with-chatgpt

help me distinguish between pure lua programming apis and neovim lua apis

help me learn lua language by code examples

help me know neovim lua apis by code examples

imagine we make neovim plugin with c-program as backend, explain to me

---

Here's a clean way to understand the difference between **pure Lua programming APIs** and **Neovim Lua APIs** — essentially, what belongs to the language itself versus what belongs to the editor environment.

🟦 1. Pure Lua Programming APIs

These are the **core features** provided by the **Lua language and standard library itself**. They exist **independently of Neovim** and can run anywhere Lua runs: CLI, embedded systems, games, standalone scripts, etc.

✔️ Examples of pure Lua APIs

Language features

- `local`, `function`, `table`, `string`, `number`
- Metatables / metamethods (`__index`, `__add`, etc.)
- Coroutines (`coroutine.create`, `coroutine.yield`)
- Modules (via `require`)

Standard libraries

- `string` (e.g., `string.gsub`, `string.match`)
- `table` (e.g., `table.insert`, `table.sort`)
- `math` (e.g., `math.random`, `math.floor`)
- `io` (e.g., `io.open`, `io.read`)
- `os` (e.g., `os.date`, `os.execute`)
- `debug` (low-level debug access)

Where they run

- Any Lua interpreter (5.1 / 5.3 / LuaJIT)
- Game engines (Love2D, Roblox Lua variants, Defold, etc.)
- Embedded systems
- Standalone Lua scripts

Key idea: Pure Lua = **General-purpose language, no editor knowledge**.

🟩 2. Neovim Lua APIs

These are Lua functions **provided by Neovim**, not by the Lua language. You cannot use them outside Neovim because they're bindings into the editor's internal functionality.

Think of them as an **editor API exposed to Lua**.

✔️ Common Neovim Lua API namespaces

`vim` (global namespace)

- Entry point to all Neovim features accessible from Lua.

Core modules:

- `vim.api` → **core editor API** (buffer, window, tabpage operations)
- `vim.loop` → libuv event loop interface
- `vim.fn` → call Vimscript functions from Lua
- `vim.keymap` → keymap creation helpers
- `vim.cmd` → run raw commands (`:commands`)
- `vim.opt` → set/get options
- `vim.o` , `vim.bo` , `vim.wo` → global/buffer/window options
- `vim.lsp` → LSP client framework
- `vim.diagnostic` → diagnostics management
- `vim.ui` → Neovim UI helpers (select, input)
- `vim.fs` → filesystem helpers (introduced in recent versions)

✔️ Examples of Neovim-only APIs

Buffers & windows

```
vim.api.nvim_get_current_buf()
vim.api.nvim_buf_set_lines(buf, 0, -1, false, lines)
vim.api.nvim_win_set_cursor(0, {10, 0})
```

Keymaps

```
vim.keymap.set("n", "qq", ":q<CR>")
```

Autocommands

```
vim.api.nvim_create_autocmd("BufWritePost", {
  pattern = "*.lua",
  callback = function() print("Saved!") end,
})
```

LSP

```
vim.lsp.buf.format()
vim.lsp.start_client({...})
```

Diagnostics

```
vim.diagnostic.set(namespace, bufnr, items)
```

UI elements

```lua
vim.ui.select({"one","two"}, {}, function(choice) end)
```

These do not exist in pure Lua.

🟥 Key Differences Summarized

Feature	Pure Lua APIs	Neovim Lua APIs
Provided by	Lua language	Neovim editor
Runs outside Neovim	✔️ Yes	❌ No
Editor manipulation	❌ None	✔️ Full control
File/IO	Basic `io` & `os`	Rich FS under `vim.fs`
Event loop	Coroutines	libuv via `vim.loop`
LSP	❌	✔️ Built-in
Keymaps	❌	✔️
Buffers, windows	❌	✔️
Autocommands	❌	✔️

🟨 How to quickly recognize which is which

Belongs to pure Lua if:

- You can find it in the *Lua 5.x reference manual*.
- It starts with: `string.`, `table.`, `math.`, `io.`, `os.`
- It doesn't mention buffers, windows, or Neovim-specific objects.

Belongs to Neovim if:

- It starts with `vim.` anything.
- It controls text, buffers, windows, keymaps, commands.
- It interacts with LSP, diagnostics, or Neovim UI.
- It references libuv (`vim.loop`).

