Lua and Love2D using Replit – Part 1

All code is written on the Replit online environment, so any computer / OS (Operating System) can be used as long as you have an active internet connection, a keyboard and a mouse.

As the free version of Replit only allows 3 projects you will need to cut/paste code from a completed project to another online source, or save to a USB drive. Examples include Google Docs, Pastebin, Dropbox or Microsoft OneDrive.

You can create a free account on pastebin.com as well. You can then copy/paste your projects between accounts as you please.

Use the browser of your choice and go to https://replit.com/

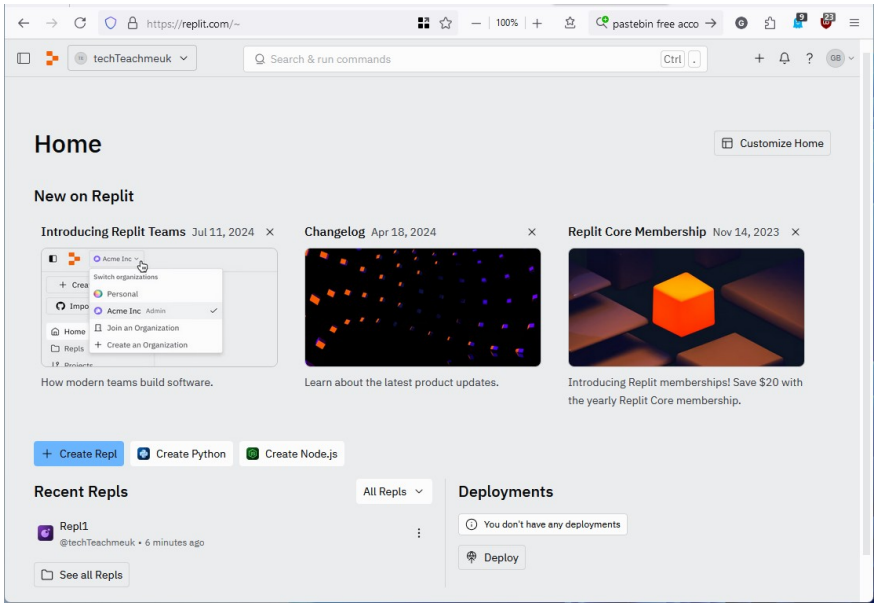
Log in if you already have an account.

If not, click the button "Sign up for free"
There is a guide to creating a new Replit and Pastebin accounts at:

https://github.com/Inksaver/LuaForSchools/tree/main/RecommendedSoftware

Once you have signed in, go to the home page. This will list any Repls you have already created (if any)

To create a new Repl, Click the blue button + Create Repl

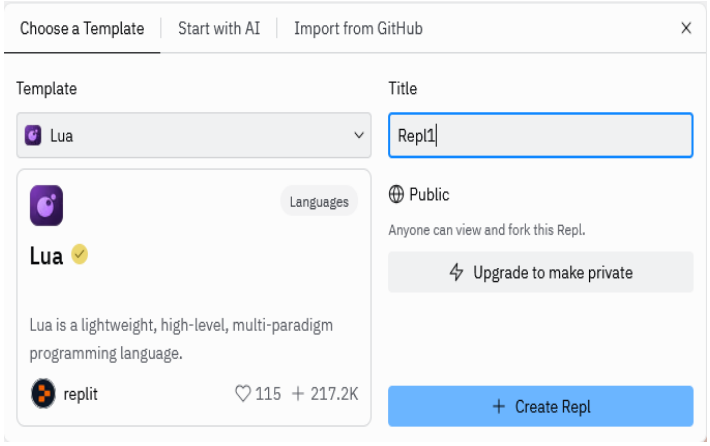


Type Lua into the search bar.

If you have already used Lua it will

come up as a ‘Favourite’

When you select Lua, it will come up with a random name for your project Title

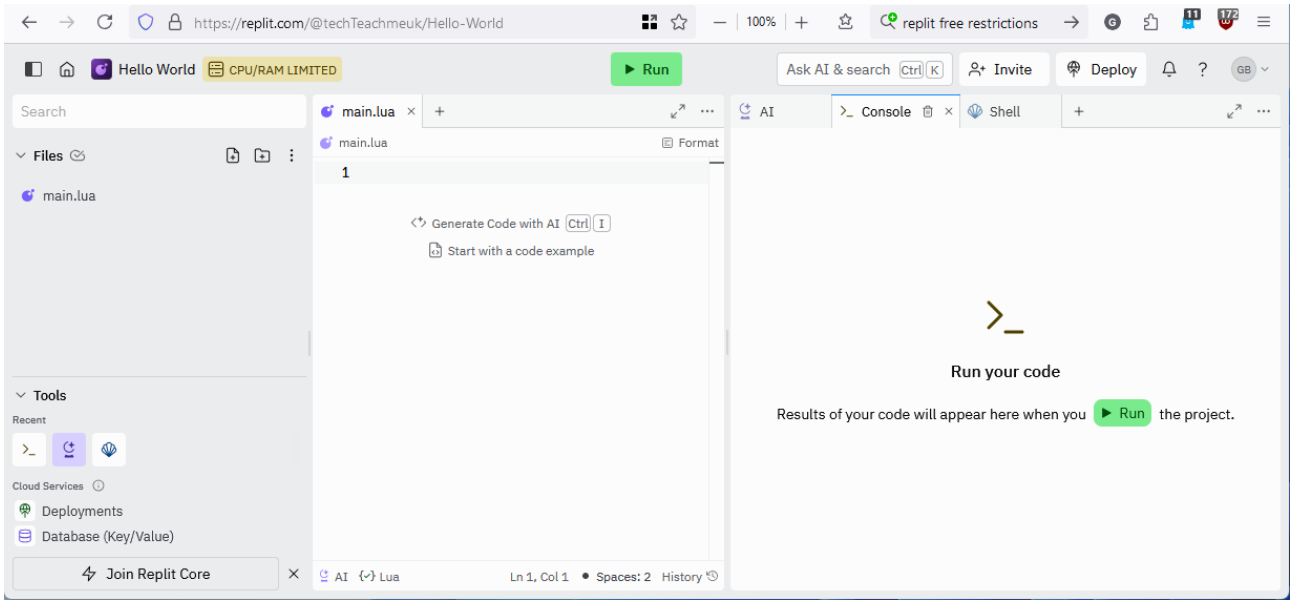


Change the name (e.g. “Repl1”) and click “+ Create Repl”.

On the free version of Replit you can only have 3 projects, so you will have to re-use them. I suggest creating all 3 and call them “Repl1”, “Repl2”, “Repl3”

If you want to edit an existing Repl, just click on it.

You will then see the code window:



DO NOT USE Replit AI

You are learning how to code the hard way. When you are competent, you can then judge any AI code produced to evaluate it properly.

DO NOT Copy/Paste code from Github or other sources unless specifically told to. The process of typing code, trying it, then correcting any errors is how you learn.

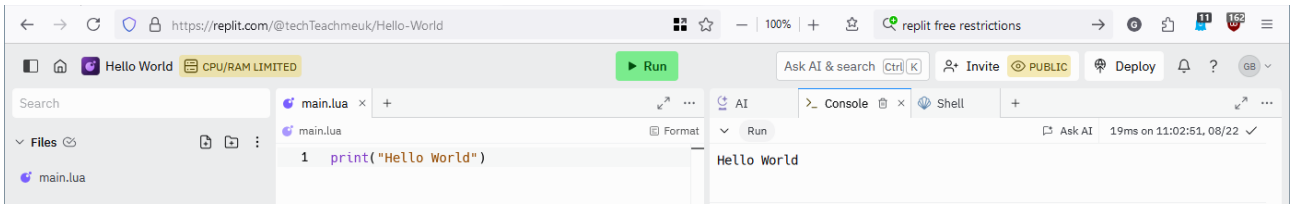
If you copy/paste, run it and it works, what have you learned?

When working through the examples below, re-use a Repl for similar projects, then copy them elsewhere for long-term storage

The Statutory “Hello World” script

Type this into the Editor window: `print("Hello World")`

Click the green triangle “Run”. This will automatically save the code and run it.



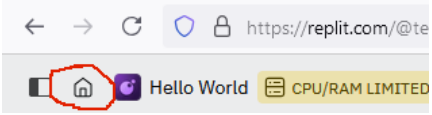
Well done! You have run your first script. It is only one line, but a script can be hundreds or even thousands of lines.

Lua (and Python) are interpreted languages. They read your script line by line, and carry out the instructions you wrote. In this case it ‘print’ s the text ‘Hello World’ to the console (screen).

You will soon be using the Love2D game library, which is heavily based on procedural programming.

