675740546dfd5ce2

Indentation and formatting

- Let's address the elephant in the room first. LuaRocks is indented with 3 spaces.

```
for i, pkg in ipairs(packages) do
   for name, version in pairs(pkg) do
      if name == searched then
         print(version)
      end
   end
end
```

> **Rationale:** There is no agreement in the Lua community as for indentation, so 3 spaces lies nicely as a middle ground between the 2-space camp and the 4-space camp. Also, for a language that nests with `do` / `end` blocks, it produces pleasant-looking block-closing staircases, as in the example above.

- One should not use tabs for indentation, or mix it with spaces.
- Use LF (Unix) line endings.

Documentation

- Document function signatures using LDoc. Specifying typing information after each parameter or return value is a nice plus.

```
--- Load a local or remote manifest describing a repository.
-- All functions that use manifest tables assume they were obtained
-- through either this function or load_local_manifest.
-- @param repo_url string: URL or pathname for the repository.
-- @param lua_version string: Lua version in "5.x" format, defaults to installed version.
-- @return table or (nil, string, [string]): A table representing the manifest,
-- or nil followed by an error message and an optional error code.
function manif.load_manifest(repo_url, lua_version)
   -- code
end
```

- Use `TODO` and `FIXME` tags in comments. `TODO` indicates a missing feature to be implemented later. `FIXME` indicates a problem in the existing code (inefficient implementation, bug, unnecessary code, etc).

```
-- TODO: implement method
local function something()
   -- FIXME: check conditions
end
```

- Prefer LDoc comments over the function that explain *what* the function does than inline comments inside the function that explain *how* it does it. Ideally, the implementation should be simple enough so that comments aren't needed. If the function grows complex, split it into multiple functions so that their names explain what each part does.

Variable names

- Variable names with larger scope should be more descriptive than those with smaller scope. One-letter variable names should be avoided except for very small scopes (less than ten lines) or for iterators.
- `i` should be used only as a counter variable in for loops (either numeric for or `ipairs`).
- Prefer more descriptive names than `k` and `v` when iterating with `pairs`, unless you are writing a function that operates on generic tables.
- Use `_` for ignored variables (e.g. in for loops:)

```
for _, item in ipairs(items) do
   do_something_with_item(item)
end
```

- Variables and function names should use `snake_case`.

```
-- bad
local OBJEcttssss = {}
local thisIsMyObject = {}
local c = function()
   -- ...stuff...
end

-- good
local this_is_my_object = {}

local function do_that_thing()
   -- ...stuff...
end
```

> **Rationale:** The standard library uses lowercase APIs, with `joinedlowercase` names, but this does not scale too well for more complex APIs. `snake_case` tends to look good enough and not too out-of-place along side the standard APIs.

- When doing OOP, classes should use `CamelCase`. Acronyms (e.g. XML) should only uppercase the first letter (`XmlDocument`). Methods use `snake_case` too. In LuaRocks, this is used in the `Api` object in the `luarocks.upload.api` module.

```
for _, name in pairs(names) do
   -- ...stuff...
end
```

- Prefer using `is_` when naming boolean functions:

```lua
-- bad
local function evil(alignment)
   return alignment < 100
end

-- good
local function is_evil(alignment)
   return alignment < 100
end
```

- `UPPER_CASE` is to be used sparingly, with "constants" only.

  > **Rationale:** "Sparingly", since Lua does not have real constants. This notation is most useful in libraries that bind C libraries, when bringing over constants from C.

- Do not use uppercase names starting with `_`, they are reserved by Lua.

Tables

- When creating a table, prefer populating its fields all at once, if possible:

```lua
local player = {
   name = "Jack",
   class = "Rogue",
}
```

- You can add a trailing comma to all fields, including the last one.

  > **Rationale:** This makes the structure of your tables more evident at a glance. Trailing commas make it quicker to add new fields and produces shorter diffs.

- Use plain `key` syntax whenever possible, use `["key"]` syntax when using names that can't be represented as identifiers and avoid mixing representations in a declaration:

```lua
table = {
   ["1394-E"] = val1,
   ["UTF-8"] = val2,
   ["and"] = val2,
}
```

Strings

- Use `"double quotes"` for strings; use `'single quotes'` when writing strings that contain double quotes.

```lua
local name = "LuaRocks"
local sentence = 'The name of the program is "LuaRocks"'
```

> **Rationale:** Double quotes are used as string delimiters in a larger number of programming languages. Single quotes are useful for avoiding escaping when using double quotes in literals.

Line lengths

- There are no hard or soft limits on line lengths. Line lengths are naturally limited by using one statement per line. If that still produces lines that are too long (e.g. an expression that produces a line over 256-characters long, for example), this means the expression is too complex and would do better split into subexpressions with reasonable names.