🟪 Quick examples comparing both

Pure Lua example

```lua
local t = {1, 2, 3}
table.insert(t, 4)
print(table.concat(t, ","))
```

Neovim example

```lua
local buf = vim.api.nvim_get_current_buf()
vim.api.nvim_buf_set_lines(buf, 0, -1, false, {"hello", "world"})
```

Pure Lua works anywhere. Neovim Lua works *only inside Neovim*.

If you want, I can also create:

- a flowchart to classify APIs automatically
- a cheat sheet or reference card
- examples of mixing pure Lua + Neovim Lua
- a guide to writing a plugin by separating “editor code” from “logic code”

Just tell me!

Below is a practical, example-driven way to learn Lua. Each section introduces one concept and gives small code samples that you can run directly using any Lua interpreter (Lua 5.1, 5.3, or LuaJIT).

---

1. Variables & Types

Lua has only a few basic types: **nil, number, string, boolean, table, function, userdata, thread**.

```lua
local a = 10
local b = 3.14
local c = "hello"
local d = true
local e = nil

print(a, b, c, d, e)
```

Numbers are always floating-point by default.

---

2. Strings

Strings are immutable. Concatenation uses `..`.

```lua
local name = "Lua"
local msg = "Hello " .. name
print(msg)

print("Length:", #msg)   -- # gives string length
print(string.upper(msg))
print(string.find(msg, "Lua"))
```

---

3. Tables (Lua’s most important structure)

Tables serve as **arrays, dictionaries, objects, structs — everything**.

Array-like

```
local arr = {10, 20, 30}
print(arr[1])   -- Lua arrays start at index 1 by convention
```

Dictionary-like

```
local person = {
  name = "Alice",
  age = 20
}

print(person.name)
print(person["age"])
```

Mutating tables

```
local t = {}
t.x = 5
t.y = 10
table.insert(t, "hello")
```

---

4. If statements

```
local x = 7

if x > 5 then
  print("big")
elseif x == 5 then
  print("exact")
else
  print("small")
end
```

---

5. Loops

Numeric for

```
for i = 1, 5 do
  print(i)
```

```
end
```

Generic for (pairs/ipairs)

```
local t = {10, 20, x=99}

for i, v in ipairs(t) do
  print("array:", i, v)
end

for k, v in pairs(t) do
  print("any:", k, v)
end
```

while

```
local n = 5
while n > 0 do
  print(n)
  n = n - 1
end
```

---

6. Functions

Functions are values and can be passed around.

```
local function add(a, b)
  return a + b
end

print(add(3, 4))
```

Anonymous functions

```
local greet = function(name)
  print("Hello", name)
end

greet("Bob")
```

Functions as arguments

```
local function apply(fn, x)
  return fn(x)
```

```lua
end

print( apply(math.sqrt, 9) )
```

---

7. Closures (important concept!)

Lua supports capturing local variables.

```lua
function counter()
  local n = 0
  return function()
    n = n + 1
    return n
  end
end

local c = counter()
print(c()) -- 1
print(c()) -- 2
print(c()) -- 3
```

---

8. Metatables (powerful feature)

Metatables allow operator overloading and custom behavior.

```lua
local v = {x=1, y=2}

local mt = {
  __add = function(a, b)
    return {x = a.x + b.x, y = a.y + b.y}
  end
}

setmetatable(v, mt)
local result = v + {x=3, y=4}
print(result.x, result.y)
```

---

9. Modules (using `require`)

Create a file `mathutil.lua`:

```lua
local M = {}

function M.double(x)
  return x * 2
```

```lua
end

return M
```

Use it:

```lua
local mathutil = require("mathutil")
print(mathutil.double(5))
```

---

10. Error handling

```lua
local status, err = pcall(function()
  error("something happened")
end)

if not status then
  print("Caught error:", err)
end
```

---

11. Coroutines

Lightweight cooperative threads.

```lua
local co = coroutine.create(function()
  for i = 1, 3 do
    print("inside", i)
    coroutine.yield()
  end
end)

coroutine.resume(co)
coroutine.resume(co)
coroutine.resume(co)
```

---

12. Reading files (standard I/O)

```lua
local f = io.open("data.txt", "r")
local content = f:read("*all")
f:close()

print(content)
```