Here is an example of using procedural programming, which we will be using right from the start:

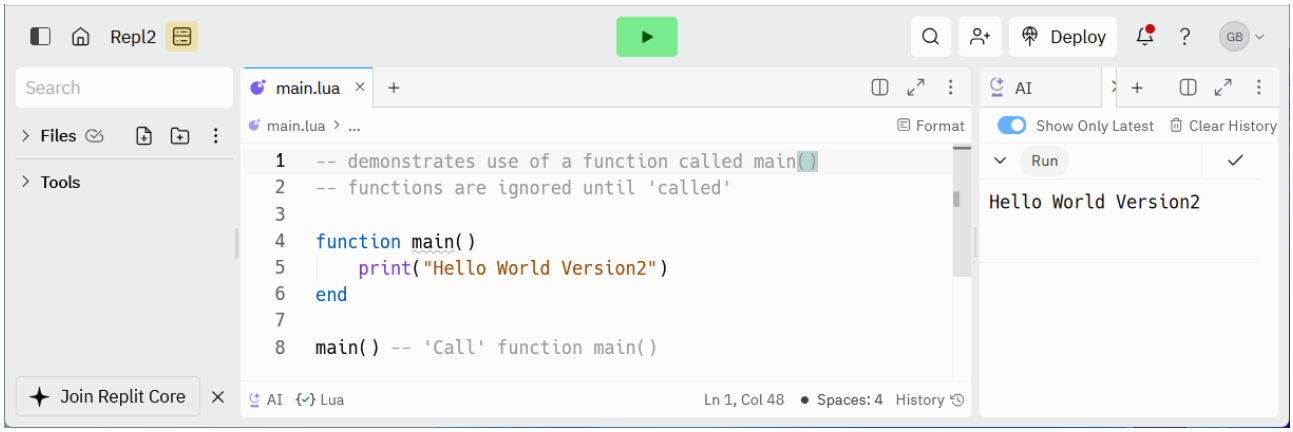
Click the “home” button on replit.com.



Create a new Repl called “Repl2” (if you have not already done so).

Any text starting with - - and appearing in green or light grey in screen shots are comments. They are for information only, and do NOT need to be typed in.

HelloWorld2.lua:



It works exactly the same as before, so why bother?

Procedural programming is much more efficient and can save you a lot of typing.

As your script grows in size, using functions (and procedures) makes it easier to maintain. As the only difference in practical terms between a function and a procedure is that a function returns at least one value, this tutorial will use the word function to cover both.

More advanced languages such as C# and Java start with a function/procedure called main() so it is a good idea to start using a similar function in Lua (and Python).

When the interpreter reads your modified script this time, it ignores any functions, but makes a mental note where they are, so when your script uses or 'calls' them, it knows where to go.
(Python works in the same way using the word 'def' instead of 'function')

When it reaches line 8 `main()`, this is a call to the main() function found at line 4, so the script jumps from line 8, which is the starting point, to line 4.

The main() function has only one line of code: `print("Hello World")`, which it executes, then control is returned back to line 8, which happens to be the end of the program.

All further examples and exercises will use a main() function.

Printing to the screen

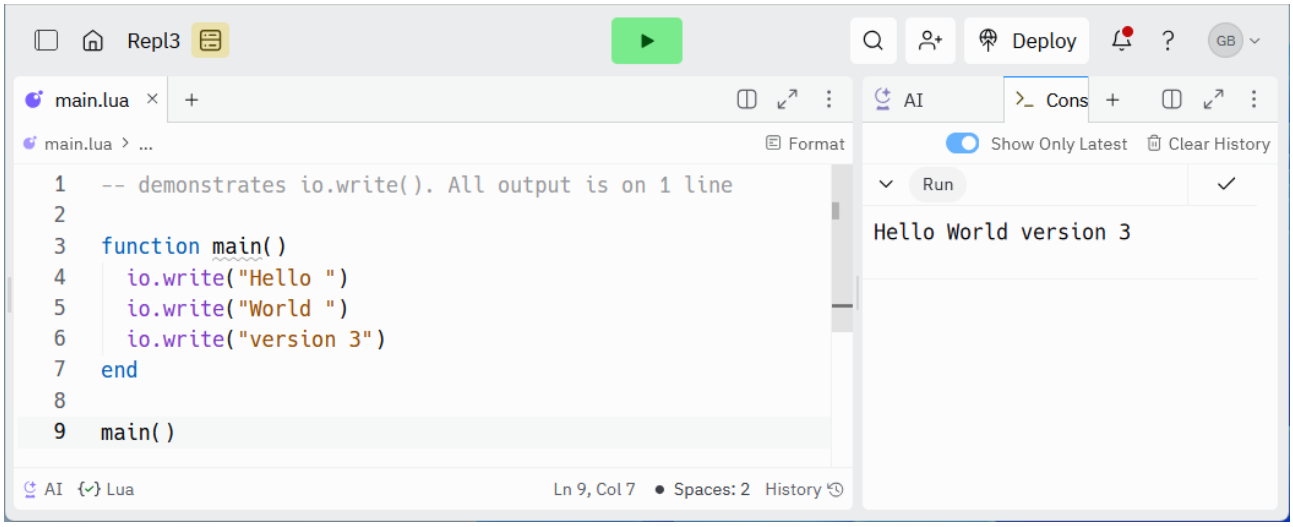
Displaying text either to a panel within the IDE, or to an external console is usually achieved with print() as you have already used.

The print() procedure automatically inserts a “newline” character which forces the output cursor down to the next line, ready to print() again.

There is a variation on this using `io.write()`, which prints to the screen, but does NOT insert a newline character.

Go to Repl3, or create it if you have not already done so.

HelloWorld3.lua:



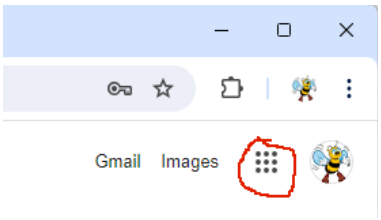
As you now have your 3 file Replit limit, it is time to move them elsewhere.

Ideally use Pastebin, as it is designed specifically for storing code, but a good alternative is Google Docs.

Both methods are shown below:

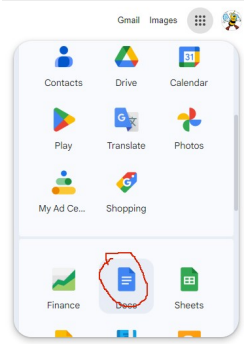
Storing code on Google docs

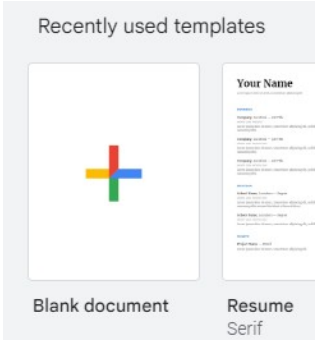
If using Google docs to copy files, here is the method:



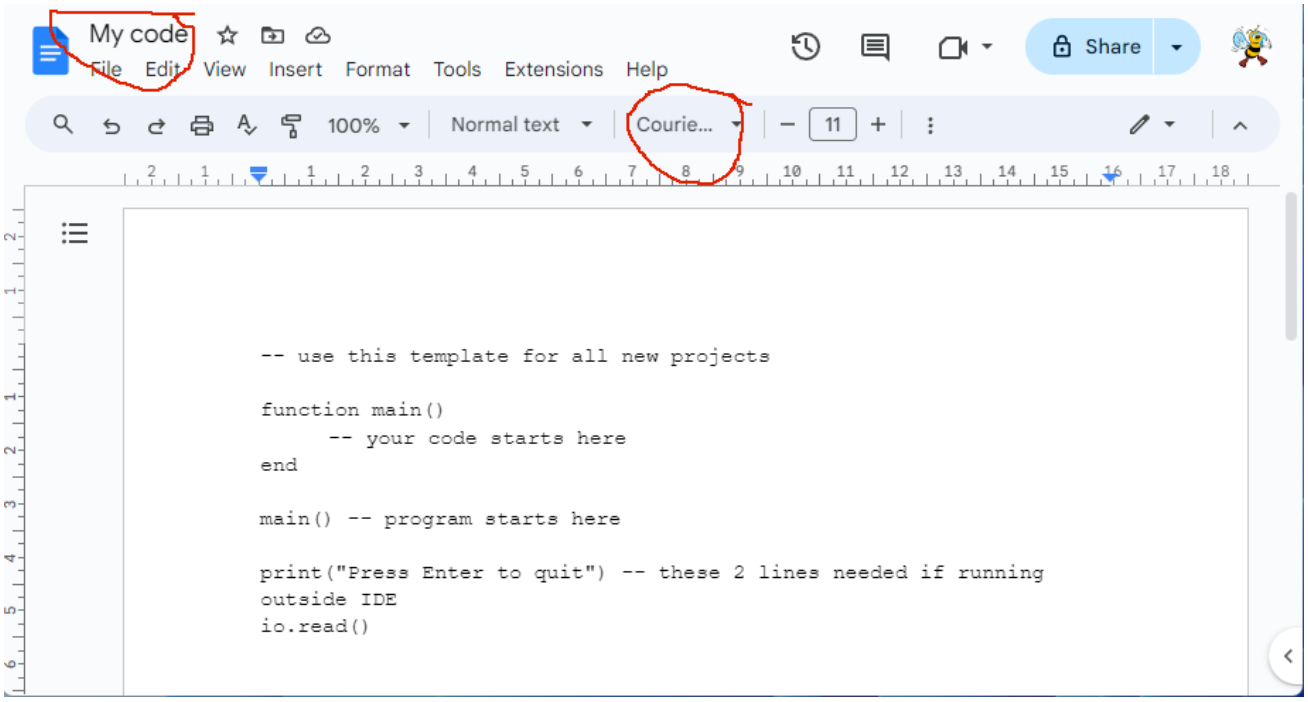
Go to google.com and login to your Google (gmail) account

Click on the “Apps” icon





Click on “Docs”
Click on “Blank Document”
Change the title to “My Code”
Change the font to “Courier new”
Paste your code in.