> **Rationale:** No one works on VT100 terminals anymore. If line lengths are a proxy for code complexity, we should address code complexity instead of using line breaks to fit mind-bending statements over multiple lines.

Function declaration syntax

- Prefer function syntax over variable syntax. This helps differentiate between named and anonymous functions.

```lua
-- bad
local nope = function(name, options)
   -- ...stuff...
end

-- good
local function yup(name, options)
   -- ...stuff...
end
```

- Perform validation early and return as early as possible.

```lua
-- bad
local function is_good_name(name, options, arg)
   local is_good = #name > 3
   is_good = is_good and #name < 30

   -- ...stuff...

   return is_good
end

-- good
local function is_good_name(name, options, args)
   if #name < 3 or #name > 30 then
      return false
   end

   -- ...stuff...

   return true
end
```

Function calls

- Even though Lua allows it, do not omit parenthesis for functions that take a unique string literal argument.

```lua
-- bad
local data = get_data"KRP"..tostring(area_number)
-- good
local data = get_data("KRP"..tostring(area_number))
local data = get_data("KRP")..tostring(area_number)
```

> **Rationale:** It is not obvious at a glace what the precedence rules are when omitting the parentheses in a function call. Can you quickly tell which of the two "good" examples in equivalent to the "bad" one? (It's the second one).

- You should not omit parenthesis for functions that take a unique table argument on a single line. You may do so for table arguments that span several lines.

```
local an_instance = a_module.new {
   a_parameter = 42,
   another_parameter = "yay",
}
```

> **Rationale:** The use as in `a_module.new` above occurs alone in a statement, so there are no precedence issues.

Table attributes

- Use dot notation when accessing known properties.

```
local luke = {
   jedi = true,
   age = 28,
}

-- bad
local is_jedi = luke["jedi"]

-- good
local is_jedi = luke.jedi
```

- Use subscript notation `[]` when accessing properties with a variable or if using a table as a list.

```
local vehicles = load_vehicles_from_disk("vehicles.dat")

if vehicles["Porsche"] then
   porsche_handler(vehicles["Porsche"])
   vehicles["Porsche"] = nil
end
for name, cars in pairs(vehicles) do
   regular_handler(cars)
end
```

> **Rationale:** Using dot notation makes it clearer that the given key is meant to be used as a record/object field.

Functions in tables

- When declaring modules and classes, declare functions external to the table definition:

```
local my_module = {}

function my_module.a_function(x)
   -- code
end
```

- When declaring metatables, declare function internal to the table definition.

```
local version_mt = {
   __eq = function(a, b)
      -- code
   end,
   __lt = function(a, b)
      -- code
   end,
}
```

> **Rationale:** Metatables contain special behavior that affect the tables they're assigned (and are used implicitly at the call site), so it's good to be able to get a view of the complete behavior of the metatable at a glance.

This is not as important for objects and modules, which usually have way more code, and which don't fit in a single screen anyway, so nesting them inside the table does not gain much: when scrolling a longer file, it is more evident that `check_version` is a method of `Api` if it says `function Api:check_version()` than if it says `check_version = function()` under some indentation level.

Variable declaration

- Always use `local` to declare variables.

```
-- bad
superpower = get_superpower()

-- good
local superpower = get_superpower()
```

> **Rationale:** Not doing so will result in global variables to avoid polluting the global namespace.

Variable scope

- Assign variables with the smallest possible scope.

```lua
-- bad
local function good()
   local name = get_name()

   test()
   print("doing stuff..")

   --...other stuff...

   if name == "test" then
      return false
   end

   return name
end

-- good
local bad = function()
   test()
   print("doing stuff..")

   --...other stuff...

   local name = get_name()

   if name == "test" then
      return false
   end

   return name
end
```

> **Rationale:** Lua has proper lexical scoping. Declaring the function later means that its scope is smaller, so this makes it easier to check for the effects of a variable.

Conditional expressions

- False and nil are falsy in conditional expressions. Use shortcuts when you can, unless you need to know the difference between false and nil.

```lua
-- bad
if name ~= nil then
   -- ...stuff...
end

-- good
if name then
   -- ...stuff...
end
```

- Avoid designing APIs which depend on the difference between `nil` and `false`.
- Use the `and` / `or` idiom for the pseudo-ternary operator when it results in more straightforward code. When nesting expressions, use parentheses to make it easier to scan visually:

```lua
local function default_name(name)
   -- return the default "Waldo" if name is nil
   return name or "Waldo"
end

local function brew_coffee(machine)
   return (machine and machine.is_loaded) and "coffee brewing" or "fill your water"
end
```

Note that the `x and y or z` as a substitute for `x ? y : z` does not work if `y` may be `nil` or `false` so avoid it altogether for returning booleans or values which may be nil.