---

13. Writing files

```lua
local f = io.open("out.txt", "w")
f:write("hello world\n")
f:close()
```

---

14. Simple object-like pattern

Lua has no classes, but you can simulate OOP:

```lua
local Enemy = {}
Enemy.__index = Enemy

function Enemy:new(x, y)
  return setmetatable({x = x, y = y}, Enemy)
end

function Enemy:move(dx, dy)
  self.x = self.x + dx
  self.y = self.y + dy
end

local e = Enemy:new(0, 0)
e:move(5, 3)
print(e.x, e.y)
```

---

15. Pattern matching (Lua's mini-regex)

```lua
local s = "abc123xyz"

for numbers in string.gmatch(s, "%d+") do
  print("found:", numbers)
end
```

Want a next step?

I can help you with:

✅ beginner → intermediate roadmap ✅ exercises with increasing difficulty ✅ project ideas to practice pure Lua ✅ comparison with Neovim's Lua API (so you know which is which) ✅ build a mini Lua-based CLI tool or game

Just tell me the direction!

Here is a **practical, example-driven guide** to learning the **Neovim Lua API**, organized by the most useful API families. Each section contains small code snippets you can paste directly into:

- `:lua <code>`
- your `init.lua`
- or a plugin file

All examples run **only inside Neovim**, because they rely on the `vim` API.

---

1. `vim.api` – Core Neovim functions

This is the "raw" API layer. Everything else is built on top of it.

Get current buffer

```lua
local buf = vim.api.nvim_get_current_buf()
print("Current buffer:", buf)
```

Get & set lines

```lua
local buf = vim.api.nvim_get_current_buf()

local lines = vim.api.nvim_buf_get_lines(buf, 0, -1, false)
print(vim.inspect(lines))

vim.api.nvim_buf_set_lines(buf, 0, 1, false, {"Replaced first line!"})
```

Create autocommand

```lua
vim.api.nvim_create_autocmd("BufWritePost", {
  pattern = "*.lua",
  callback = function()
    print("Saved a Lua file!")
  end,
})
```

Create keymap (raw API)

```lua
vim.api.nvim_set_keymap("n", "<leader>x", ":echo 'hi'<CR>", { noremap = true })
```

---

2. `vim.keymap.set` – Best modern way to map keys

Cleaner and recommended.

```lua
vim.keymap.set("n", "<leader>p", function()
  print("Pressed leader+p!")
end)
```

Any mode

```lua
vim.keymap.set({"n", "v"}, "<leader>y", '"+y')
```

---

3. `vim.fn` – Call Vimscript functions from Lua

Use this when you need Vimscript built-ins.

Call a Vimscript function

```lua
local cwd = vim.fn.getcwd()
print("Current directory:", cwd)
```

Use `expand()`

```lua
print(vim.fn.expand("%:p"))  -- full path of current file
```

---

4. `vim.cmd` – Execute Vimscript commands

```lua
vim.cmd("echo 'hello from Lua'")
vim.cmd("write")     -- same as :write
```

Multiline commands

```lua
vim.cmd([[
  set number
  set relativenumber
]])
```

---

5. `vim.opt` – Set/get options cleanly

```lua
vim.opt.number = true
vim.opt.tabstop = 4
vim.opt.expandtab = false
```

---

6. `vim.o`, `vim.bo`, `vim.wo`

Global, buffer, window scoped options.

Global option

```
print(vim.o.background)
```

Buffer option

```
vim.bo.shiftwidth = 2
```

Window option

```
vim.wo.cursorline = true
```

---

7. `vim.loop` – libuv event loop

This gives you timers, async FS, pipes, etc.

Timer example

```
local timer = vim.loop.new_timer()

timer:start(1000, 1000, function()
  print("Timer tick!")
end)
```

(Press `<C-c>` in Neovim terminal if it becomes spammy.)

---

8. `vim.ui` – Built-in UI helpers

Select menu

```
vim.ui.select({ "apple", "banana", "cherry" }, {
  prompt = "Pick a fruit",
}, function(choice)
  print("You chose:", choice)
end)
```

Text input box

```lua
vim.ui.input({ prompt = "Your name: " }, function(value)
  print("Hello, " .. (value or ""))
end)
```

---

9. `vim.lsp` – Language Server Protocol client

These APIs integrate with built-in LSP features.

