Lua and Love2D Tutorial – Part 1

Note code has been copy/pasted from Visual Studio 2022 to obtain coloured text and the colour highlighting does NOT match ZeroBrane Studio.

You will be following this tutorial as one of the following:

1. I have never done any text-based coding.
2. I have used Python.
3. I have used C#, Java or other languages.
4. I am an expert in everything: Show me something new.

This tutorial has been designed to cover all the above, so references to Python/Pygame, C#/Unity and other languages/game engines used for comparison will be made throughout.

Lua code can be written using Notepad or any text editor, but in school environments running the Lua scripts can be difficult, as you do not have access to the command line, and the chances of an association being set for .lua files to an interpreter are slim.

It is better to use an IDE (Integrated Development Environment).
One of the best is a free application called ZeroBrane Studio which can be downloaded from https://studio.zerobrane.com/

The first thing to get used to when coding in any language is to **stop double-clicking** on your files to launch the editor. This may work for Word, Excel or simple text files, but will not work for coding. At best this will run the file, at worst you will get Windows asking you how it should handle the file type.

The second thing to get used to is organising your coding files properly. This makes finding them easier, and most IDEs in most languages allow you to select a folder and edit or run any of the files within it.

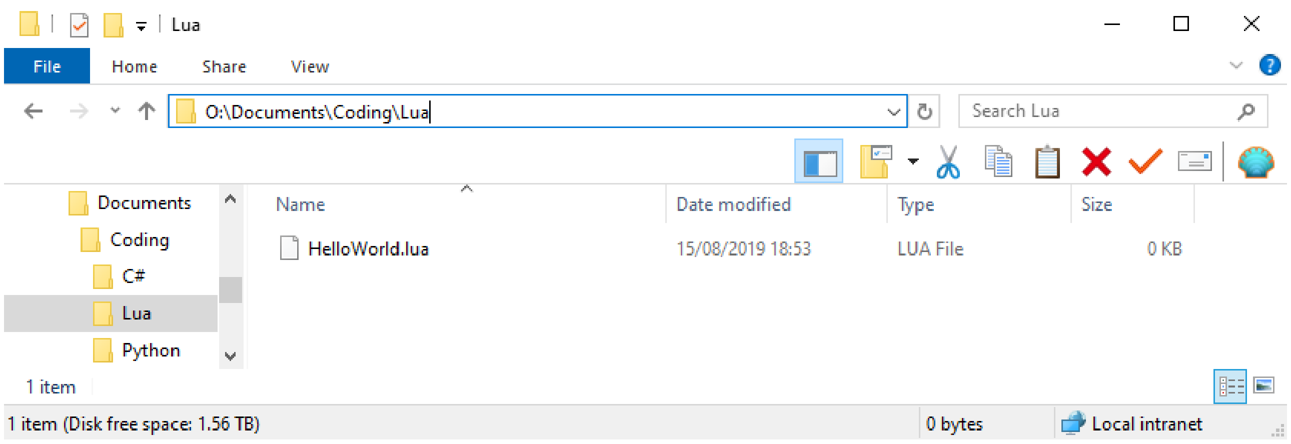
As you progress with coding skills, you will find you are using 'projects' or 'solutions' which consist of multiple files and folders, so it is a good idea to start with that principle.

Organising your files and folders

Suggested scheme to get started:

1. Create a new folder in Documents called 'Coding'
2. Create a new folder inside Coding called 'Lua'
3. Optional (for use later):
4. Create a new folder inside Lua called 'Love2D'

These folders will be empty initially, until you write new files from the IDE. The "HelloWorld.lua" file seen below is an example of this.



Name	Date modified	Type	Size
HelloWorld.lua	15/08/2019 18:53	LUA File	0 KB

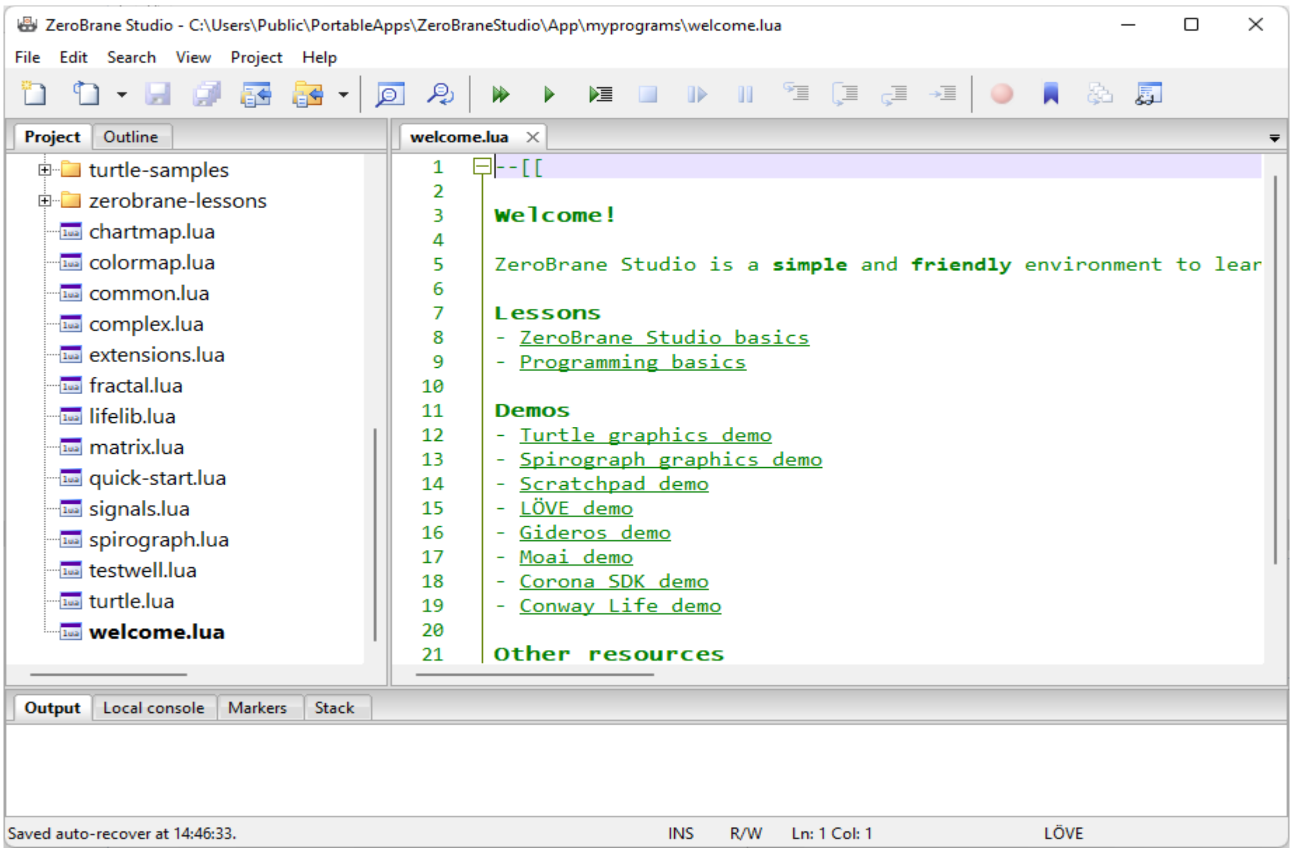
This tutorial has a number of files associated with it, which can be found at:

https://github.com/Inksaver/LuaForSchools

These files have links when they are referred to in this tutorial.