Blocks

- Use single-line blocks only for `then return`, `then break` and `function return` (a.k.a "lambda") constructs:

```lua
-- good
if test then break end

-- good
if not ok then return nil, "this failed for this reason: " .. reason end

-- good
use_callback(x, function(k) return k.last end)

-- good
if test then
  return false
end

-- bad
if test < 1 and do_complicated_function(test) == false or seven == 8 and nine == 10 then
do_other_complicated_function() end

-- good
if test < 1 and do_complicated_function(test) == false or seven == 8 and nine == 10 then
   do_other_complicated_function()
   return false
end
```

- Separate statements onto multiple lines. Do not use semicolons as statement terminators.

```lua
-- bad
local whatever = "sure";
a = 1; b = 2

-- good
local whatever = "sure"
a = 1
b = 2
```

Spacing

- Use a space after `--`.

```
--bad
-- good
```

- Always put a space after commas and between operators and assignment signs:

```
-- bad
local x = y*9
local numbers={1,2,3}
numbers={1 , 2 , 3}
numbers={1 ,2 ,3}
local strings = { "hello"
                , "Lua"
                , "world"
                }
dog.set( "attr",{
  age="1 year",
  breed="Bernese Mountain Dog"
})

-- good
local x = y * 9
local numbers = {1, 2, 3}
local strings = {
   "hello",
   "Lua",
   "world",
}
dog.set("attr", {
   age = "1 year",
   breed = "Bernese Mountain Dog",
})
```

- Indent tables and functions according to the start of the line, not the construct:

```
-- bad
local my_table = {
                    "hello",
                    "world",
                 }
using_a_callback(x, function(...)
                      print("hello")
                    end)

-- good
local my_table = {
   "hello",
   "world",
}
using_a_callback(x, function(...)
   print("hello")
end)
```

> **Rationale:** This keep indentation levels aligned at predictable places. You don't need to realign the entire block if something in the first line changes (such as replacing `x` with `xy` in the `using_a_callback` example above).

- The concatenation operator gets a pass for avoiding spaces:

```lua
-- okay
local message = "Hello, "..user.."! This is your day # "..day.." in our platform!"
```

> **Rationale:** Being at the baseline, the dots already provide some visual spacing.

- No spaces after the name of a function in a declaration or in its arguments:

```lua
-- bad
local function hello ( name, language )
   -- code
end

-- good
local function hello(name, language)
   -- code
end
```

- Add blank lines between functions:

```lua
-- bad
local function foo()
   -- code
end
local function bar()
   -- code
end

-- good
local function foo()
   -- code
end

local function bar()
   -- code
end
```

- Avoid aligning variable declarations:

```lua
-- bad
local a               = 1
local long_identifier = 2

-- good
local a = 1
local long_identifier = 2
```

> **Rationale:** This produces extra diffs which add noise to `git blame`.

- Alignment is occasionally useful when logical correspondence is to be highlighted:

```lua
-- okay
sys_command(form, UI_FORM_UPDATE_NODE, "a",      FORM_NODE_HIDDEN,  false)
sys_command(form, UI_FORM_UPDATE_NODE, "sample", FORM_NODE_VISIBLE, false)
```

Typing

- In non-performance critical code, it can be useful to add type-checking assertions for function arguments:

```
function manif.load_manifest(repo_url, lua_version)
   assert(type(repo_url) == "string")
   assert(type(lua_version) == "string" or not lua_version)

   -- ...
end
```

> **Rationale:** This is a practice adopted early on in the development of LuaRocks that has shown to be beneficial in many occasions.

- Use the standard functions for type conversion, avoid relying on coercion:

```
-- bad
local total_score = review_score .. ""

-- good
local total_score = tostring(review_score)
```

Errors

- Functions that can fail for reasons that are expected (e.g. I/O) should return `nil` and a (string) error message on error, possibly followed by other return values such as an error code.
- On errors such as API misuse, an error should be thrown, either with `error()` or `assert()`.

Modules

Follow these guidelines for writing modules. In short:

- Always require a module into a local variable named after the last component of the module's full name.

```
local bar = require("foo.bar") -- requiring the module

bar.say("hello") -- using the module
```

- Don't rename modules arbitrarily:

```
-- bad
local skt = require("socket")
```

> **Rationale:** Code is much harder to read if we have to keep going back to the top to check how you chose to call a module.

- Start a module by declaring its table using the same all-lowercase local name that will be used to require it. You may use an LDoc comment to identify the whole module path.

```
--- @module foo.bar
local bar = {}
```

- Try to use names that won't clash with your local variables. For instance, don't name your module something like “size”.
- Use `local function` to declare *local* functions only: that is, functions that won’t be accessible from outside the module.

