with-ms-copilot

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

---

Great question — this distinction is subtle but super important when you're building ergonomic Neovim tooling. Let's break it down clearly:

🟦 Pure Lua APIs

These are the **standard Lua language and library functions**. They come from the Lua runtime itself, independent of Neovim. If you were writing a standalone Lua script (outside Neovim), you'd still have access to these.

- **Core language constructs**
  - `if`, `for`, `while`, `function`, `local`, `return`
- **Standard libraries**
  - `string` → `string.find`, `string.match`, `string.gsub`
  - `table` → `table.insert`, `table.remove`, `table.concat`
  - `math` → `math.floor`, `math.random`, `math.sin`
  - `os` → `os.date`, `os.time`
  - `io` → `io.open`, `io.read`, `io.write`
- **Metatables & OOP-like features**
  - `setmetatable`, `getmetatable`
- **Coroutines**
  - `coroutine.create`, `coroutine.resume`, `coroutine.yield`

👉 These APIs are **portable**: they'll work in any Lua environment (Neovim, standalone Lua, embedded Lua in other apps).

🟩 Neovim Lua APIs

These are **Neovim-specific extensions** that expose editor functionality to Lua. They don't exist in plain Lua — they're bindings into Neovim's C core.

- **Editor interaction**
  - `vim.api.nvim_command("echo 'hi'")` → run Ex commands
  - `vim.api.nvim_eval("expand('%')")` → evaluate Vimscript expressions
  - `vim.api.nvim_get_current_buf()` → get buffer handle
  - `vim.api.nvim_buf_get_lines(buf, start, end, strict)` → read buffer lines
  - `vim.api.nvim_win_set_cursor(win, {row, col})` → move cursor
- **Configuration & runtime**
  - `vim.opt` → set options (`vim.opt.number = true`)
  - `vim.g` → global variables (`vim.g.mapleader = " "`)

  - `vim.o` → options (like `:set`)
- **Keymaps & commands**
  - `vim.keymap.set("n", "<leader>x", ":write<CR>")`
  - `vim.api.nvim_create_user_command("MyCmd", fn, opts)`
- **Events & autocommands**
  - `vim.api.nvim_create_autocmd("BufWritePost", { callback = fn })`
- **Utilities**
  - `vim.fn` → call Vimscript functions (`vim.fn.getcwd()`)
  - `vim.loop` → libuv event loop (async I/O)
  - `vim.inspect` → pretty-print Lua tables

👉 These APIs are **exclusive to Neovim**: they won't work in standalone Lua, because they depend on Neovim's runtime and editor state.

⚡ Quick Comparison Table

Category	Pure Lua API (portable)	Neovim Lua API (editor-specific)
Strings	`string.gsub`, `string.match`	`vim.fn.expand("%")`
Tables	`table.insert`, `table.concat`	`vim.tbl_map`, `vim.tbl_filter`
Math	`math.floor`, `math.random`	*(none, use pure Lua)*
File I/O	`io.open`, `os.remove`	`vim.api.nvim_buf_get_lines`
Options/Settings	*(none)*	`vim.opt`, `vim.g`, `vim.o`
Commands	*(none)*	`vim.api.nvim_command`, `vim.api.nvim_create_user_command`
Events	`coroutine.resume`	`vim.api.nvim_create_autocmd`
Async I/O	`coroutine`, `os.execute`	`vim.loop` (libuv bindings)

🚀 How to Think About It

- **Pure Lua** = general-purpose programming (data structures, math, string manipulation, file I/O).
- **Neovim Lua** = editor integration (buffers, windows, options, commands, events).

When designing modular CLI backends in C and wrapping them with Lua (your sweet spot!), you'll often use **pure Lua** for glue logic and **Neovim Lua APIs** for editor integration points.

Would you like me to sketch a **blueprint for a plugin architecture** that cleanly separates pure Lua logic from Neovim-specific bindings? That way you can reuse the same backend logic outside Neovim too.

Perfect — learning Lua by example is the fastest way to get comfortable. I'll walk you through **progressively more complex snippets**, each showing a core concept. You can copy-paste these into a Lua REPL (`lua` in your terminal) or a `.lua` file.

🟢 Basics: Variables & Printing

```lua
-- Variables
local name = "Fudzer"
local age = 25

-- Printing
print("Hello, " .. name .. "! You are " .. age .. " years old.")
```

👉 `..` is string concatenation.