Using ZeroBrane Studio IDE

Start Zerobrane:



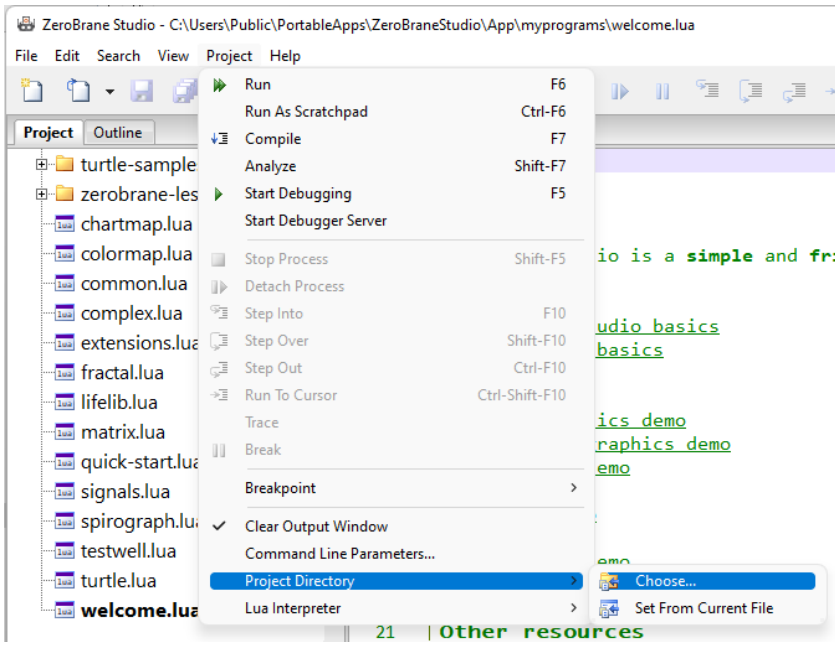
ZeroBrane Studio - C:\Users\Public\PortableApps\ZeroBraneStudio\App\myprograms\welcome.lua

```
--[[

Welcome!

ZeroBrane Studio is a simple and friendly environment to lear

Lessons
- ZeroBrane Studio basics
- Programming basics

Demos
- Turtle graphics demo
- Spirograph graphics demo
- Scratchpad demo
- LÖVE demo
- Gideros demo
- Moai demo
- Corona SDK demo
- Conway Life demo

Other resources
```

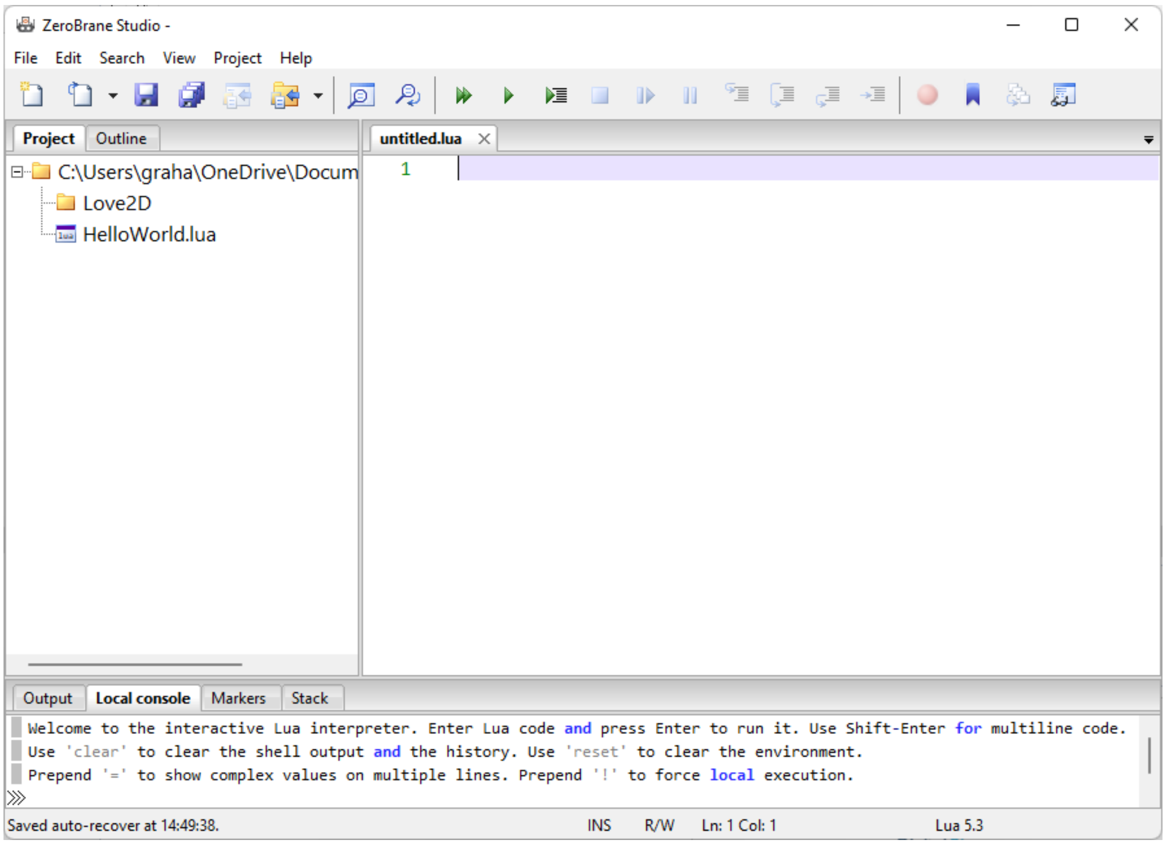
Saved auto-recover at 14:46:33. INS R/W Ln: 1 Col: 1 LÖVE

Normally, the last project worked on will be automatically loaded, but at schools this is unlikely, as the settings are found in C:\Users\<USERNAME>\AppData\Roaming\ZeroBraneStudio.ini.
You would have to use the same workstation every time to ensure your settings are re-used.

Click on the Project Menu → Project Directory → Choose



Locate your newly created Lua Folder and click ‘Select folder’:


ZeroBrane Studio -
File Edit Search View Project Help
Project Outline
C:\Users\graha\OneDrive\Docum
Love2D
HelloWorld.lua
untitled.lua
Output Local console Markers Stack

```
Welcome to the interactive Lua interpreter. Enter Lua code and press Enter to run it. Use Shift-Enter for multiline code.
Use 'clear' to clear the shell output and the history. Use 'reset' to clear the environment.
Prepend '=' to show complex values on multiple lines. Prepend '!' to force local execution.
>>
```

Saved auto-recover at 14:49:38. INS R/W Ln: 1 Col: 1 Lua 5.3

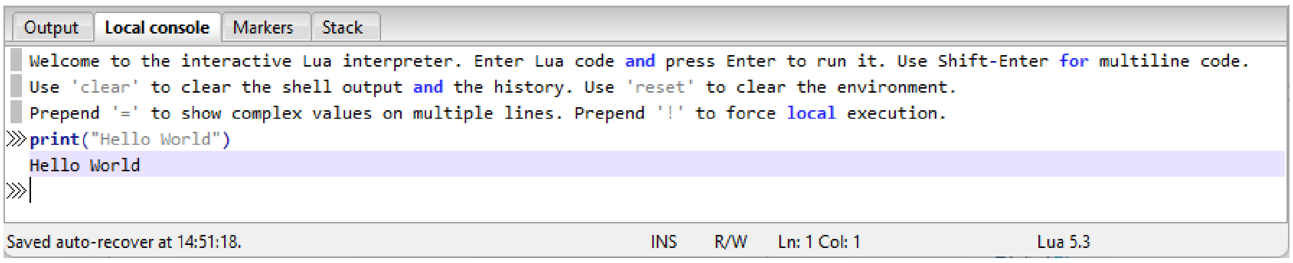
Direct Lua Commands

If you have used Python before, you will be familiar with the ‘Shell’ which allows you to type commands in one at a time.

The same can be done with Lua.
Click on the ‘Local console’ tab.

Type: `print(“Hello World”)`

Press enter. The words “Hello World” appear in the console:



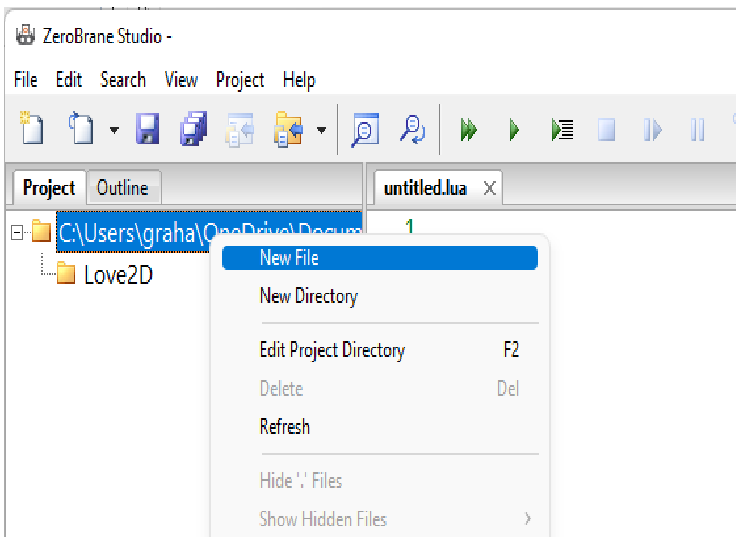
```
Welcome to the interactive Lua interpreter. Enter Lua code and press Enter to run it. Use Shift-Enter for multiline code.
Use 'clear' to clear the shell output and the history. Use 'reset' to clear the environment.
Prepend '=' to show complex values on multiple lines. Prepend '!' to force local execution.
>>print("Hello World")
Hello World
>>
```

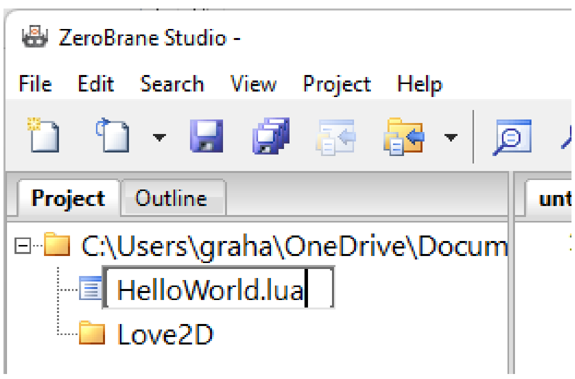
Saved auto-recover at 14:51:18. INS R/W Ln: 1 Col: 1 Lua 5.3

The line is temporarily saved in memory, and can be recalled using the up arrow on the keyboard, but to save it permanently it needs to be saved to a file, exactly the same as Python.