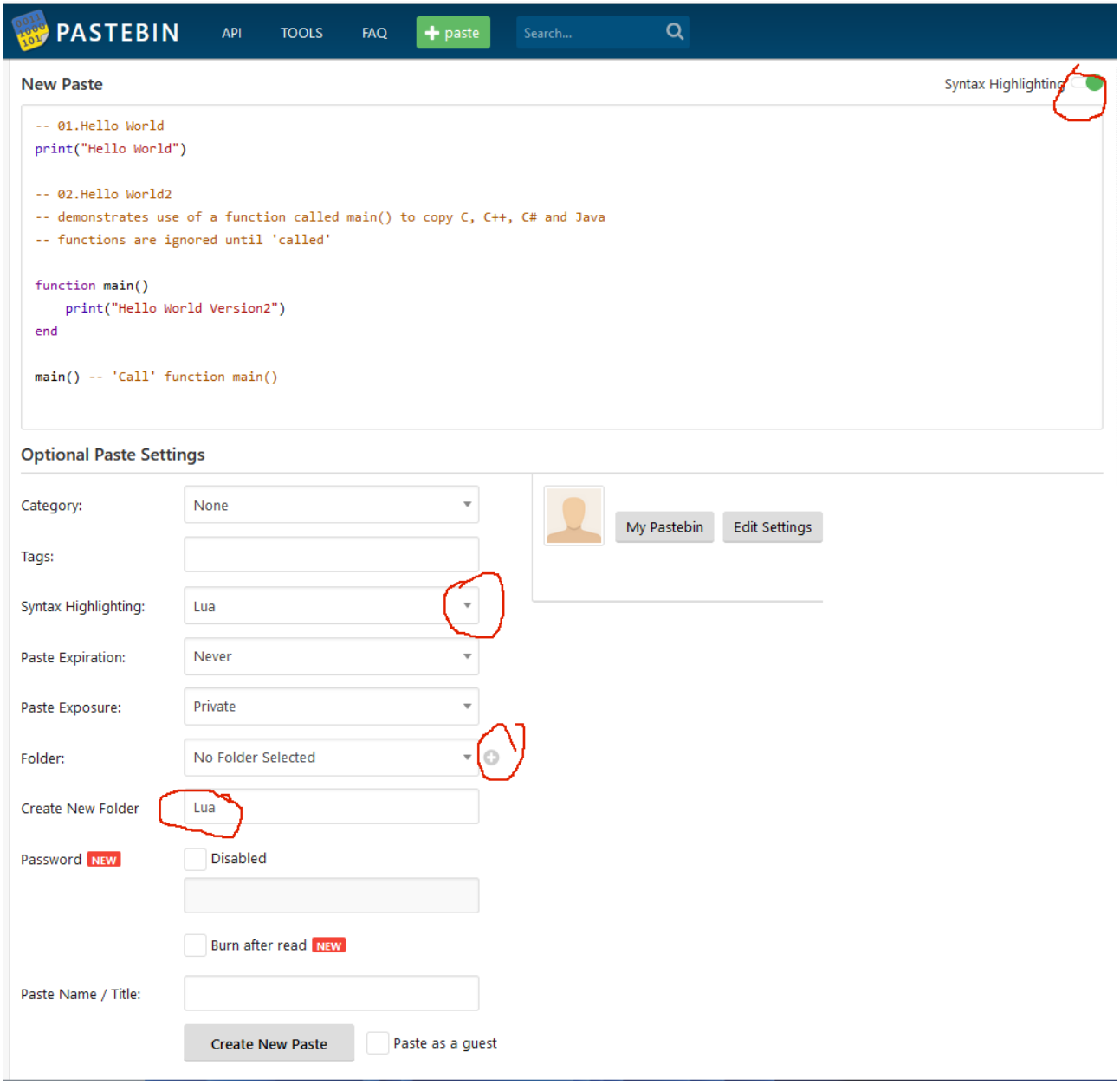


The document auto-saves as you type it. You can close it at any time.

Storing code on Pastebin

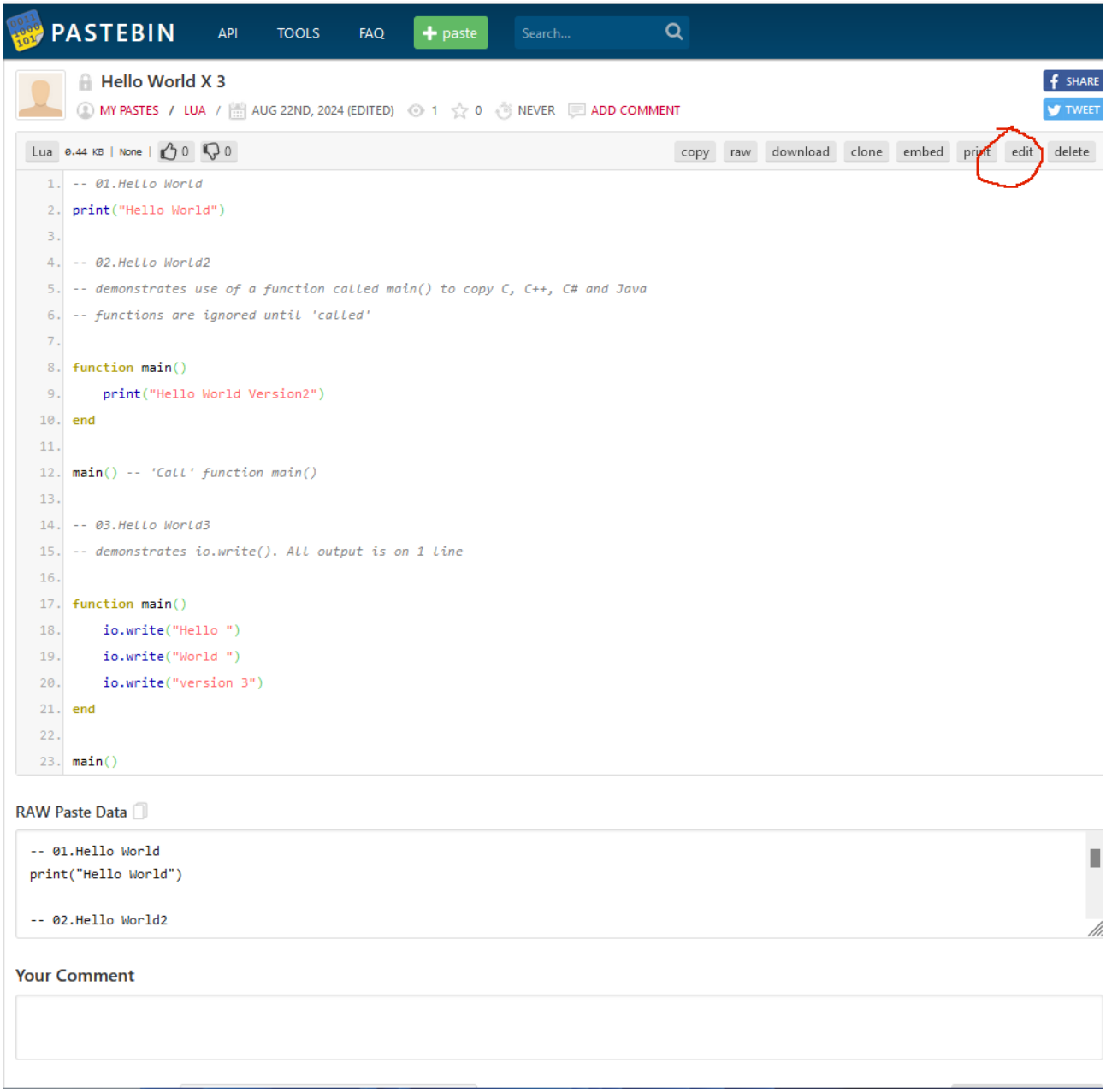
Go to pastebin.com and log in.

Copy / paste the code from one or more of your Repls. Enable Syntax highlighting.
Set Syntax highlighting to Lua. Add a new Folder and call it Lua.
Fill in the paste name / Title



Click Create new paste

Once created, you can edit it at any time. I forgot to give it a title:



Click the “edit” button, change any of the code or the Title / syntax / comment etc. then save.

You can now over-write any of your 3 Repls to continue.

