LUA QUICK REFERENCE

Tables, functions, metatables, coroutines, modules, patterns

Basics

Hello World

```
print("Hello, Lua!")
```

Variables & Assignment

```
local name = "Lua"      -- local variable
x = 10                  -- global (avoid)
local a, b = 1, 2       -- multiple assignment
a, b = b, a             -- swap values
```

Comments

```
-- single line comment
--[[ multi-line
     comment ]]
```

Operators

`+ - * / %`	Arithmetic operators
`//`	Floor division (5.3+)
`^`	Exponentiation
`..`	String concatenation
`#`	Length operator
`== ~=`	Equal / not equal
`and or not`	Logical operators

Types

Data Types

`nil`	Absence of value; falsy
`boolean`	true or false
`number`	Double-precision float (or integer in 5.3+)
`string`	Immutable byte sequence
`table`	Associative array (only compound type)
`function`	First-class closure
`userdata`	C data wrapped for Lua
`thread`	Coroutine handle

Type Checking

```
print(type(42))         -- "number"
print(type("hi"))       -- "string"
print(type(nil))        -- "nil"
print(type({}))         -- "table"
```

Tables

Array-style Tables

```
local fruits = {"apple", "banana", "cherry"}
print(fruits[1])           -- "apple" (1-indexed)
table.insert(fruits, "date")
table.remove(fruits, 2)    -- remove "banana"
print(#fruits)             -- length
```

Dictionary-style Tables

```
local user = {name = "Alice", age = 30}
user.email = "a@b.com"    -- add field
user["name"] = "Bob"      -- bracket access
user.age = nil            -- remove field
```

Table Functions

`table.insert(t, v)`	Append value to array
`table.insert(t, i, v)`	Insert at position i
`table.remove(t, i)`	Remove element at position i
`table.sort(t [,cmp])`	Sort array in-place
`table.concat(t, sep)`	Join array elements into string
`table.move(t,a,b,c)`	Move elements from a..b to position c

Functions

Function Definition

```
local function add(a, b)
  return a + b
end
local mul = function(a, b) return a * b end
print(add(2, 3))  -- 5
```

Variadic & Multiple Returns

```
local function sum(...)
  local s = 0
  for _, v in ipairs({...}) do s = s + v end
  return s
end
local function swap(a, b) return b, a end
local x, y = swap(1, 2)
```

Closures

```
local function counter()
  local n = 0
  return function()
    n = n + 1; return n
  end
end
local c = counter()
print(c(), c())  -- 1  2
```

Control Flow

Conditionals

```
if x > 0 then
  print("positive")
elseif x == 0 then
  print("zero")
else
  print("negative")
end
```

Loops

```
for i = 1, 10 do print(i) end
for i = 10, 1, -1 do print(i) end
for k, v in pairs(tbl) do print(k, v) end
for i, v in ipairs(arr) do print(i, v) end
```

While & Repeat

```
while x > 0 do x = x - 1 end
repeat
  x = x + 1
until x >= 10
```

Strings

String Functions

`string.len(s) / #s`	String length in bytes
`string.sub(s, i, j)`	Substring from i to j
`string.upper(s)`	Convert to uppercase
`string.lower(s)`	Convert to lowercase
`string.rep(s, n)`	Repeat string n times
`string.reverse(s)`	Reverse string
`string.format(fmt, ...)`	Printf-style formatting
`string.find(s, pat)`	Find pattern, return indices
`string.gsub(s, pat, rep)`	Global substitution
`string.gmatch(s, pat)`	Iterator over pattern matches

Pattern Characters

`.`	Any character
`%a / %A`	Letters / non-letters
`%d / %D`	Digits / non-digits
`%w / %W`	Alphanumeric / non-alphanumeric
`%s / %S`	Whitespace / non-whitespace
`%p`	Punctuation
`* + - ?`	Greedy, greedy, lazy, optional

Metatables

Setting Metatables

```
local mt = {}
mt.__add = function(a, b)
  return {val = a.val + b.val}
end
local a = setmetatable({val=1}, mt)
local b = setmetatable({val=2}, mt)
local c = a + b  -- c.val == 3
```

Common Metamethods

`__index`	Lookup missing keys (table or function)
`__newindex`	Intercept new key assignment
`__add / __sub / __mul`	Arithmetic operators
`__eq / __lt / __le`	Comparison operators
`__tostring`	Custom string representation
`__len`	Custom # operator
`__call`	Call table as function
`__concat`	Custom .. operator

OOP with Metatables

```
local Dog = {}; Dog.__index = Dog
function Dog.new(name)
  return setmetatable({name=name}, Dog)
end
function Dog:bark() print(self.name.." says Woof") end
local d = Dog.new("Rex"); d:bark()
```

Coroutines

Coroutine Lifecycle

`coroutine.create(f)`	Create coroutine from function
`coroutine.resume(co, ...)`	Start or continue coroutine
`coroutine.yield(...)`	Suspend execution, return values
`coroutine.status(co)`	"running", "suspended", "dead"
`coroutine.wrap(f)`	Create callable coroutine wrapper

Coroutine Example

```
local function gen(max)
  for i = 1, max do coroutine.yield(i) end
end
local co = coroutine.wrap(gen)
print(co(5))  -- 1
print(co())   -- 2
```

Modules

Creating a Module

```
-- mylib.lua
local M = {}
function M.greet(name)
  return "Hello, " .. name
end
return M
```

Using Modules

```
local mylib = require("mylib")
print(mylib.greet("World"))
```

Standard Libraries

`math`	Math functions (sin, random, huge, etc.)
`string`	String manipulation and patterns
`table`	Table manipulation (insert, sort, etc.)
`io`	File I/O operations
`os`	OS facilities (time, clock, execute)
`debug`	Debug interface (use sparingly)

Common Patterns

Ternary Idiom

```
-- Lua has no ternary; use and/or idiom
local val = condition and "yes" or "no"
-- Caution: fails if "yes" is false/nil
```

Safe Table Access

```
local function get(t, ...)
  for _, k in ipairs({...}) do
    if type(t) ~= "table" then return nil end
    t = t[k]
  end
  return t
end
get(config, "db", "host")  -- safe nested access
```

Iterating with ipairs vs pairs

```
-- ipairs: array part, stops at first nil
for i, v in ipairs(arr) do print(i, v) end
-- pairs: all keys (unordered)
for k, v in pairs(tbl) do print(k, v) end
```