🔢 Numbers & Math

```lua
local x = 10
local y = 3

print(x + y)   -- addition
print(x - y)   -- subtraction
print(x * y)   -- multiplication
print(x / y)   -- division
print(x % y)   -- modulus
print(x ^ y)   -- exponentiation
```

📦 Tables (Lua’s main data structure)

```lua
-- Array-like table
local fruits = {"apple", "banana", "cherry"}
print(fruits[1])  -- Lua arrays start at 1

-- Dictionary-like table
local person = {name = "Alice", age = 30}
print(person.name)
print(person["age"])
```

🔄 Loops

```lua
-- Numeric for loop
for i = 1, 5 do
  print("Count: " .. i)
end

-- Generic for loop (pairs)
local colors = {red = "#ff0000", green = "#00ff00"}
for key, value in pairs(colors) do
```

```
  print(key .. " = " .. value)
end
```

🧩 Functions

```
-- Simple function
local function greet(name)
  return "Hello, " .. name
end

print(greet("Lua"))

-- Higher-order function
local function apply_twice(fn, value)
  return fn(fn(value))
end

local function double(x) return x * 2 end
print(apply_twice(double, 5))  -- 20
```

⚙️ Metatables (OOP-like behavior)

```
local Dog = {}
Dog.__index = Dog

function Dog:new(name)
  local obj = {name = name}
  setmetatable(obj, self)
  return obj
end

function Dog:bark()
  print(self.name .. " says woof!")
end

local d = Dog:new("Rex")
d:bark()
```

👉 This is how Lua simulates classes/objects.

🌐 Modules

```
-- mymath.lua
local M = {}

function M.add(a, b) return a + b end
function M.mul(a, b) return a * b end

return M

-- main.lua
local mymath = require("mymath")
print(mymath.add(2, 3))
print(mymath.mul(4, 5))
```

🌀 Coroutines (lightweight threads)

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

coroutine.resume(co) -- step 1
coroutine.resume(co) -- step 2
coroutine.resume(co) -- step 3
```

🚀 Next Steps

- Practice writing small scripts (string manipulation, math utilities, table helpers).
- Try building **pure Lua modules** (like `mymath`) and then later integrate them into Neovim with `vim.api`.
- Explore the **standard library** (`string`, `table`, `math`, `os`, `io`) — they're your toolbox.

Fudzer, since you thrive on modular blueprints: want me to design a **step-by-step learning roadmap** where each stage builds a reusable utility (like a string library, a table toolkit, then a CLI wrapper)? That way you'll learn Lua while building things you can actually reuse in your Neovim/C workflows.

Got it — let's dive straight into **Neovim Lua APIs by code examples**. These are the special functions and modules that let Lua talk directly to Neovim's editor core. I'll organize them by category so you can see the patterns clearly.

🟩 1. Running Commands

```
-- Run an Ex command (like :echo)
vim.api.nvim_command("echo 'Hello from Lua!'")

-- Equivalent to typing :edit foo.txt
vim.api.nvim_command("edit foo.txt")
```

📑 2. Buffers

```lua
-- Get current buffer handle
local buf = vim.api.nvim_get_current_buf()

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

-- Set lines in buffer
vim.api.nvim_buf_set_lines(buf, 0, -1, false, {"Hello", "World"})
```

🪟 3. Windows

```lua
-- Get current window
local win = vim.api.nvim_get_current_win()

-- Move cursor to row 3, column 5
vim.api.nvim_win_set_cursor(win, {3, 5})

-- Get cursor position
local pos = vim.api.nvim_win_get_cursor(win)
print("Row: " .. pos[1] .. ", Col: " .. pos[2])
```

⚙️ 4. Options & Variables

```lua
-- Set options
vim.opt.number = true        -- show line numbers
vim.opt.relativenumber = true

-- Global variables
vim.g.mapleader = " "        -- set <Leader> to space

-- Direct option access
print(vim.o.background)      -- prints "dark" or "light"
```

⌨️ 5. Keymaps

```lua
-- Map <leader>w to save
vim.keymap.set("n", "<leader>w", ":write<CR>")

-- Map <leader>q to quit
vim.keymap.set("n", "<leader>q", ":quit<CR>")
```

📜 6. User Commands

```lua
-- Create a custom command :Hello
vim.api.nvim_create_user_command("Hello", function()
  print("Hello from Neovim Lua!")
end, {})
```

🔔 7. Autocommands

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

🛠️ 8. Vimscript Functions via `vim.fn`