UTF8 characters

Available from https://www.w3schools.com/charsets/ref_utf_box.asp

utf8 box characters stored here for easy copy/paste (not from printed document!)

```
┌ ┬ ┐ ─ ╓ ╥ ╖ ═
├ ┼ ┤ │ ╟ ╫ ╢ ║
└ ┴ ┘   ╙ ╨ ╜
```

These characters can improve your console based projects in Lua, Python, C# and many other languages.
They cannot be easily typed in, so are best copy/pasted into your code.

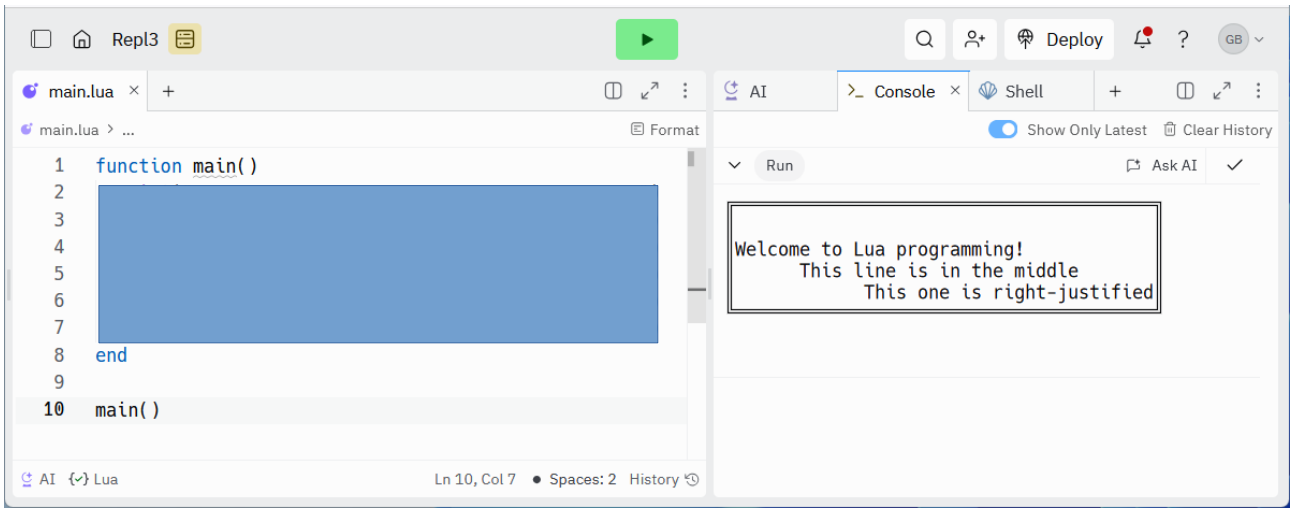
Assignment 1.lua

Get the code below from https://pastebin.com/qxxS26r3 and paste it into a Repl

main.lua

```lua
--[[
utf8 box characters stored here for easy copy/paste:

┌ ┬ ┐ ─ ╓ ╥ ╖ ═

├ ┼ ┤ │ ╟ ╫ ╢ ║

└ ┴ ┘   ╙ ╨ ╜
Use the template below to create an output that looks like this:
Ignore the [ and ]. They just allow multi-line comments!
]]

--[[

╔═══════════════════════════════════════╗
╟                                       ╢
╟Welcome to Lua programming!            ╢
╟      This line is in the middle       ╢
╟            This one is right-justified╢
╚═══════════════════════════════════════╝
]]

-- Use either print() or io.write()
-- use spaces to pad the text
-- (remember io.write() does NOT move to a new line

function main()
    -- your code goes here
end

main()
```

Use the print() or io.write() functions to get this output. The code has been hidden below:



You can use the UTF8 characters for fancy box construction. They can also be found here:

https://www.w3schools.com/charsets/ref_utf_box.asp (/ref_utf_box.asp)

Copy / paste the characters into your code.

Variables

Variables are memory locations reserved to hold some kind of data, and given a label to identify them. They must begin with a letter or an underscore character, and must not contain any spaces.

```
Examples:

myName = "Fred"
myAge = 15
isWorking = false
```

```
These DO NOT WORK!:

1myName (starts with a number)
!myName (starts with a !)
my Name (contains a space)
```

The first one 'myName' is the label, and it has been given the word 'Fred' as the data stored in it.
Variables containing a string of letters or words are called 'string' variables

myAge contains a whole number. It is called an 'integer' variable

isWorking contains either true or false. It is called a 'boolean' variable

Lua and Python have a fairly flexible approach to variables. You can change the data stored in them from string to integer or boolean without any problem. (This is not the case with C# and Java)

Comments

As your script gets bigger, it is helpful to write in some comments to help others understand what you are doing, or to remind yourself if you come back to it after some time.

Single-line comments are started with two hyphens: - - (no space between them)
(Python comments start with a #, C# and Java use //)

Comments are ignored by the interpreter.

Multi-line comments start with --[[and end with]]
(Python uses ‘’’ or “”” at the beginning and end, C# and Java use /* at the start and */ at the end)

```
--[[ This is a multi-Line comment.
    It allows plenty of room to make notes]]

print(“Hello World”) -- this line prints ‘Hello World’
```

User Input

Most programs need some sort of input from the user, either from the keyboard, mouse or other device such as a joystick.

Lua and Python cannot handle anything except the keyboard without using other libraries (pre-written code modules), so the next part of the tutorial is based on keyboard input.

When moving on to the Love2D game engine, then the mouse can be used. (Pygame or Tkinter in Python)

Python users will be aware of the input() function to get keyboard data.

Lua has a built-in library called *io* (input/output)

To read what the user types on the keyboard use `io.read()`

It is usual to assign the keyboard input to a variable:

```
typedInText = io.read()
```

The variable typedInText now contains whatever the user typed at the keyboard until they hit the 'Enter' key.

If they did not type anything and just hit the Enter key, then the variable holds an empty string (length 0)

03-Input1.lua:

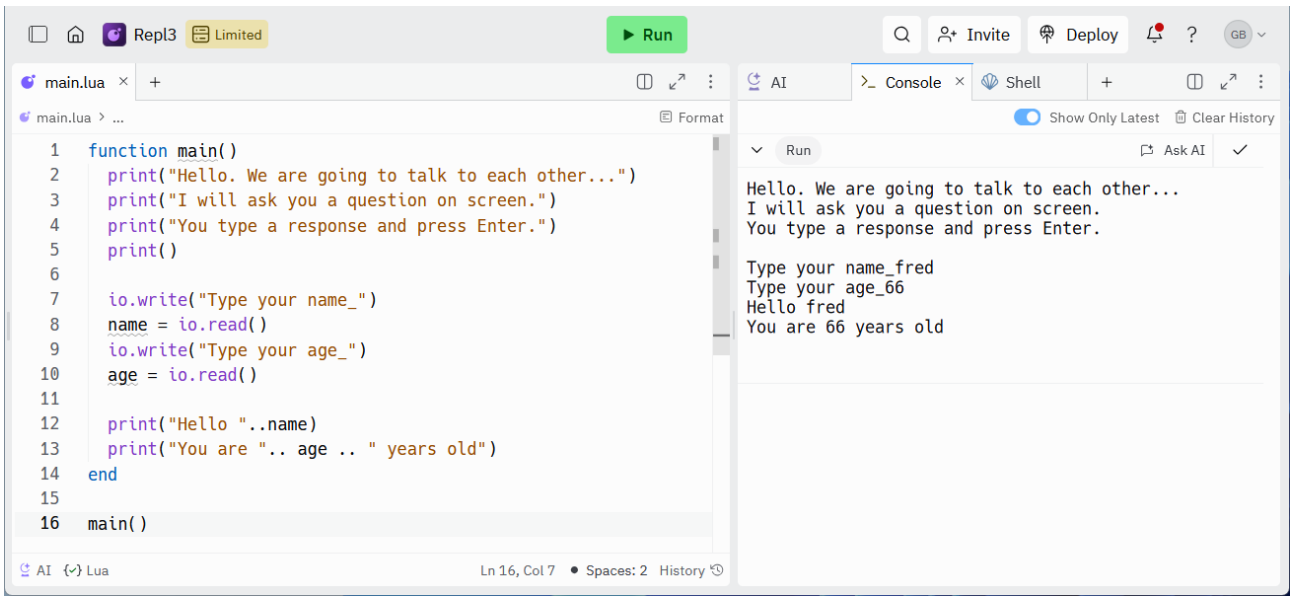
Type this into one of the Repls you no longer need. (Screenshot on next page)

```
function main()
    print("Hello. We are going to talk to each other...") -- Spooky!!!
    print("I will ask you a question on screen.")
    print("You type a response and press Enter.")
    print()                                    -- Print a blank line

    io.write("Type your name_")  -- io.write() does NOT move to the next line
    name = io.read()             -- io.read() stores what you type when you press Enter
    io.write("Type your age_")
    age = io.read()

    --[[You can see we are using the 2 lines
      io.write(Some text here)
      var = io.read() -- var is any variable such as name, age, height etc
    ]]
    print("Hello "..name)
    print("You are ".. age .. " years old")  -- the .. dots join strings together
end

main()
```

Note joining 2 strings uses **. .** Python and other languages use **+**

```
print("Hello "..name)
```

This line of code means: print "Hello " followed by the *contents* of the variable 'name'. If name contained "Fred" then the output would be "Hello Fred".