That is, `local function helper_foo()` means that `helper_foo` is really local.

- Public functions are declared in the module table, with dot syntax:

```
function bar.say(greeting)
   print(greeting)
end
```

> **Rationale:** Visibility rules are made explicit through syntax.

- Do not set any globals in your module and always return a table in the end.
- If you would like your module to be used as a function, you may set the `__call` metamethod on the module table instead.

> **Rationale:** Modules should return tables in order to be amenable to have their contents inspected via the Lua interactive interpreter or other tools.

- Requiring a module should cause no side-effect other than loading other modules and returning the module table.
- A module should not have state (this still needs to be fixed for some LuaRocks modules). If a module needs configuration, turn it into a factory. For example, do not make something like this:

```
-- bad
local mp = require "MessagePack"
mp.set_integer("unsigned")
```

and do something like this instead:

```
-- good
local messagepack = require("messagepack")
local mpack = messagepack.new({integer = "unsigned"})
```

- The invocation of require should look like a regular Lua function call, because it is.

```
-- bad
local bla = require "bla"

-- good
local bla = require("bla")
```

> **Rationale:** This makes it explicit that require is a function call and not a keyword. Many other languages employ keywords for this purpose, so having a "special syntax" for require would trip up Lua newcomers.

OOP

- Create classes like this:

```lua
--- @module myproject.myclass
local myclass = {}

-- class table
local MyClass = {}

function MyClass:some_method()
   -- code
end

function MyClass:another_one()
   self:some_method()
   -- more code
end

function myclass.new()
   local self = {}
   setmetatable(self, { __index = MyClass })
   return self
end

return myclass
```

- The class table and the class metatable should both be local. If containing metamethods, the metatable may be declared as a top-level local, named `MyClass_mt`.

> **Rationale:** It's easy to see in the code above that the functions with `MyClass` in their signature are methods. A deeper discussion of the design rationale for this is found here.

- Use the method notation when invoking methods:

```lua
-- bad
my_object.my_method(my_object)
-- good
my_object:my_method()
```

> **Rationale:** This makes it explicit that the intent is to use the function as an OOP method.

- Do not rely on the `__gc` metamethod to release resources other than memory. If your object manage resources such as files, add a `close` method to their APIs and do not auto-close via `__gc`. Auto-closing via `__gc` would entice users of your module to not close resources as soon as possible. (Note that the standard `io` library does not follow this recommendation, and users often forget that not closing files immediately can lead to "too many open files" errors when the program runs for a while.)

> **Rationale:** The garbage collector performs automatic *memory* management, dealing with memory only. There is no guarantees as to when the garbage collector will be invoked, and memory pressure does not correlate to pressure on other resources.

File structure

- Lua files should be named in all lowercase.
- Lua files should be in a top-level `src` directory. The main library file should be called `modulename.lua`.
- Tests should be in a top-level `spec` directory. LuaRocks uses Busted for testing.
- Executables are in `src/bin` directory.

Static checking

It's best if code passes luacheck. If it does not with default settings, it should provide a `.luacheckrc` with sensible exceptions.

- luacheck warnings of class 6xx refer to whitespace issues and can be ignored. Do not send pull requests "fixing" trailing whitespaces.

> **Rationale:** Git is paranoid about trailing whitespace due to the patch-file email-based workflow inherited from the Linux kernel mailing list. When using the Git tool proper, exceeding whitespace makes no difference whatsoever except for being highlighted by Git's coloring (for the aforementioned reasons). Git's pedantism about it has spread over the year to the syntax highlighting of many text editors and now everyone says they hate trailing whitespace without being really able to answer why (the actual cause being that tools complain to them about it, for no good reason).

- luacheck warnings of class 211, 212, 213 (unused variable, argument or loop variable) should be ignored, if the unused variable was added explicitly: for example, sometimes it is useful, for code understandability, to spell out what the keys and values in a table are, even if you're only using one of them. Another example is a function that needs to follow a given signature for API reasons (e.g. a callback that follows a given format) but doesn't use some of its arguments; it's better to spell out in the argument what the API the function implements is, instead of adding `_` variables.

- luacheck warning 542 (empty if branch) can also be ignored, when a sequence of `if` / `elseif` / `else` blocks implements a "switch/case"-style list of cases, and one of the cases is meant to mean "pass". For example:

```
if warning >= 600 and warning <= 699 then
   print("no whitespace warnings")
elseif warning == 542 then
   -- pass
else
   print("got a warning: "..warning)
end
```

> **Rationale:** This avoids writing negated conditions in the final fallback case, and it's easy to add another case to the construct without having to edit the fallback.