The Statutory “Hello World” script

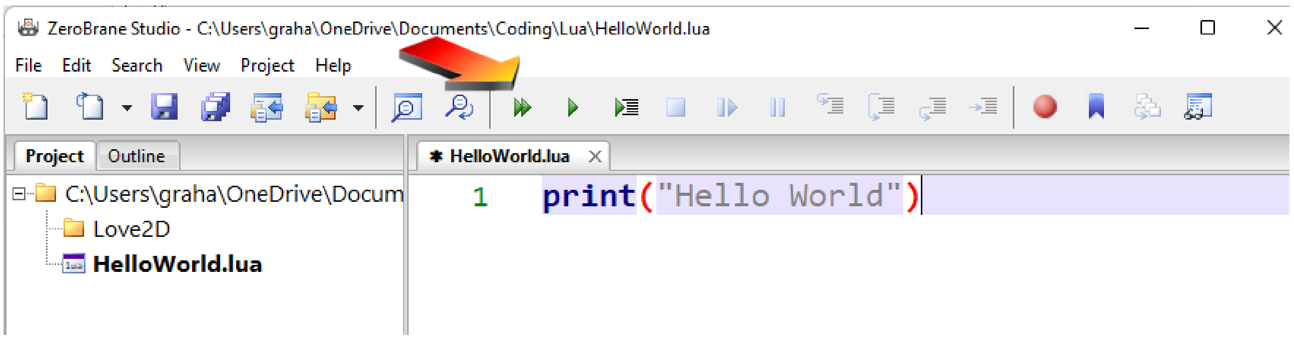
Right-Click on your project folder name and select ‘New File’. Type “HelloWorld.lua” into the empty box and press ‘Enter’:




ZeroBrane Studio -
File Edit Search View Project Help
Project Outline
C:\Users\graha\OneDrive\Docum
HelloWorld.lua
Love2D

Double-Click on the new file to bring it into the editor.
If you have an open file “untitled.lua” → close it.

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


ZeroBrane Studio - C:\Users\graha\OneDrive\Documents\Coding\Lua\HelloWorld.lua
File Edit Search View Project Help
Project Outline
HelloWorld.lua
C:\Users\graha\OneDrive\Docum
Love2D
HelloWorld.lua

```
print("Hello World")
```

Click the double green triangles indicated above. This will automatically save the code and run it.

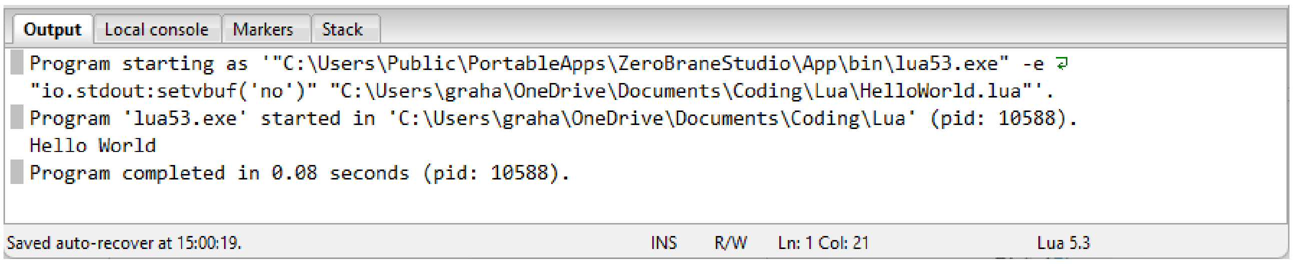
If you see a message in the Output tab similar to this:

```
Can't find 'main.lua' file in the current project folder: 'O:\Documents\Coding\Lua'.
```

Take the following steps:

Menu: Project → Lua Interpreter → Lua 5.3

Try again:



```
Program starting as '"C:\Users\Public\PortableApps\ZeroBraneStudio\App\bin\lua53.exe" -e
"io.stdout:setvbuf('no')" "C:\Users\graha\OneDrive\Documents\Coding\Lua\HelloWorld.lua"'.
Program 'lua53.exe' started in 'C:\Users\graha\OneDrive\Documents\Coding\Lua' (pid: 10588).
Hello World
Program completed in 0.08 seconds (pid: 10588).
```

Saved auto-recover at 15:00:19. INS R/W Ln: 1 Col: 21 Lua 5.3

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).

If you run this file in a console, it will close before you can see the results, so alter it as below:

Section 1

https://github.com/Inksaver/LuaForSchools/tree/main/Beginners/Section1

01-HelloWorld1.lua:

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section1/01-Hello%20World1.lua

```
-- obligatory 'Hello World' when learning a new language
print("Hello World")

-- Tell the user to press Enter to quit
-- Only needed if running in a console to stop it closing, NOT in IDE
print("Press Enter to quit")
io.read()
```

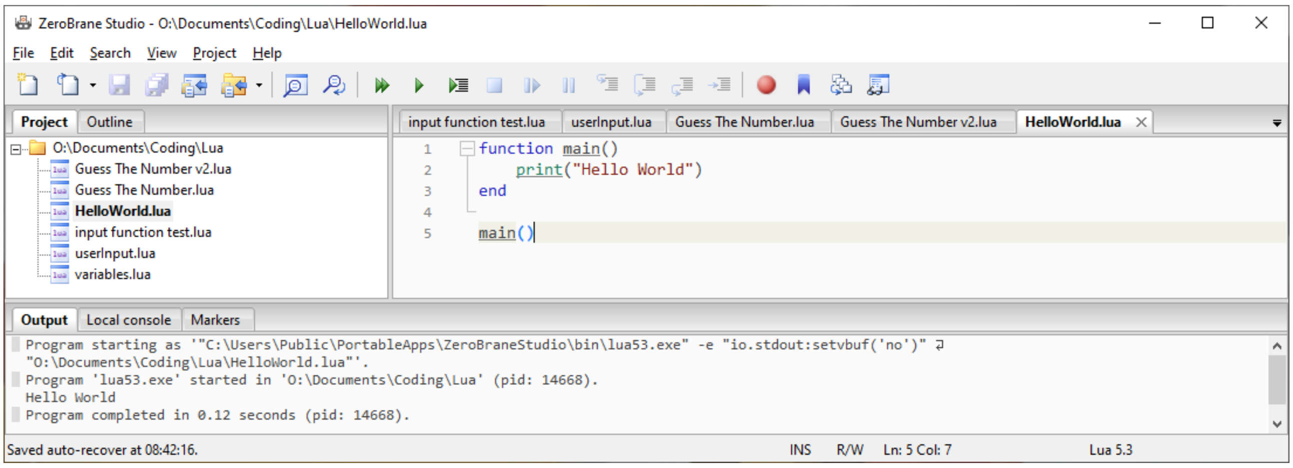
You will soon be using the Love2D engine, which is heavily based on procedural programming.
To get you started, use the script below:

02-HelloWorld2.lua:

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section1/02-Hello%20World2.lua

```
-- demonstrates use of a function called main() to copy C, C++, C# and Java
-- functions are ignored until 'called'

function main()
    print("Hello World Version2")
end

main() -- 'Call' function main()

print("Press Enter to quit")
io.read()
```

The last 2 lines are not required in ZeroBrane, but are needed to run in a console


ZeroBrane Studio - O:\Documents\Coding\Lua\HelloWorld.lua
File Edit Search View Project Help
Project Outline
O:\Documents\Coding\Lua
Guess The Number v2.lua
Guess The Number.lua
HelloWorld.lua
input function test.lua
userInput.lua
variables.lua
input function test.lua userInput.lua Guess The Number.lua Guess The Number v2.lua HelloWorld.lua
Output Local console Markers
INS R/W Ln: 5 Col: 7 Lua 5.3

```
1  function main()
2      print("Hello World")
3  end
4
5  main()
```

```
Program starting as '"C:\Users\Public\PortableApps\ZeroBraneStudio\bin\lua53.exe" -e "io.stdout:setvbuf('no')"
"O:\Documents\Coding\Lua\HelloWorld.lua"'.
Program 'lua53.exe' started in 'O:\Documents\Coding\Lua' (pid: 14668).
Hello World
Program completed in 0.12 seconds (pid: 14668).
```

Saved auto-recover at 08:42:16.