Note the repeated use of:

```
io.write("Type your name_")
name = io.read()

io.write("Type your age_")
age = io.read()
```

A general programming rule is never to write lines of code repetitively.
The line `io.write("Type your name_")` prints the "prompt" onto the screen first, then waits for the user to enter their response using `age = io.read()`

This needs another function, to replicate Python's `input()` which allows a single line of code to output the 'prompt' and receive the user input. Once written it can be re-used as often as you want.

```
function input(prompt)
     io.write(prompt .. "_")
     return io.read()
end
```

Use it to get the user to enter their age:

```
name = input("Type your name")
```

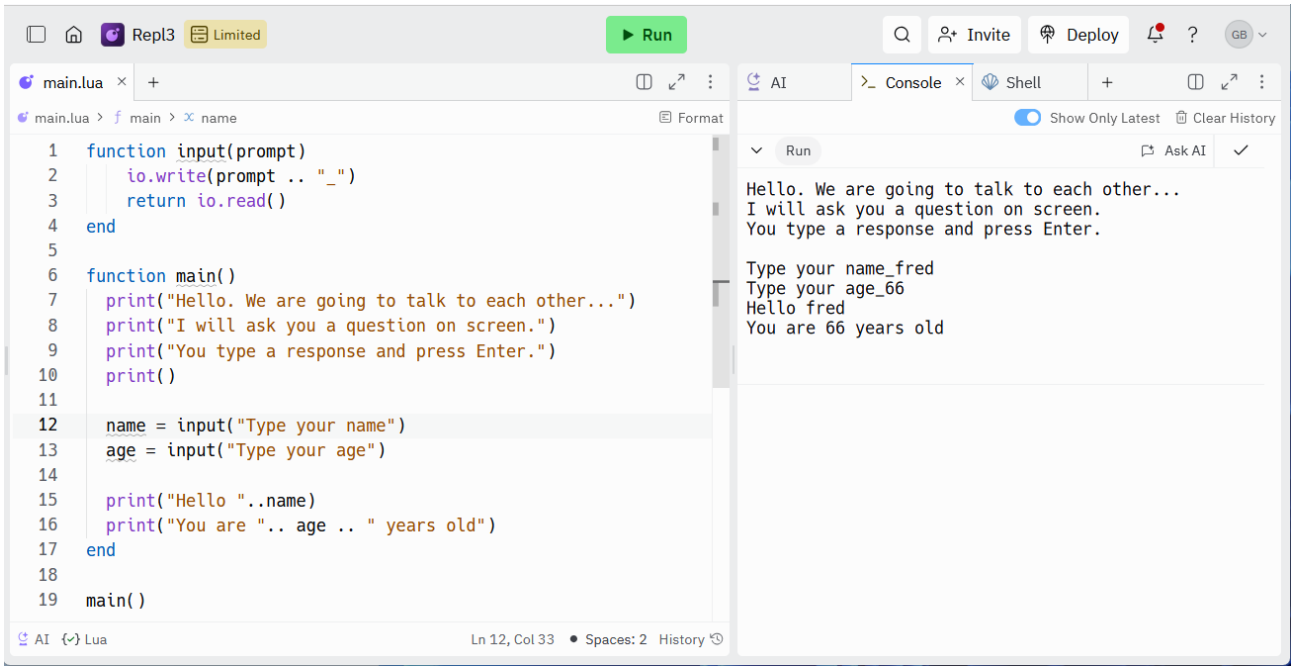
The words "Type your name" are passed to the function and stored in the variable 'prompt'. The function prints them out, then returns the user's response back.

04-Input2.lua:

```lua
 -- does exacly the same as 03-Input1.lua
 -- demonstrates new function called input() with return value

function input(prompt)
    io.write(prompt .. "_") -- "_" is added to prompt eg Type your name_
    return io.read()      -- io.read() sends what you typed in to where it was called
end

function main()
    print("Hello. We are going to talk to each other...")
    print("I will ask you a question on screen.")
    print("You type a response and press Enter.")
    print()

    --[[ These 2 lines are no longer required:
      io.write("Type your name_")
      name = io.read()
      Use the brand new input() function instead
    ]]

    name = input("Type your name") -- does the same job as above, but can be re-used
    age = input("Type your age")

    print("Hello " .. name)
    print("You are ".. age .. " years old")
end

main()
```

The output is exactly the same, but the new function `input()` is used twice.

```lua
function input(prompt)
     io.write(prompt .. "_") -- "_" is added to prompt eg Type your name_
     return io.read()     -- io.read() sends what you typed in to where it was called
end
```



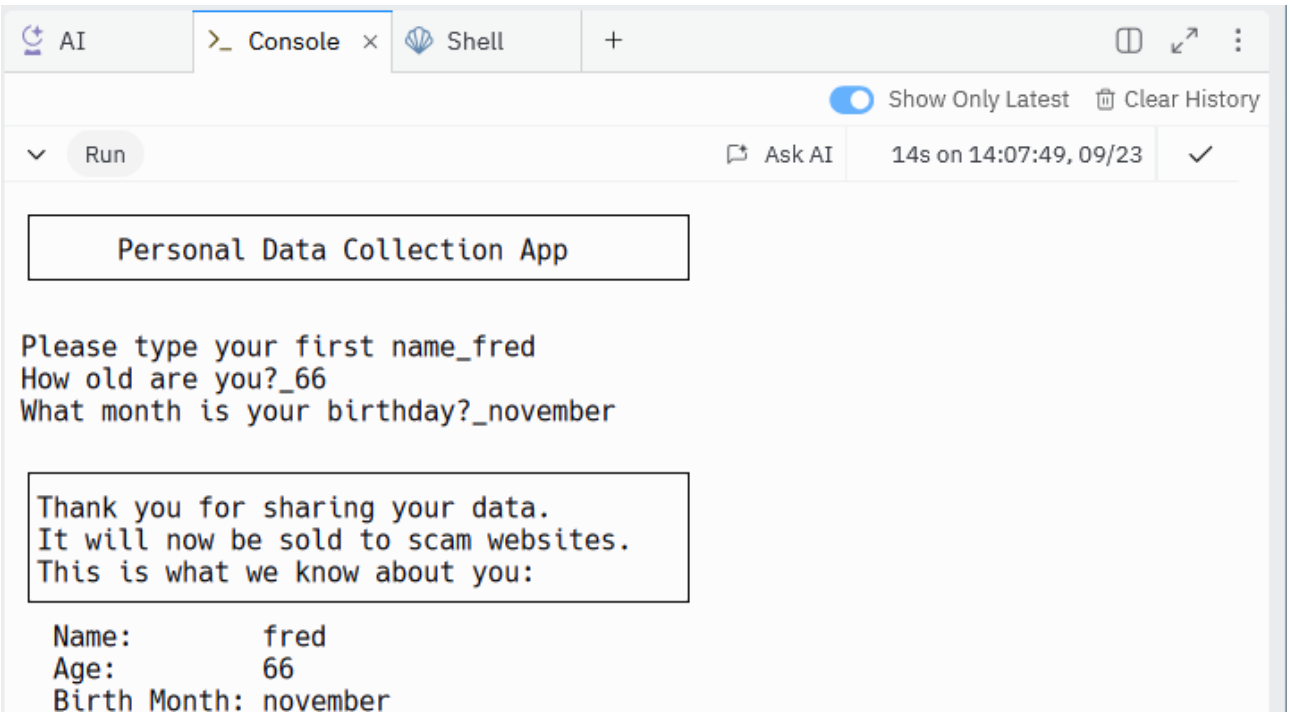
The only problem here is there is no validation of what the user is typing.
You might want a number, but that is not checked, or you might want a string of a particular length.

There is another assignment first, then an input library can be written.

05-Assignment 2.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section1/09-Assignment%202.lua

Use the template code to produce the output below. You can use UTF8 characters or repeated hyphen (minus) ------- , plus +++++ , pipe |||| (shift and \)



First help to get you started:

```lua
-- create 3 variables called name, age, month to store the data
-- use print(), io.write(), input()
-- complete the output to include this data


function input(prompt)
    io.write(prompt .. "_") -- io.write() does NOT move to the next line
    return io.read()      -- io.read() sends what you typed in to where it was called
end

function main()
    -- your code starts here
end

main() -- program starts here
```

Second help: Here is (literally) half the code:

```
6   function main()
7     -- your code starts here
8     local top = "┌──────────────
9     local bottom = "└────────────
10
11    print(top)
12    print("|     Personal Data (
13    print(bottom)
14    print()
15
16    local name = input("Please t
17    local age = input("How old a
18    local month = input("What mo
19
20    print()
21    print(top)
22
23    print("|Thank you for sharir
24    print("|It will now be sold
25    print("|This is what we know
26
27    print(bottom)
28    print("  Name:        "..nam
29    print("  Age:         "..age
30    print("  Birth Month: "..mor
31  end
```

Why are the words ‘local’ appearing here?