Format buffer

```lua
vim.lsp.buf.format()
```

Hover

```lua
vim.keymap.set("n", "K", vim.lsp.buf.hover)
```

Jump to definition

```lua
vim.keymap.set("n", "gd", vim.lsp.buf.definition)
```

---

10. `vim.diagnostic` – Diagnostics (errors, warnings, etc.)

Print all diagnostics in current buffer

```lua
local diags = vim.diagnostic.get(0)
print(vim.inspect(diags))
```

Navigate diagnostics

```lua
vim.keymap.set("n", "]d", vim.diagnostic.goto_next)
vim.keymap.set("n", "[d", vim.diagnostic.goto_prev)
```

---

11. `vim.fs` – Neovim's modern filesystem API

Scan for files

```lua
for path, _ in vim.fs.dir(".") do
  print(path)
```

```
end
```

Find upward (like searching for .git)

```
local root = vim.fs.find(".git", { upward = true })[1]
print(root)
```

---

12. Create a small plugin-like function

Shows buffers, uses API, creates a command.

```
local function show_buffers()
  local bufs = vim.api.nvim_list_bufs()
  print("Buffers:")
  for _, b in ipairs(bufs) do
    print("-", vim.api.nvim_buf_get_name(b))
  end
end

vim.api.nvim_create_user_command("ShowBuffers", show_buffers, {})
```

Run in Neovim:

```
:ShowBuffers
```

---

13. Example: floating window

Neovim Lua can create custom windows.

```
local buf = vim.api.nvim_create_buf(false, true)
vim.api.nvim_buf_set_lines(buf, 0, -1, false, {"Hello", "Floating window!"})

local width = 30
local height = 5
local opts = {
  relative = "editor",
  width = width,
  height = height,
  col = (vim.o.columns - width) / 2,
  row = (vim.o.lines - height) / 2,
  style = "minimal",
  border = "rounded",
}

vim.api.nvim_open_win(buf, true, opts)
```

---

14. Example: highlight text

```
vim.api.nvim_set_hl(0, "MyHighlight", { fg = "yellow", bg = "blue" })

vim.api.nvim_buf_add_highlight(
  0,                -- current buffer
  -1,               -- namespace
  "MyHighlight",    -- highlight group
  0,                -- line
  0,                -- start column
  -1                -- end column (whole line)
)
```

Want a full learning roadmap?

I can generate:

✅ A progressive Neovim-Lua course ✅ Exercises for each API category ✅ A mini plugin project (tree viewer, popup dialog, code actions UI, etc.) ✅ Cheat sheet summarizing all important APIs ✅ "Pure Lua vs Neovim Lua" comparisons for every example

Tell me what you'd like next!

Here's a clear mental model for how to build a **Neovim plugin where Lua is the frontend and C is the backend**. This is a real, practical architecture used in many advanced Neovim plugins (treesitter, telescope extensions, LSP clients, linters, formatters, etc.)

🟩 Big Picture: How Neovim + Lua + C Work Together

Neovim accepts **Lua** as the primary scripting language. Lua interacts with Neovim through the `vim` API.

But if your plugin needs:

- high-performance computation
- heavy data parsing
- native OS operations
- specialized algorithms (image processing, AST parsing, etc.)
- existing C libraries

...then writing the heavy parts in **C** is the right move.

Lua (your plugin) calls C functions via:

1. **Native shared library loaded by Lua** (`.so` , `.dll` , `.dylib`)
2. **FFI (foreign function interface)** via LuaJIT FFI
3. **External job process** using `vim.loop` or `vim.fn.jobstart`

Below are the three approaches.

🟦 Approach 1: Lua loads a C shared library

(*Fastest. Best for direct function calls.*)

1. C side: create a shared library

Example: `backend.c`

```c
#include <lua.h>
#include <lauxlib.h>

static int sum(lua_State* L) {
    int a = luaL_checkinteger(L, 1);
    int b = luaL_checkinteger(L, 2);
    lua_pushinteger(L, a + b);
    return 1;
}

int luaopen_backend(lua_State* L) {
    luaL_Reg funcs[] = {
        {"sum", sum},
        {NULL, NULL}
    };
    luaL_newlib(L, funcs);
    return 1;
}
```

Compile (Linux/Mac):

```
gcc -shared -o backend.so -fPIC backend.c `pkg-config --cflags --libs lua5.1`
```

2. Lua side: load it like a normal module

```lua
local backend = require("backend")

print( backend.sum(10, 20) )   -- → 30
```

Neovim executes this inside plugins exactly the same way.