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 they are.
(Python works in the same way with 'def')

When it reaches line 5 (main()), this is a call to the main() function found at line 1, so the script jumps from line 5, which is the starting point, to line 1, the main() function.

The main() function has only one line of code: `print("Hello World")`, which it executes, then control is returned back to line 5, 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 the `print()` procedure, 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.

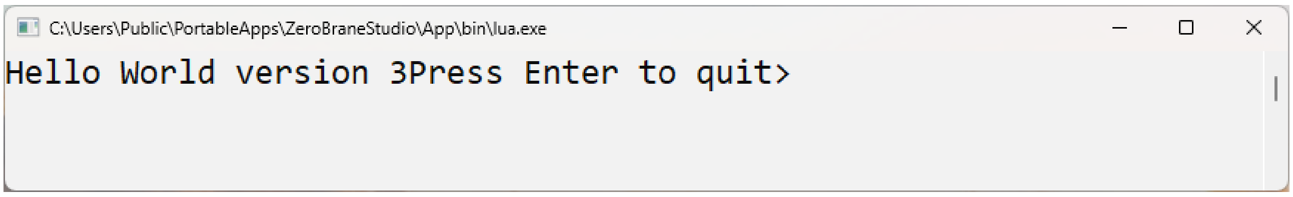
The next 3 files demonstrate this, along with the use of "escape characters" which help with the formatting of output.

03-HelloWorld3.lua:

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section1/03-Hello%20World3.lua

```lua
-- demonstrates io.write(). All output is on 1 line
-- use F5 to step through in IDE

function main()
	io.write("Hello ")
	io.write("World ")
	io.write("version 3")
end

main()

io.write("Press Enter to quit>")
io.read()
```

Console output: Note how all text is on one line



C:\Users\Public\PortableApps\ZeroBraneStudio\App\bin\lua.exe

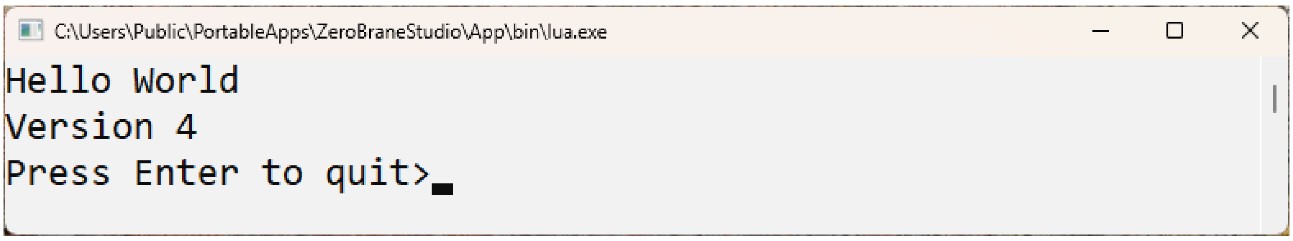
```
Hello World version 3Press Enter to quit>
```

04-HelloWorld4.lua:

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section1/04-Hello%20World4.lua

```lua
-- demonstrates io.write() with \n newline character

function main()
	io.write("Hello ")
	io.write("World \n") -- note \n
	io.write("Version 4\n")
end

main()

io.write("Press Enter to quit>")
io.read()
```

Console output:



C:\Users\Public\PortableApps\ZeroBraneStudio\App\bin\lua.exe

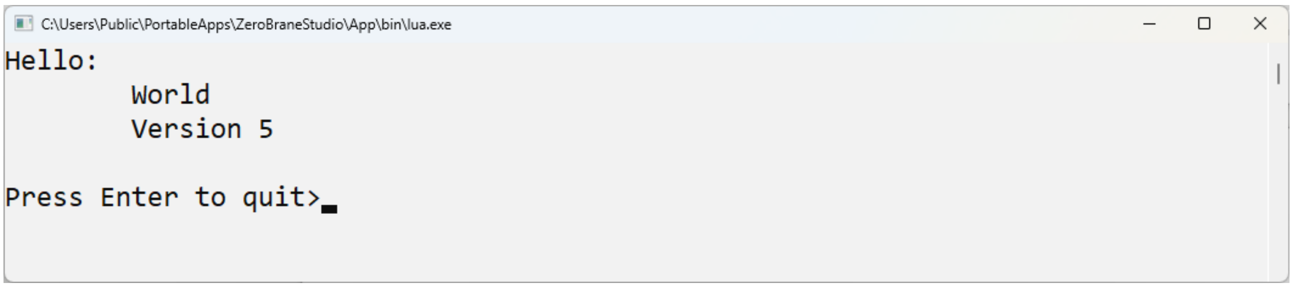
```
Hello World
Version 4
Press Enter to quit>
```

05-HelloWorld5.lua:

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section1/05-Hello%20World5.lua

```lua
-- demonstrates print(), io.write()
-- with \n newline and \t Tab

function main()
      print("Hello:")
      print("\tWorld ")
      io.write("\tVersion 5")
end

main()

io.write("\n\nPress Enter to quit>") -- note 2 newlines
io.read()
```

Console Output:



```
C:\Users\Public\PortableApps\ZeroBraneStudio\App\bin\lua.exe
Hello:
        World
        Version 5

Press Enter to quit>_
```

The backslash before n represents the newline character. Similar hidden characters you can play with are:

```
1. \'    single quote:    embed a single quote in the output string
2. \"    double quote:    embed a double quote in the output string
3. \\    backslash:       embed a backslash in the output string
4. \n    new line:        move cursor to a new line
5. \r    carriage return: move cursor back to the beginning of the line
6. \t    tab:             move cursor forward (usually) 4 spaces
7. \b    backspace:       move cursor back one space
8. \f    form feed:       move cursor down to next line
```

Many of these 'Escape Characters' are based on the actions of mechanical typewriters.

The typewriter had a tab key, which moved the 'carriage' (the roller that held the paper) to the left by 4 spaces, so the next character position was effectively moved 4 spaces to the right.

There was also a 'backspace' key which moved the carriage one space to the right, so you could then insert a thin piece of white tape to over-print the incorrect character, and allow you to type the correct one. The resulting mess was acceptable at the time.
The console backspace works in the same way: it moves the cursor back one space, but does NOT delete the character in front of it.

There was a prominent lever which did two things:

1. Carriage Return: which pushed the carriage fully to the right, so the left edge of the paper was under the print head

2. Line Feed (Form feed) which rotated the roller to move the paper upwards, so the print head was on a new line.

The newline character \n is effectively a combination of ‘carriage return’ (cr) and ‘line feed’ (lf) so it moves the virtual carriage to the left edge of the console, then moves it down by one line.

Windows uses both cr and lf to move to the next line, Unix, Linux and others use just use one of them, so ‘newline’ covers all operating systems.

UTF8 characters

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

utf8 box characters stored here for easy copy/paste:

```
┌ ┬ ┐ ─ ╔ ╦ ╗ ═
├ ┼ ┤ │ ╠ ╬ ╣ ║
└ ┴ ┘   ╚ ╩ ╝
```

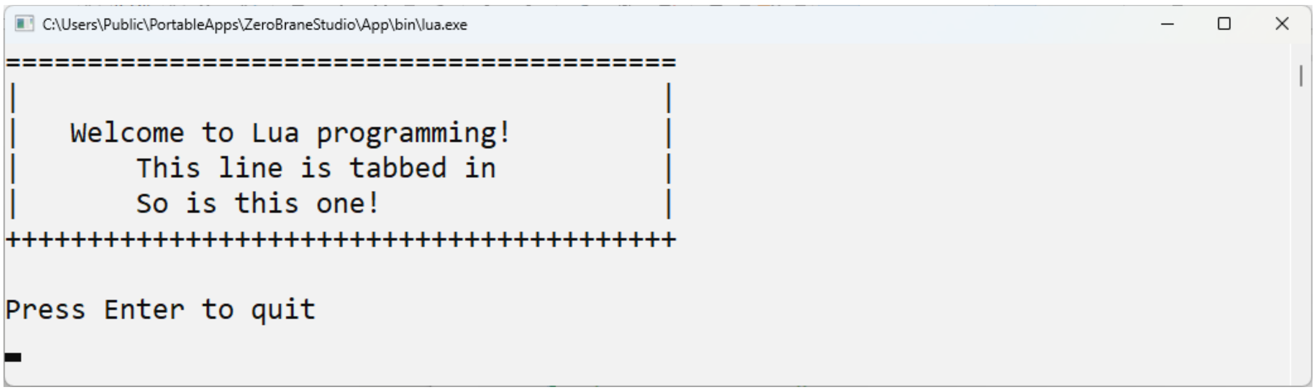
These characters can improve your console based projects in Lua, Python, C# and many other languages They cannot be typed in, so are best copy/pasted into your code.