This is part of a topic called variable ‘scope’

The variables marked ‘local’ only exist inside the main() function.
It is good practice to always use ‘local’ as part of the variable declaration.

It runs faster and can avoid mistakes in long scripts. This will be covered fully in the future.

Section 2

Validating input

The next 3 files explain how to handle numbers and strings when input from a user, but there is still no checking. Errors will occur if the user types characters that cannot be converted to a number.

01-Variables-integer1.lua

```
-- demonstrates tostring()
function input(prompt)
    io.write(prompt .. "_")
    return io.read()
end

function main()
    -- create a number variable and give it a value of 10
    myNumber = 10
    -- Try and Print out the value of the variable
    print("the variable myNumber contains: "..myNumber)

    -- This works perfectly as Lua uses .. to join strings
    -- and converts the number 10 to the string "10" on the fly
    -- With Python, this will not work, and you get this message:
    -- TypeError: Can't convert 'int' object to str implicitly
    -- use tostring() to be certain
    print("myNumber = "..tostring(myNumber))
end

main()
```

Because Lua has a string concatenation symbol instead of using + there is no problem joining a string with a number. When compiled, the number is converted to a string automatically.

There is a converter that can be used if required, eg checking for nil values:

`print(tostring(nil))` → “nil”

`print(tostring(true))` → “true”

`print(tostring(5 + 5))` → “10”

02-Variables-integer2.lua

This file demonstrates the use of `tonumber(value)` as well as `tostring(value)`.
If you type in 'ten' when asked for a number, the result of `tonumber("ten")` is nil
Using `tostring(userInput)` when outputting the final print() ensures there is no error trying to concatenate a string with nil

main.lua

```lua
-- tonumber(), tostring()
function input(text)
    io.write(text.."_")
    return io.read()
end

function main()
    -- Anything typed in is a string, INCLUDING numbers!
    -- Ask the user to type something, and store what they typed.
    userInput = input("Type in any letters, numbers or symbols and press Enter")
    print("You typed in the characters "..userInput)
    -- Ask the user to enter only numbers
    userInput = input("Type in any number, and press Enter")

    -- If they typed in 3, it is the character "3"
    -- You have to convert it to a number
    userInput = tonumber(userInput) -- if it cannot be converted it becomes nil
    print("The variable userInput now contains:  "..tostring(userInput))
end

main()
```

First test: type fred66 : ok
Type 1234: ok

```
Type in any letters, numbers or symbols and press Enter_fred66
You typed in the characters fred66
Type in any number, and press Enter_1234
The variable userInput now contains:  1234
```

Second test: type fred66 : ok

Type fr3d (Not a number) result = nil

```
Type in any letters, numbers or symbols and press Enter_fred66
You typed in the characters fred66
Type in any number, and press Enter_fr3d
The variable userInput now contains:  nil
```

03. Variables-float.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/03-Variables-float.lua

Similar to above, but using real numbers. Lua does not have specific integer maths like other languages.

```lua
-- demonstrate float and use of tostring() and tonumber()
function input(prompt)
    io.write(prompt .. "_")
    return io.read()
end

function main()
    -- Float variables
    myFloat  = 1.5 -- create a float variable
    -- Note the use of the tostring() function
    print("The variable myFloat contains: ".. tostring(myFloat))

    -- get a number from the user
    userInput = input("Type a number from 1 to 100 (program will crash if not a number)")
    -- convert userInput to number
    userInput = tonumber(userInput) -- if it cannot be converted it becomes nil
    -- program will crash if you multiply a number with nil!!!
    -- error message: attempt to perform arithmetic on global 'userInput' (a nil value)
    print("Your number: ".. userInput .." multiplied by " .. myFloat .. " = " .. userInput * myFloat)
end

main()
```

Conditional Statements

These are an essential part of any programming languages. The examples in Blue textboxes are NOT on Github!

The first keyword is **'if'**

You will be asked to use something called **pseudocode** during your studies. This is a code-like approach, but not in the syntax of a specific language. Here is an example:

```
if some condition is true then
    do some code
end
```

This code block will only 'do some code' if some condition checked in the line containing 'if' is true.

A more realistic pseudocode block is:

```
if userInput == "your name" then
    print("That is correct!")
end
```

The double == means 'Is the variable `userInput` equal to'

(The statement: userInput = would assign a value to the variable.)

Lua is the closest programming language to pseudocode. The block above is actually correct Lua syntax!

The second keyword is `'else'`.

This allows an alternative block of code to run if the original condition is not true.

You can check for an alternative condition by using `'elseif'` (Python uses `'elif'`, C# uses `'else if'`)

```
if userInput == "your name" then
    print("That is correct!")
elseif userInput == "name" then
    print("That is half correct")
end
```

04-If.lua

This file uses an if statement to check if the user entered a value that cannot be converted to a number.

If they did enter a number then the calculation is performed and output.

```
-- demonstrate if statement to prevent crash
function input(prompt)
    io.write(prompt .. "_")
    return io.read()
end

function main()
    -- Float variables
    myFloat  = 1.5 -- create a float variable
    -- Note the use of the tostring() function
    print("The variable myFloat contains: ".. tostring(myFloat))

    -- get a number from the user
    userInput = input("Type a number from 1 to 100")
    -- convert userInput to number
    userInput = tonumber(userInput) -- if it cannot be converted it becomes nil

    -- use of if statement checks if user did not type a number
    if userInput == nil then -- did not convert so now nil. note: ==
        print("You did not type a number")
    else
        print("Your number: ".. userInput .." multiplied by " .. myFloat .. " = "
.. userInput * myFloat)
    end
end

main()
```

05-IfElseifElse.lua

```lua
function input(prompt)
    io.write(prompt .. "_")
    return io.read()
end

function main()
    -- Boolean variables can only be either true or false
    -- Most languages associate true = 1, false = 0
    -- You can also think of yes = true, no = false

    choice = false -- variable called 'choice' is given the default value false
    userInput = input("Do you like Lua? (y/n)")
    -- user SHOULD have typed a 'y' or 'n'
    if userInput == "" then                          -- Enter only
        print("You only pressed the Enter key")
    elseif userInput == 'y' then                     -- 'y' typed in
        print("Great! variable 'choice' is now true")
        choice = true                                -- set choice to true as the user typed 'y'
    elseif userInput == 'n' then                     -- 'n' typed in
        print("Oh. That is disappointing")
    else                                             -- some other characters typed in
        print("You typed "..userInput.." I can't translate that to true/false")
    end

    print("\nThe value of the boolean variable 'choice' is: " .. tostring(choice))
end

main()
```

String Operations

The next two files deal with the string library, and the use of Lua's "syntactic sugar" where a colon can be used instead of a dot:

```lua
string.upper(value))
```

can be re-written as

```lua
value:upper()
```

06-Strings.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/06-Strings.lua

Repl1 Limited Run

main.lua

main.lua > ...

```lua
-- demonstrates  #, string.upper(), string.lower()
-- Lua 'syntactic sugar' : instead of .
function input(prompt)
    io.write(prompt .. "_")
    return io.read()
end

function main()
    userInput = input("Type your name")
    numberOfChars = #(userInput) -- # = length of UserInput
    print("Your name ".. userInput .. " has " .. numberOfChars .. " characters in it")

    -- string.upper() converts all characters to UPPER CASE
    print("Your name ".. userInput .. " in upper case is ".. string.upper(userInput))
    print("Your name ".. userInput .. " in upper case is ".. (userInput):upper())

    -- string.lower() converts all characters to lower case
    print("Your name ".. userInput .. " in lower case is ".. string.lower(userInput))
    print("Your name ".. userInput .. " in lower case is ".. (userInput):lower())
end

main()
```

AI Console Shell

Show Only Latest Clear History

Run Ask AI

```
Type your name_fReD
Your name fReD has 4 characters in it
Your name fReD in upper case is FRED
Your name fReD in upper case is FRED
Your name fReD in lower case is fred
Your name fReD in lower case is fred
```

07-Strings2.lua

The start of a very silly game, where the user has to type “your name” instead of their real name

```lua
function input(prompt)
    io.write(prompt .. "_")
    return io.read()
end

function main()
    userInput = input('Type "your name"')

    if userInput == "your name" then -- user typed in 'your name' as instructed!
        print("That is correct!")
    else -- user typed in their real name
         -- shout at the user! (UPPER CASE)
        print(("Unfortunately, that is wrong!"):upper()..." Try again...\n")
    end
end

main()
```

The downside is you only get one try. If you mess up you have to run it again.

```
Type "your name"_fred
UNFORTUNATELY, THAT IS WRONG! Try again...
```

To fix that, there needs to be a way of looping round and try again until the correct input is entered.

The next 4 files deal with loops.