🟨 Approach 2: LuaJIT FFI (Neovim uses LuaJIT by default)

Simplest integration. Compile C normally and load symbols.

1. C side: compile normal `.so` exposing symbols

`mathlib.c`

```c
int multiply(int a, int b) {
    return a * b;
}
```

Compile:

```
gcc -shared -o mathlib.so -fPIC mathlib.c
```

2. Lua side using FFI

```lua
local ffi = require("ffi")

ffi.cdef[[
int multiply(int a, int b);
]]

local lib = ffi.load("mathlib")

print(lib.multiply(6, 7))  -- 42
```

This is extremely fast and avoids writing Lua bindings manually.

🟥 Approach 3: C program as external job

(*Best for long-running backend processes, formatters, parsers, LSP-like daemons.*)

Neovim can run your C program as a subprocess, pass data, and read the result.

1. C program: read from stdin, write to stdout

`backend_cli.c`

```c
#include <stdio.h>

int main() {
    char input[256];

    while (fgets(input, sizeof(input), stdin)) {
        printf("Processed: %s", input);
        fflush(stdout);
    }
```

```
    return 0;
}
```

Compile:

```
gcc backend_cli.c -o backend_cli
```

2. Lua side: call it from Neovim

```lua
local uv = vim.loop

local handle, pid
handle = uv.spawn("./backend_cli", {
    stdio = {nil, uv.new_pipe(false), uv.new_pipe(false)}
},
function(code)
    print("Process exited:", code)
    handle:close()
end)

local stdin = handle.stdin
stdin:write("hello\n")
stdin:write("world\n")
stdin:shutdown()
```

This makes your C program behave like a language server or formatter.

🟪 Which approach should you choose?

Approach	Purpose	Pros	Cons
1. Shared library (luaopen)	Direct function calls	Fast, clean, idiomatic	Must write binding code
2. LuaJIT FFI	Direct function calls, minimal boilerplate	Extremely easy; no binding code	Neovim must use LuaJIT (it does)
3. External process	Long-running daemon or complex tools	Works like LSP, async, parallel	Communication overhead

---

🟩 Example Neovim plugin structure

```
lua/myplugin/init.lua
lua/myplugin/ui.lua
lua/myplugin/backend.lua  -- Lua wrapper to C
backend/processor.c
backend/processor.so
```

`init.lua` :

```
local backend = require("myplugin.backend")

vim.api.nvim_create_user_command("Compute", function()
  local result = backend.sum(40, 2)
  print("Result:", result)
end, {})
```

`backend.lua` :

```
local backend = require("processor")
return backend
```

🟦 Adding Neovim API integration

You can combine Neovim APIs with your backend results.

Example: process buffer lines in C:

Lua:

```
local backend = require("processor")

vim.api.nvim_create_user_command("ProcessBuffer", function()
  local buf = vim.api.nvim_get_current_buf()
  local lines = vim.api.nvim_buf_get_lines(buf, 0, -1, false)
  local result = backend.process_lines(lines)  -- C function
  vim.api.nvim_buf_set_lines(buf, 0, -1, false, result)
end, {})
```

C:

- Accept a Lua table of strings
- Process
- Return a new table

This makes your plugin “feel Native”.

🟩 Real Neovim plugins using C backends

Neovim ecosystem already uses this pattern:

Plugin	Backend	Notes
nvim-treesitter	C parser	Shared library via FFI
luv	libuv C library	Core async support

fzy-lua-native	C fuzzy matcher	LuaJIT FFI + shared library
telescope-fzf-native	C fuzzy engine	Compiled and loaded by Lua
luarocks.nvim	C modules	Lua/C hybrid

Learning from their repo is extremely helpful.

🟧 Want to go deeper?

I can help you:

- Create a full sample plugin with real C backend
- Write a C library returning Lua tables
- Wrap C structs in Lua userdata
- Build a full async Neovim backend with libuv
- Make a Treesitter-like parser architecture
- Create a mini-LSP in C and connect via Neovim jobs

Tell me which direction you want next!