06-Assignment 1.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section1/06-Assignment%201.lua

```lua
--[[
https://www.w3schools.com/charsets/ref_utf_box.asp
utf8 box characters stored here for easy copy/paste:

┌ ┬ ┐ ─ ╓ ╥ ╖ ═

├ ┼ ┤ │ ╟ ╫ ╢ ║

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

--[[
                                           Pre Windows 10 (1909) will not display UTF8
╔═══════════════════════════════════════╗  =========================================
║                                       ║  |                                       |
║    Welcome to Lua programming!        ║  |    Welcome to Lua programming!        |
║        This line is tabbed in x2      ║  |        This line is tabbed in x2      |
║        So is this one!                ║  |        So is this one!                |
╚═══════════════════════════════════════╝  +++++++++++++++++++++++++++++++++++++++++
]]

-- Use either print() or io.write()
-- (remember io.write() does NOT move to a new line
-- Also use \t to line up the text (or spaces)
function main()
    -- your code goes here
end

main()
print("Press Enter to quit")
io.read()
```

Use the print(), io.write() functions and escape characters to get this output. This one using ASCII:


C:\Users\Public\PortableApps\ZeroBraneStudio\App\bin\lua.exe

```
=========================================
|                                       |
|    Welcome to Lua programming!        |
|        This line is tabbed in         |
|        So is this one!                |
+++++++++++++++++++++++++++++++++++++++++

Press Enter to quit
▬
```

You can use the UTF8 characters for fancy box construction:

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

Copy / paste the characters into your code.

They will NOT display in Windows console below version 10 (1909) except in ConEmu emulator. They will display inside Zerobrane.

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’

--[[[[This comment allows the use of ‘[[‘ inside it,
      By adding 2 more square brackets at the start.]]
```

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:

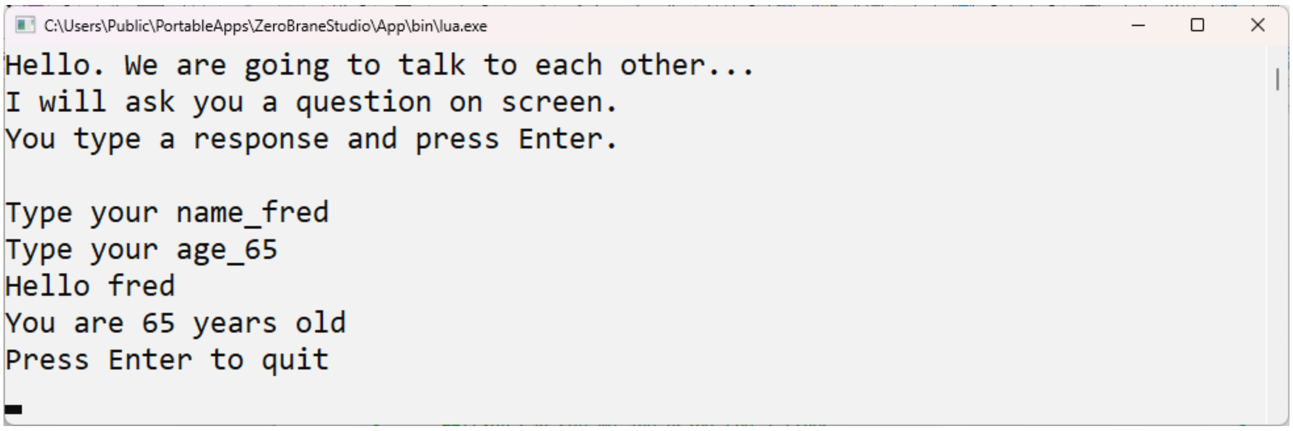
```
typedInText = io.read()
```

07-Input1.lua:

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section1/07-Input1.lua

```
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()

print("Press Enter to quit")
io.read()
```

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


C:\Users\Public\PortableApps\ZeroBraneStudio\App\bin\lua.exe

```
Hello. We are going to talk to each other...
I will ask you a question on screen.
You type a response and press Enter.

Type your name_fred
Type your age_65
Hello fred
You are 65 years old
Press Enter to quit
```

Note the repeated use of:

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

This needs another function, to replicate Python's `input()`

08-Input2.lua:

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section1/08-Input2.lua

```
-- does exacly the same as 07-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()

input("Press Enter to quit") -- this is now using the new input() function
```

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

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

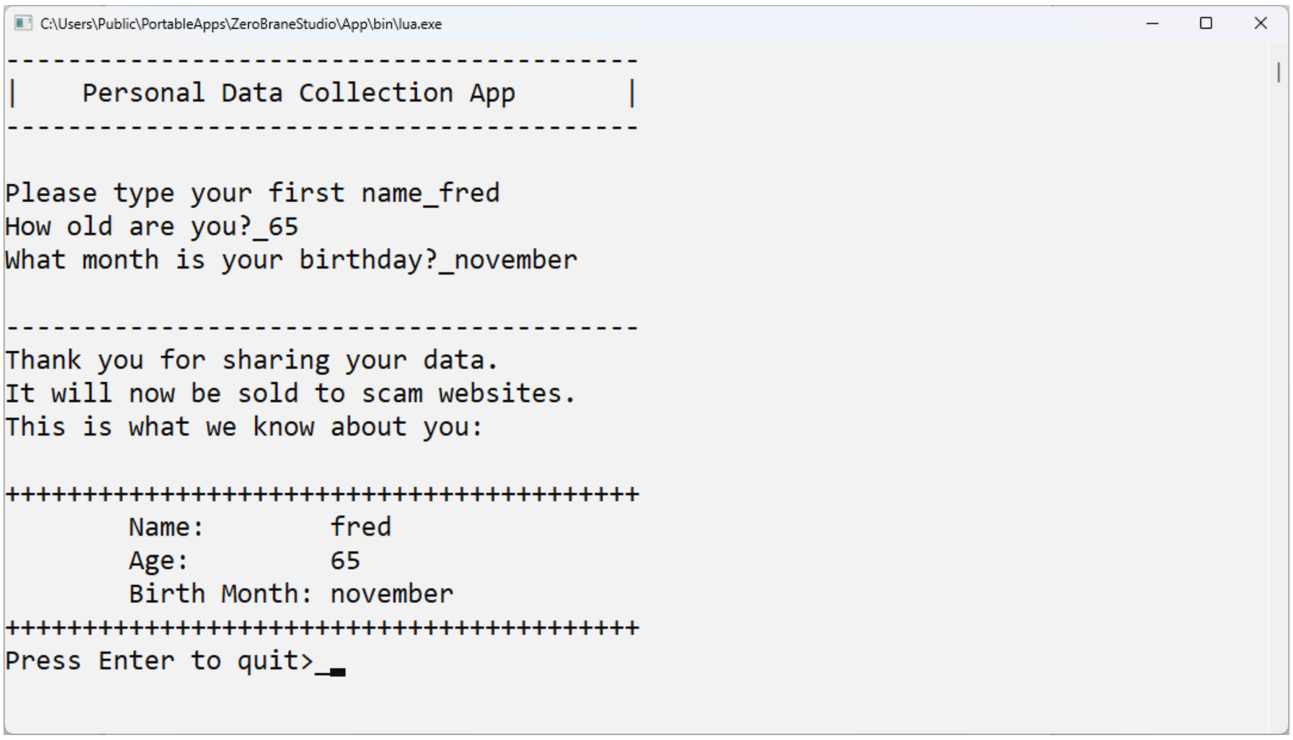
The only problem here is there is no validation of what the user is typing.
You might want a number, but that is not checked, or you might want a string of a particular length.

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

09-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 also use UTF8 characters:



```
C:\Users\Public\PortableApps\ZeroBraneStudio\App\bin\lua.exe
-----------------------------------------
|    Personal Data Collection App        |
-----------------------------------------

Please type your first name_fred
How old are you?_65
What month is your birthday?_november

-----------------------------------------
Thank you for sharing your data.
It will now be sold to scam websites.
This is what we know about you:

+++++++++++++++++++++++++++++++++++++++++
        Name:         fred
        Age:          65
        Birth Month:  november
+++++++++++++++++++++++++++++++++++++++++
Press Enter to quit>_
```