Loops

08-Loop1.lua Infinite while loop

main.lua

main.lua > main > while > ...

```lua
-- demonstrates use of the while..do..end loop
--[[
 In the previous script, it only runs once. If you type in the
 words "your name" you get a brownie point. Otherwise you are wrong.
 But you can only do it once, then you have to re-start.
 Use a loop to allow another chance:
]]--
function input(prompt)
    io.write(prompt .. "_")
    return io.read()
end

function main()
    while true do -- true is always true, so this is an infinite loop
        userInput = input('Type "your name"')
        if userInput == "your name" then    -- user typed in 'your name' as instructed!
            print("That is correct!")
            break                           -- break out of the loop
        else                        -- user typed in their real name
            -- shout at the user! (UPPER CASE)
            print(string.upper("Unfortunately, that is wrong! Try again...\n"))
        end
    end
end

main()
```

The line `while true do` will repeat all the following lines up to it's closing 'end' statement continuously, or until a break statement is encountered.

This is called an infinite loop because the while condition cannot change. True is always true.

09-Loop2.lua Improved while loop

A more specific while loop could be used, where the condition being checked is what userInput contains:

main.lua

main.lua > main > while > ...

```lua
-- demonstrates while loop without break
function input(prompt)
    io.write(prompt .. "_")
    return io.read()
end

function main()
    userInput = ""
    while userInput ~= "your name" do      -- ~= means 'is NOT equal to'
        userInput = input('Type "your name"')
        if userInput == "your name" then   -- user typed in 'your name' as instructed!
            print("That is correct!")
            --break no longer needed. the loop will not run again as userInput is now = 'your name'
        else    -- user typed in their real name
            print(string.upper("Unfortunately, that is wrong! Try again...\n"))
        end
    end
end

main()
```

The symbol `~=` means 'not equal'

The line

```lua
while userInput ~= "your name" do
```

translates to:

while `userInput` is **not** equal to '**your name**' do

As it was set to an empty string when the loop started, this condition is true, so the loop runs at least once.

If the user types in 'your name', the message `"That is correct!"` is printed out, but the loop exits because it's condition is no longer true. UserInput IS equal to 'your name'.

A variation on the while loop is a repeat until loop:

10.Loop3.lua – repeat until

main.lua

```lua
-- demonstrates repeat..until loop

function input(prompt)
    io.write(prompt .. "_")
    return io.read()
end

function main()
    -- userInput == "" <- no longer needed
    repeat -- starting a loop with repeat forces it to run at least once
        userInput = input('Type "your name"')
        if userInput == "your name" then    -- user typed in 'your name' as instructed!
            print("That is correct!")
        else                    -- user typed in their real name
            print(string.upper("Unfortunately, that is wrong! Try again...\n"))
        end
    until userInput == "your name"    -- as soon as userInput == 'your name' the loop will end
end

main()
```

This loop always runs at least once, which is the main reason for using it.

Another loop called a ’for’ loop will be covered next.

Section 3

01-ForLoops.lua Drawing triangles with ASCII

main.lua

```lua
-- demonstration of for loops and string.rep()
function input(prompt)
    io.write(prompt .. "_")
    return io.read()
end

function main()
    numberOfRows = input("Type a number between 5 and 20")

    numberOfRows = tonumber(numberOfRows)
    if numberOfRows ~= nil then
        -- draw a triangle
        -- for variable = start, finish, step do
        for i = 1, numberOfRows, 1 do -- eg start at 1 step to 6: 1, 2, 3, 4, 5, 6
            -- i starts at 1, then steps 2, 3, 4, 5, etc -> numberOfRows
            -- string.rep() repeats the character(s) given by the number supplied
            -- eg string.rep("*", 4) returns "****"
            lineOfChars = string.rep("*", i)
            print(lineOfChars)
        end
        -- reverse the triangle by starting with a high number and using -1 for the step: 6, 5, 4, 3, 2, 1
        for i = numberOfRows, 1, -1 do
            lineOfChars = ("*"):rep(i)
            print(lineOfChars)
        end
    end
end

main() -- program starts here
```

Run Ask AI 17s on 14:50:43, 09/23

```
Type a number between 5 and 20_5
*
**
***
****
*****
*****
****
***
**
*
```

For loops differ from while loops because the number of times they iterate is limited and fixed.

A typical for loop such as:

```lua
for i = 1, 5, 1 do
```

runs 5 times only.

The variable **`i`** is the loop counter, and its value changes by 1 every time the loop runs.

When the value of `i` reaches 5 the loop runs one last time then stops. The value of `i` can be used by the code within the loop itself, as in this code where it is used to create a string of "*" characters of the length defined by the value of `i`.

```
lineOfChars = string.rep("*", i)
```

or using lua syntactic sugar: `lineOfChars = ("*"):rep(i)`

Note the second for loop uses -1

```
for i = numberOfRows, 1, -1 do
```

this runs the loop in reverse. The counter starts at the highest value, and drops by -1 each iteration until it reaches 1.
The "step" is -1 but can be set to any integer value.
General 'for loop' construction:

```
for counter = startValue, endValue, step do
```

Random Numbers

Games often use random numbers, and Lua has a method of generating them, using it's maths library

If you want a random number between 1 and 99 use this:

```
local randomNumber = math.random(1, 99)
print(randomNumber)
```

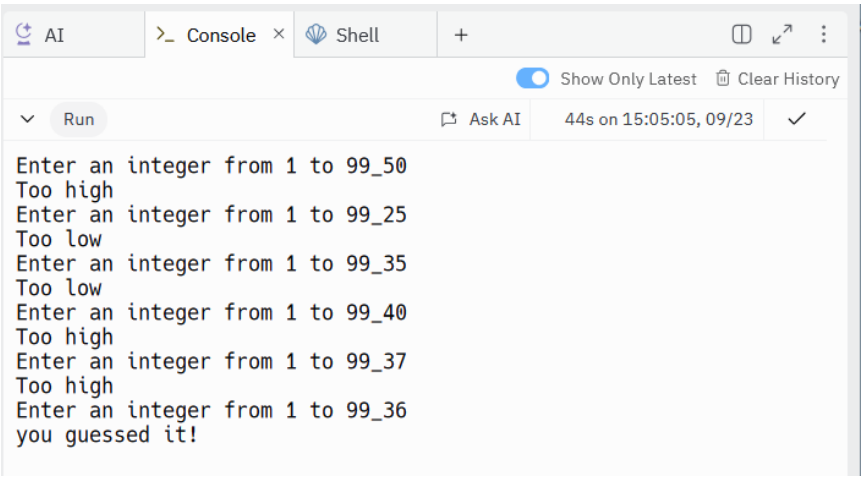
The numbers are not truly random and the same sequence is generated each time the program runs. This can be overcome by setting a seed for the generator, based on the current time:

```
math.randomseed(os.time())
local randomNumber = math.random(1, 99)
```

This simple game uses the random number function to create a number between 1 and 99
The user is asked to guess the number using a modified version of the input() function developed earlier inside a repeat-until loop.

main.lua

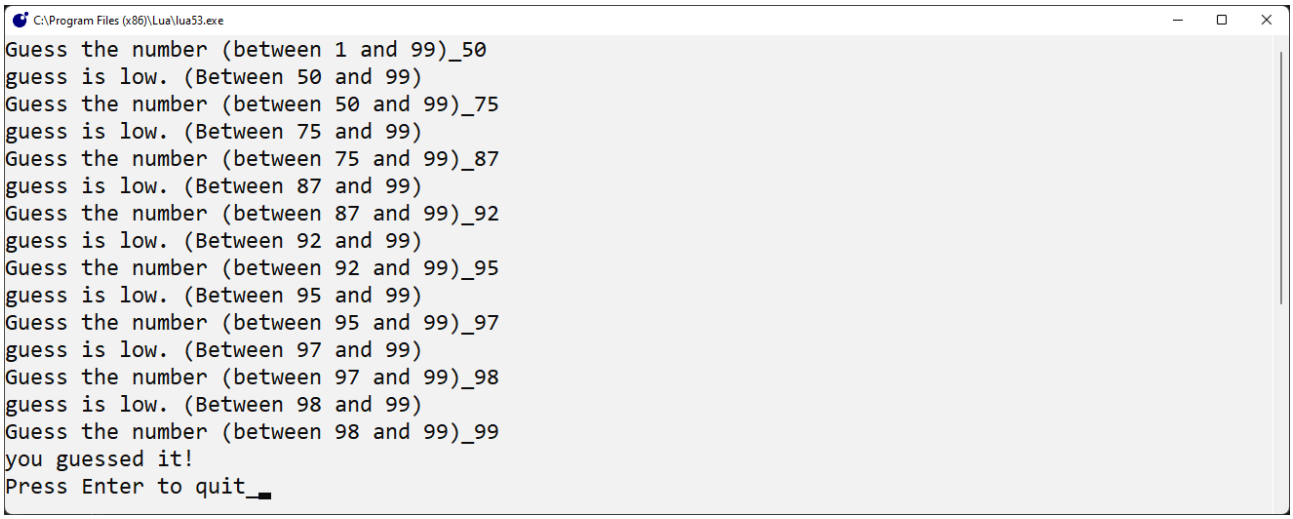
```lua
local function getNumber(prompt) -- get input from user
    while true do           -- break not required as return used instead
        io.write(prompt .. "_")
        local userInput = io.read()

        if tonumber(userInput) ~= nil then
            return tonumber(userInput)
        else
            print("Enter a number ".. userInput .. " does not work")
        end
    end
end

local function main()
    math.randomseed(os.time())      -- set the random seed
    local n = math.random(1, 99)    -- pick a number between 1 and 99

    repeat
        -- no need to convert guess to a number, as return type is guaranteed
        local guess = getNumber("Enter an integer from 1 to 99")
        if guess < n then
            print("Too low")
        elseif guess > n then
            print("Too high")
        else
            print("you guessed it!")
        end
    until guess == n
end

main()
```



05-Assignment-Improve GuessTheNumber.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section3/05-GuessTheNumber%20Assignment.lua

The version above only tells you if you are too low or too high. It would be great if you were given some help remembering what you have already tried:



Modify the code above to give the helpful guidance shown in the screenshot → the range of numbers remaining.