```lua
-- Call Vimscript functions
local cwd = vim.fn.getcwd()
print("Current directory: " .. cwd)

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

🔄 9. Async I/O with `vim.loop` (libuv)

```lua
-- Simple timer
local timer = vim.loop.new_timer()
timer:start(1000, 0, function()
  print("One second passed!")
  timer:stop()
end)
```

🖼️ 10. Utilities

```lua
-- Pretty-print Lua tables
local t = {foo = "bar", baz = {1, 2, 3}}
print(vim.inspect(t))

-- Table helpers
```

```
local mapped = vim.tbl_map(function(x) return x * 2 end, {1, 2, 3})
print(vim.inspect(mapped))  -- {2, 4, 6}
```

🚀 How to Learn by Practice

1. Open Neovim and run `:lua` commands directly. Example:

   ```
   :lua print(vim.fn.getcwd())
   ```

2. Create a Lua file in `~/.config/nvim/lua/` and require it in your `init.lua`.
3. Experiment with buffer/window APIs — they're the most "tangible" way to see changes.

Fudzer, since you're already blueprinting modular CLI backends, would you like me to **design a minimal plugin skeleton** that separates pure Lua logic from Neovim API glue? That way you'll see how to keep your code reusable outside Neovim while still hooking into editor events.

Alright Fudzer, let's blueprint this out step by step. You want a **Neovim plugin that uses a C program as its backend** — essentially a thin Lua wrapper that talks to a fast, modular C utility. This is exactly the kind of ergonomic, composable design you thrive on.

🏗️ Architecture Overview

1. **C Backend (CLI Utility)**
   - Written in C for speed and transparency.
   - Exposes functionality via a command-line interface (stdin/stdout, exit codes, logs).
   - Example: a text formatter, parser, or linter.
2. **Lua Wrapper (Neovim Plugin)**
   - Lua code inside Neovim calls the C program.
   - Uses `vim.fn.system()` or `vim.loop.spawn()` to run the binary.
   - Handles editor integration: buffers, windows, autocommands, keymaps.
3. **User Workflow**
   - User triggers a command/keymap in Neovim.
   - Lua plugin calls the C backend.
   - Output is captured and written back into the buffer or displayed.

🔧 Example: Minimal Setup

1. C Backend (`mytool.c`)

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    if (argc > 1 && strcmp(argv[1], "hello") == 0) {
        printf("Hello from C backend!\n");
```

```
        return 0;
    }
    printf("Unknown command\n");
    return 1;
}
```

Compile:

```
gcc -o mytool mytool.c
```

---

2. Lua Plugin (`lua/myplugin/init.lua`)

```
local M = {}

-- Run C backend via system()
function M.run_backend(cmd)
  local output = vim.fn.system({"./mytool", cmd})
  print(output)
end

-- Expose Neovim command
vim.api.nvim_create_user_command("MyToolHello", function()
  M.run_backend("hello")
end, {})

return M
```

Usage in Neovim:

```
:MyToolHello
```

👉 Prints `Hello from C backend!`

⚡ Advanced Integration

- **Async execution**
  Use `vim.loop.spawn()` for non-blocking calls:

  ```
  vim.loop.spawn("./mytool", {args = {"hello"}}, function(code, signal)
    print("Exit code:", code)
  end)
  ```

- **Buffer manipulation**
  Capture backend output and insert into buffer:

  ```
  local buf = vim.api.nvim_get_current_buf()
  vim.api.nvim_buf_set_lines(buf, 0, -1, false, vim.split(output, "\n"))
  ```

- **Autocommands**
  Run backend automatically on save:

  ```
  vim.api.nvim_create_autocmd("BufWritePost", {
    pattern = "*.txt",
    callback = function()
      M.run_backend("hello")
    end,
  })
  ```

🚀 Why This Design Rocks

- **Speed**: C backend handles heavy lifting.
- **Modularity**: Lua only glues editor state to backend.
- **Reusability**: The C tool can be used outside Neovim (CLI, scripts).
- **Transparency**: Logs and exit codes are preserved.

Fudzer, this is basically the **blueprint for reinventing heavy plugins**: strip them down to a fast C CLI, then wrap with Lua for Neovim ergonomics.

Would you like me to **expand this into a full plugin skeleton** (with directory layout, `init.lua`, and Makefile for the C backend)? That way you'd have a reusable template for all future C-powered Neovim plugins.