```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

input("Press Enter to quit>")
```

Section 2

https://github.com/Inksaver/LuaForSchools/tree/main/Beginners/Section2

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

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/01-Variables-integer1.lua

```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("tostring(myNumber) = "..tostring(myNumber))
end

main()

input("Press Enter to quit")
```

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: `tostring(value)`

02-Variables-integer2.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/02-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 string with nil

```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()

input("Press Enter to quit")
```

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()

input("Press Enter to quit")
```

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

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/04-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.

```lua
-- 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()

input("Press Enter to quit")
```

05-IfElseifElse.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/05-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()

input("Press Enter to quit")
```

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:

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

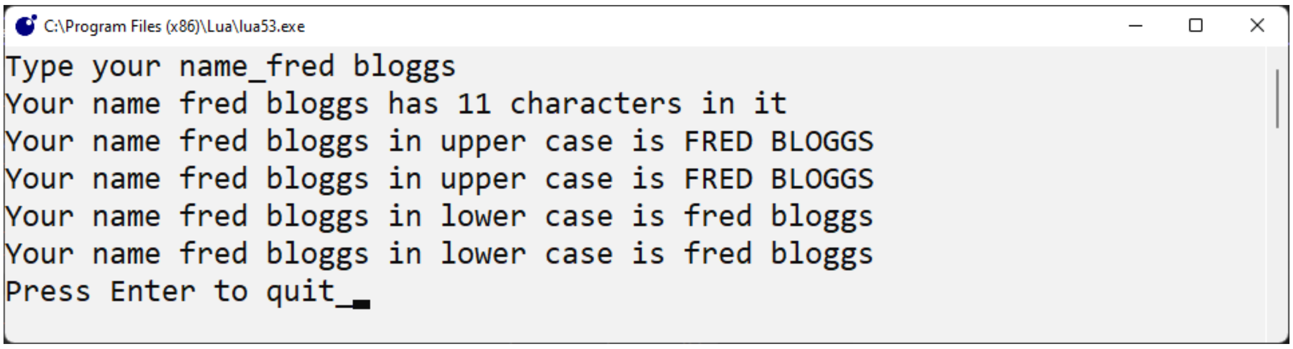
can be re-written as

```
value:upper()
```

06-Strings.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/06-Strings.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")
    -- # returns the length of the string inside the brackets as an INTEGER
    numberOfChars = #(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()

input("Press Enter to quit")
```



```
Type your name_fred bloggs
Your name fred bloggs has 11 characters in it
Your name fred bloggs in upper case is FRED BLOGGS
Your name fred bloggs in upper case is FRED BLOGGS
Your name fred bloggs in lower case is fred bloggs
Your name fred bloggs in lower case is fred bloggs
Press Enter to quit_
```

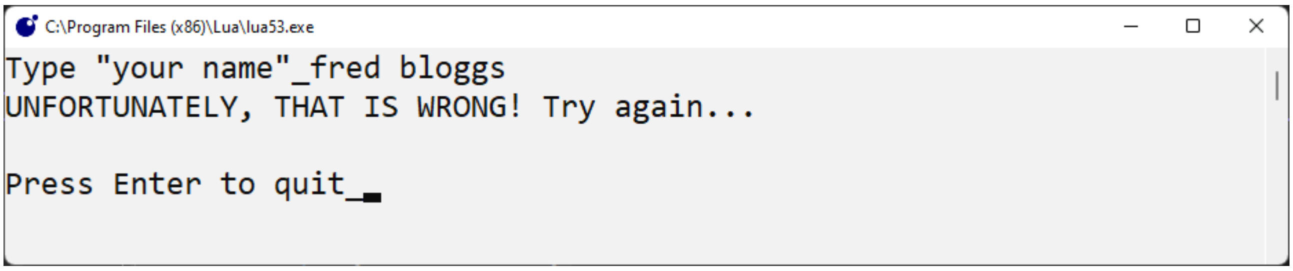
07-Strings2.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/07-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()

input("Press Enter to quit")
```

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



C:\Program Files (x86)\Lua\lua53.exe

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

Press Enter to quit_
```

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

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/08-Loop1.lua

```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

    input("Press Enter to quit")
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

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/09-Loop2.lua

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

```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

    input("Press Enter to quit")
end

main()
```

The symbol `~=` means ‘not equal’

The line

```
    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

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/10-Loop3.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()
input("Press Enter to quit")
```

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.

There is a version of Magic Ball using multiple `if elseif else` statements at:

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section2/11-MagicBall.lua

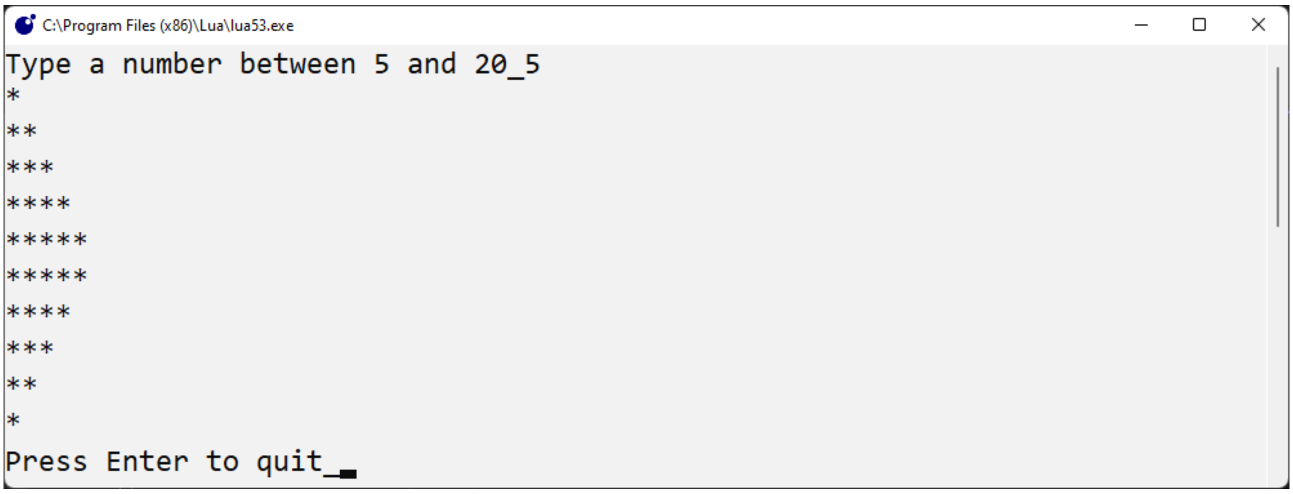
(Code not included in this document.)

Section 3

01-ForLoops.lua Drawing triangles with ASCII

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section3/01-ForLoops.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

input("Press Enter to quit")
```



```
Type a number between 5 and 20_5
*
**
***
****
*****
*****
****
***
**
*
Press Enter to quit_
```

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
```

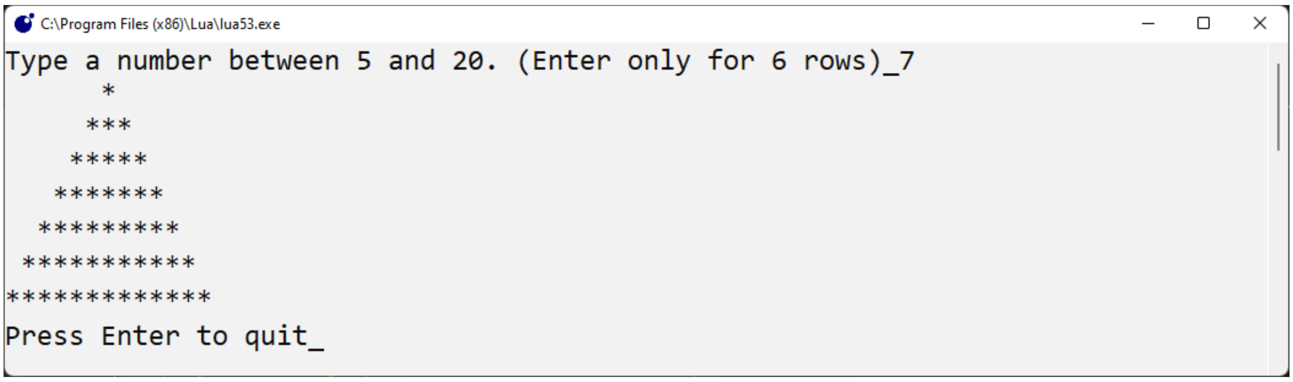
this runs the loop in reverse. The counter starts at the highest value, and drops by -1 each iteration until it reaches 1.
The "step" is -1 but can be set to any integer value.
General 'for loop' construction:

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

02-ForLoopsAssignment.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section3/02-ForLoops%20Assignment.lua

Using the template below which draws the left side of a triangle, produce the following output:



```
Type a number between 5 and 20. (Enter only for 6 rows)_7
      *
     ***
    *****
   *******
  *********
 ***********
*************
Press Enter to quit_
```

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

function main()
    numberOfRows = input("Type a number between 5 and 20. (Enter only for 6 rows)")
    numberOfRows = tonumber(numberOfRows)
    if numberOfRows == nil then
        numberOfRows = 6 -- user has chosen the default
    end
    -- draw a triangle
    -- for variable = start, finish, step do
    for i = 1, numberOfRows, 1 do
        -- string.rep() repeats the character(s) given by the number supplied
        -- eg string.rep("*", 4) returns "****"
        lineOfChars = string.rep("*", i)
        print(lineOfChars)
    end
end

main() -- program starts here

input("Press Enter to quit")
```

Hints:

This example is 7 rows deep. You could do this in a number of ways:

1. Lazy: write chosen number string variables with the correct layout, then print them: 1 mark!

2. Better: create chosen number string variables using code to fill them, then print them: 2 marks!
   (you could use a combination of string.rep() and concatenation -> ..)
   eg row1 = string.rep(" ", 4) .. "*" .. string.rep(" ", 4)

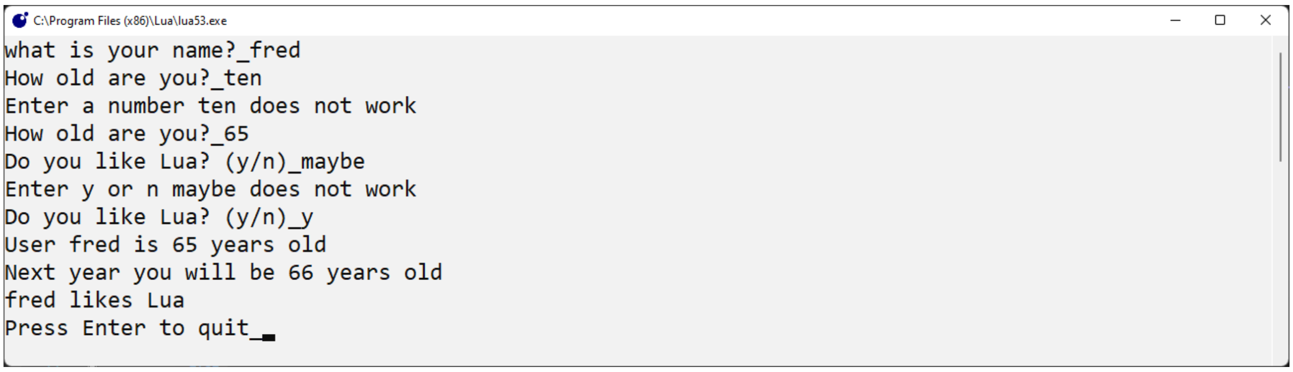
3. Use a for loop to draw both halves of the tree one row at a time similar to 2.
   (join the string.rep(# ,#) combinations in the for loop)
   but able to cope with as many rows as the user chooses: 10 marks!

Validating User Input

03-InputValidation.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section3/03-InputValidation.lua

```lua
function input(prompt, dataType)    -- get input from user
    dataType = dataType or "string" -- if not supplied then give default value
    while true do                   -- break not required as return used instead
        io.write(prompt .. "_")
        local userInput = io.read()
        if dataType == "string" then
            return userInput
        elseif dataType == "int" or dataType == "float" then
            if tonumber(userInput) ~= nil then
                return tonumber(userInput)
            else
                print("Enter a number ".. userInput .. " does not work")
            end
        elseif dataType == "bool" then
            if userInput == "y" or userInput == "yes" then
               return true
            elseif userInput == "n" or userInput == "no" then
               return false
            else
               print("Enter y or n ".. userInput .. " does not work")
            end
        end
    end
end

function main()
    name = input("what is your name?")
    age = input("How old are you?", "int")
    likesLua = input("Do you like Lua? (y/n)", "bool")

    print("User "..name.." is "..age.. " years old")
    print("Next year you will be ".. age + 1 .. " years old")
    if likesLua then
        print(name.." likes Lua")
    else
        print(name.." does not like Lua")
    end
end

main() -- program starts here
input("Press Enter to quit")
```



```
what is your name?_fred
How old are you?_ten
Enter a number ten does not work
How old are you?_65
Do you like Lua? (y/n)_maybe
Enter y or n maybe does not work
Do you like Lua? (y/n)_y
User fred is 65 years old
Next year you will be 66 years old
fred likes Lua
Press Enter to quit_
```

The input function has been expanded to take an additional parameter: `function input(prompt, dataType)`
This allows the function to check whether the input is numerical (if a number is required), or a yes/no answer if a boolean (true / false) is needed.

To use it, supply either “string”, “int”, “float” or “bool” after the prompt text.
If nothing is supplied the line `dataType = dataType or "string"` will use the default value “string”

This is an improvement, but cannot distinguish between integer or real numbers, and will return an empty string if the user simply presses the enter key.

Note the use of the local keyword on `local userInput = io.read()`

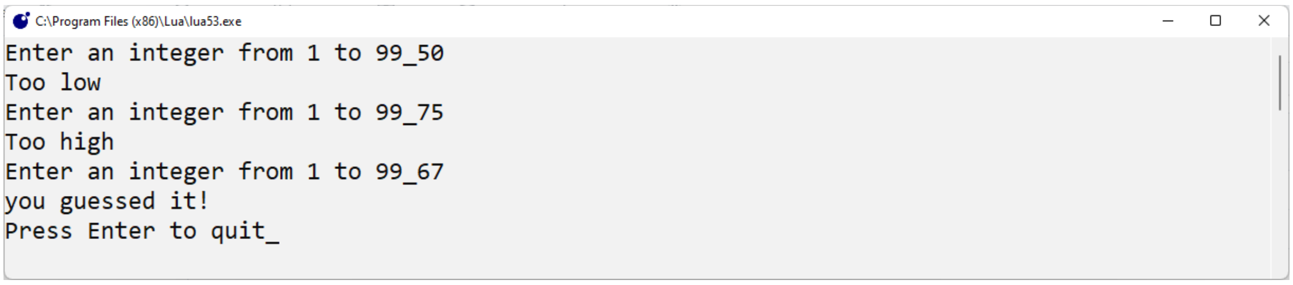
Lua recommends using the local keyword whenever possible, so it is best to start using it at an early stage. Further information at the end of this tutorial.

The input function now has an infinite while loop.
User input is checked to see if it is valid. If so the value is returned
If not the loop continues.

The dataType of the returned value is either string, number or boolean, so can be safely be used directly in the calling code.

04-GuessTheNumber.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section3/04-GuessTheNumber.lua



```
Enter an integer from 1 to 99_50
Too low
Enter an integer from 1 to 99_75
Too high
Enter an integer from 1 to 99_67
you guessed it!
Press Enter to quit_
```

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)
```

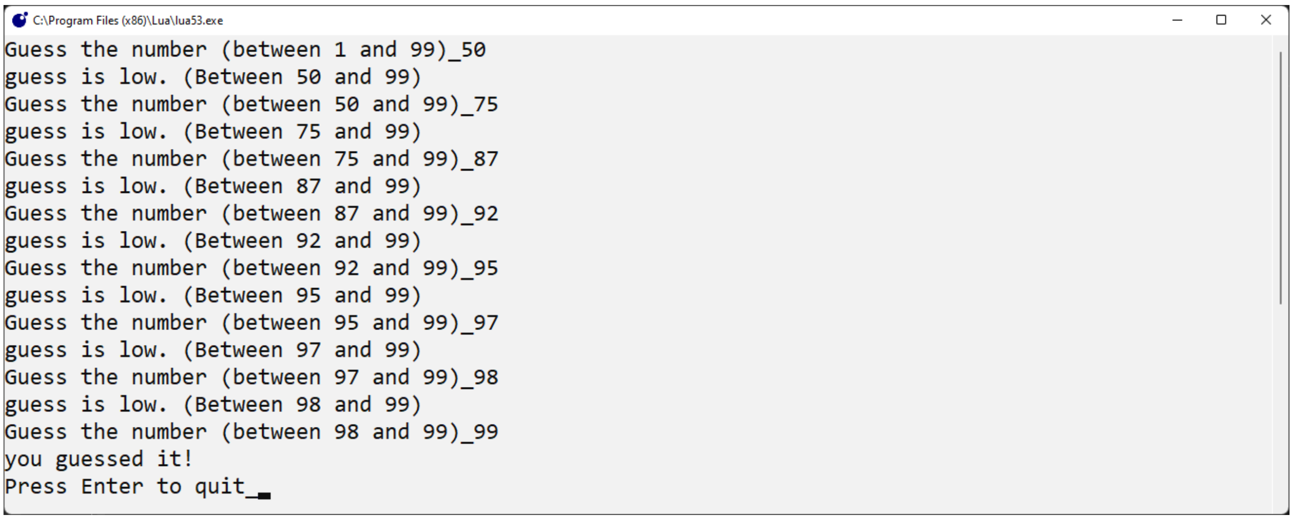
This simple game uses the random number function to create a number between 1 and 99
The user is asked to guess the number using the input() function developed earlier inside a repeat-until loop.