Hints:

Create 2 variables to hold the largest and smallest numbers guessed so far.

Re-assign these variables as guesses are made

Output an appropriate message after each guess, telling the user how they have fared, and the range they now need to use.

Section 5

Lua Tables

So far you have used simple variables to store a single string, number or boolean value.

But what if you wanted to store a list of strings?

Or need some kind of super-variable that could hold many different types of data?

A simple example would be to write a program to ask the user for their name, and compare it to a list of names already stored in memory. If they are on the list, they are welcome to continue, otherwise the program quits.

Python, C# and Java use either/or Arrays, Lists and Dictionaries to hold this data:

Python List: `myFriends = [“Fred”, “Alice”, “Jim”, “Karen”]`

C#, Java: `List<string> myFriends = new List<string> {“Fred”, “Alice”, “Jim”, “Karen”};`

Lua: `local myFriends = {“Fred”, “Alice”, “Jim”, “Karen”}`

Use of the curly braces `{}` is all that is needed in Lua, but it is not an array, or a list or a dictionary. It is a **table.**

You can use a numerical index to read/write to specific parts of the list/table:

Python, C# Java: `myFriends[0] = “Fred”`

Lua: `myFriends[1] = “Fred”`

The only difference is the start of the index:

Lua’s index starts at 1, all other languages start at 0

Lua tables can also be used like this:

```
local myFriends = {} -- empty table
myFriends.Best = “Fred” (or myFriends[“Best”])
myFriends[“SecondBest”] = “Alice” (or myFriends.SecondBest)
```

The closest equivalent to this in other languages is the Dictionary:

Python dictionary: `myFriends = {“Best”:”Fred”, “SecondBest”:”Alice”}`

Python has a very useful ‘in’ keyword to check if an item is in a list:

Python:

```
myFriends = [“Fred”, “Alice”, “Jim”, “Karen”]

myName = input(“Type your name”)
if myName in myFriends:
     print(“Hello Friend”)
```

Although Lua has an ‘in’ keyword, it does not work in the same way, so you have to loop through the whole table to check if the item is in there.

01-Tables1.lua

main.lua

```lua
-- introduction to tables

function input(prompt) -- get input from user
    io.write(prompt .. "_")
    return io.read()
end

function main()
    -- Python has a list dataType. Lua has a table instead

    colours = {"red", "green", "blue"} --create a table of colours

    print("print(colours): \t\t\t"..tostring(colours))     -- table: 0x00698170 Yuk!
    print("print(colours[1]): \t\t\t"..colours[1])        -- 'red'
    print("print(colours[2]): \t\t\t"..colours[2])        -- 'green'
    print("print(colours[3]): \t\t\t"..colours[3])        -- 'blue'
    print("print(table.concat(colours, ',')): \t"..table.concat(colours, ",")) -- 'red,green,blue'
end

main() -- program starts here
```

AI | Console | Shell

Show Only Latest Clear History

Run Ask AI

```
print(colours):                 table: 0xfcd6d0
print(colours[1]):              red
print(colours[2]):              green
print(colours[3]):              blue
print(table.concat(colours, ',')):  red,green,blue
```

This code created a table of 3 colours and printed out the contents using `table.concat(colours, ",")`

This is a simple way of getting all the contents into a single string for display purposes.

02-TableInsert.lua – Adding to a table

main.lua

```lua
-- adding to a table
function input(prompt) -- get input from user
    io.write(prompt .. "_")
    return io.read()
end

function main()
    colours = {"red", "green", "blue"} --create a table of colours
    myFavourites = {} -- create an empty table

    print("Colours table: "..table.concat(colours, ", "))
    print("Creating a table of 3 of your favourite colours...")
    for i = 1, 3 do
        userInput = input("Type your favourite colour (number"..i..")")
        table.insert(myFavourites, userInput)
    end
    print("colours table: "..table.concat(colours, ", "))
    print("Favourite colours table: "..table.concat(myFavourites, ", "))
end

main() -- program starts here
```

AI | Console | Shell

Show Only Latest | Clear History

Run | Ask AI

```
Colours table: red, green, blue
Creating a table of 3 of your favourite colours...
Type your favourite colour (number1)_purple
Type your favourite colour (number2)_orange
Type your favourite colour (number3)_black
colours table: red, green, blue
Favourite colours table: purple, orange, black
```

A new empty table is created:

```
myFavourites = {}
```

The user is asked for a favourite colour

```
userInput = input("Type your favourite colour (number"..i..")")
```

This uses `table.insert(tableName, value)` to insert the new value into the table:

```
table.insert(myFavourites, userInput)
```

Note: there is no check to see if the same colour is inserted more than once.

Both tables are printed out using `table.concat(table, separator)`

03-TableRemove.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section5/03-TableRemove.lua

main.lua

```lua
-- removing from a table
function input(prompt) -- get input from user
    io.write(prompt .. "_")
    return io.read()
end

function main()
    colours = {"red", "green", "blue"} --create a table of colours

    print("Colours table: "..table.concat(colours, ", "))
    while #colours > 0 do
        userInput = input("Which colour do you want to remove?")
        local found = false
        for index = 1, #colours do
            if colours[index] == userInput then
                table.remove(colours, index)
                found = true
                break
            end
        end
        if found then
            print("colours table: "..table.concat(colours, ", "))
        else
            print(userInput.." not found in the table")
        end
    end
end

main() -- program starts here
```

```
Colours table: red, green, blue
Which colour do you want to remove?_green
colours table: red, blue
Which colour do you want to remove?_red
colours table: blue
Which colour do you want to remove?_blue
colours table:
```

During the while loop, the user is asked which colour they want to remove.
If they type in a colour that is not in the table, a message tells them.
If they type in a matching colour, it is removed.
The while loop ends when the table is empty: `#colours > 0 = false`

Lua Tables and for loops

04-TableIteration.lua

```
Please type your first name_fred
Checking key: 1 value: Fred: found = true
Checking key: 2 value: Alice: found = false
Checking key: 3 value: Jim: found = false
Checking key: 4 value: Karen: found = false
Hello friend
```

```lua
        else
            print(": found = false")
        end
    end
    if found then
        print("Hello friend")
    else
        print("I do not know you")
    end
end

main()
```

The for loop used here is specifically used to iterate list-type tables.

```
for key, value in ipairs(myFriends) do
```

It can only be used when the table has a numerical index as used in Python / C# lists

It will not work on dictionary-type tables.

A variation is used for these:

`for key, value in pairs(table) do` (note pairs, not ipairs)

The code above iterates the table and prints out the index (key) and value of each entry:

```
io.write("Checking key: "..key.." value: "..value)
```

The input is checked using the lower case version of the table value and user input:

```
if userName:lower() == myFriends[key]:lower() then
    found = true
    print(": found = true")
else
    print(": found = false")
end
```

06-TableRectangleReplit.lua

This file introduces a table similar to one that can be used in the Love2D environment.

It draws a rectangle that moves across the page in Replit.

The functions update() and draw() mimics Love2D update / draw, which are called 60x per second.

In this file they are called just 20 times, and the resulting rectangle printed out

here is the code. Follow it through to see how it works

You can alter the change in width or x coordinate for fun, or even the number of "frames" it draws.

Do not spend too much time on it, as this will be an early assignment in Love2D

main.lua

main.lua > draw > ...

```lua
local rect = {} -- create an empty table populated in main()

local function clear()
    os.execute("sleep 0.25")     -- wait 1/4 second
    os.execute("clear")          -- ask the operating system to clear the console
end

local function update()
    -- increase the X position
    rect.X = rect.X + 2          -- add 2 to X position
end

local function draw()
    local x = (" "):rep(rect.X)  -- string of spaces equivalent to position of X "" -> "     "
    print(x..rect.top)           -- print top of rectangle
    print(x..rect.middle)        -- print middle of rectangle
    print(x..rect.bottom)        -- print bottom of rectangle
end

local function main()
    rect.X = 0                  -- set X value to 0
    rect.top =    "┌──────┐"
    rect.middle = "│      │"
    rect.bottom = "└──────┘"
    for frames = 1, 20 do
        update()                 -- update values of rect.X
        clear()                  -- clear screen and pause for 1 second
        draw()                   -- draw updated rectangle
    end
end

main()
```