```
local function input(prompt, dataType) -- get input from user
    dataType = dataType or "string"    -- if not supplied then give default value
    while true do                      -- break not required as return used instead
        io.write(prompt .. "_")
        local userInput = io.read()
        if dataType == "string" then
            return userInput
        elseif dataType == "int" or dataType == "float" then
            if tonumber(userInput) ~= nil then
                return tonumber(userInput)
            else
                print("Enter a number ".. userInput .. " does not work")
            end
        elseif dataType == "bool" then
            if userInput == "y" or userInput == "yes" then
                return true
            elseif userInput == "n" or userInput == "no" then
                return false
            else
                print("Enter y or n ".. userInput .. " does not work")
            end
        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 = input("Enter an integer from 1 to 99", "int")
        if guess < n then
            print("Too low")
        elseif guess > n then
            print("Too high")
        else
            print("you guessed it!")
        end
    until guess == n
    input("Press Enter to quit")
end
```

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:



```
Guess the number (between 1 and 99)_50
guess is low. (Between 50 and 99)
Guess the number (between 50 and 99)_75
guess is low. (Between 75 and 99)
Guess the number (between 75 and 99)_87
guess is low. (Between 87 and 99)
Guess the number (between 87 and 99)_92
guess is low. (Between 92 and 99)
Guess the number (between 92 and 99)_95
guess is low. (Between 95 and 99)
Guess the number (between 95 and 99)_97
guess is low. (Between 97 and 99)
Guess the number (between 97 and 99)_98
guess is low. (Between 98 and 99)
Guess the number (between 98 and 99)_99
you guessed it!
Press Enter to quit_
```

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 4 Keyboard Input Library (Simple)

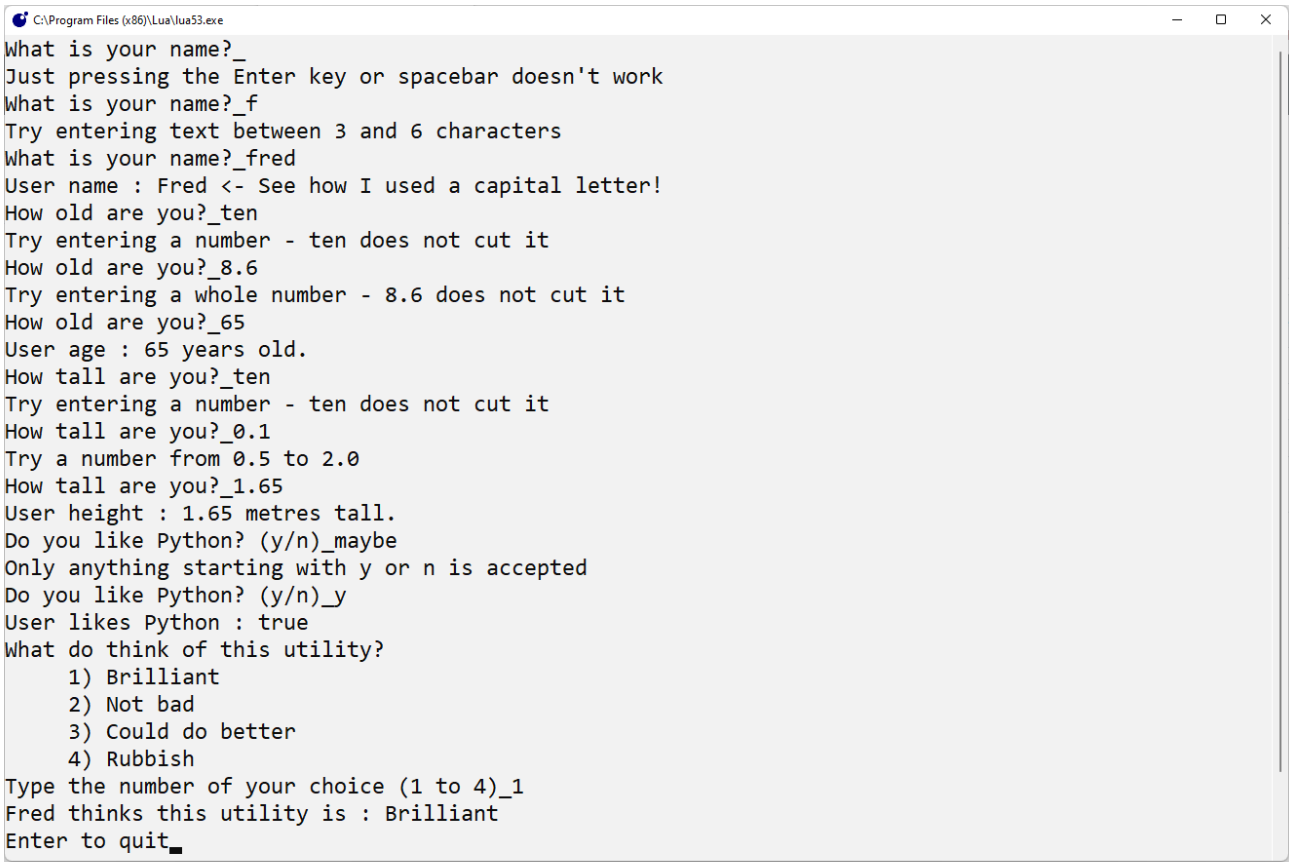
https://github.com/Inksaver/LuaForSchools/tree/main/Beginners/Section4

Download or copy/paste BOTH files kboard.lua and main.lua and make sure they are both in the same directory.

The file kboard.lua is a simple library to obtain reliable input data.

It does not run on its own.

The file main.lua is the test file to make use of the kboard library:



```
What is your name?_
Just pressing the Enter key or spacebar doesn't work
What is your name?_f
Try entering text between 3 and 6 characters
What is your name?_fred
User name : Fred <- See how I used a capital letter!
How old are you?_ten
Try entering a number - ten does not cut it
How old are you?_8.6
Try entering a whole number - 8.6 does not cut it
How old are you?_65
User age : 65 years old.
How tall are you?_ten
Try entering a number - ten does not cut it
How tall are you?_0.1
Try a number from 0.5 to 2.0
How tall are you?_1.65
User height : 1.65 metres tall.
Do you like Python? (y/n)_maybe
Only anything starting with y or n is accepted
Do you like Python? (y/n)_y
User likes Python : true
What do think of this utility?
     1) Brilliant
     2) Not bad
     3) Could do better
     4) Rubbish
Type the number of your choice (1 to 4)_1
Fred thinks this utility is : Brilliant
Enter to quit_
```

This is an expansion on the code used in 03-InputValidation.lua on page 26 above.

kboard.lua can be used in any of your own projects by adding the file to the project folder and using

kb = require “kboard” at the top of your starting file.

This is how it is done in main.lua

main.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section4/main.lua

```lua
kb = require "kboard"

local function quit()
    io.write("Enter to quit")
    io.read()
end

local function testKboard()
    local name = kb.GetString("What is your name?", true, 3, 6)
    print("User name : "..name.." <- See how I used a capital letter!")

    local age = kb.GetInteger("How old are you?", 5, 110)
    print("User age : ".. age.. " years old.")

    local height = kb.GetFloat("How tall are you?", 0.5, 2.0)
    print("User height : ".. height.." metres tall.")

    local likesPython = kb.GetBoolean("Do you like Python? (y/n)")
    print("User likes Python : "..tostring(likesPython))
    kb.Sleep(2)-- pause 2 secs

    local title = "What do think of this utility?"
    local options = {"Brilliant", "Not bad", "Could do better", "Rubbish"}
    local choice = kb.Menu(title, options)
    print(name.." thinks this utility is : "..options[choice])
end

local function main()
    --[[ Everything runs from here ]]
    testKboard()
    quit()
end
main()
```

The kboard library is basically a table, and is imported (as in Python) into the main file with the name 'kb'

You can use any of it's functions by calling them, and passing parameters to qualify what you want to be returned.

```lua
    local name = kb.GetString("What is your name?", true, 3, 6)
```

Returns a string of between 3 and 6 characters in length, in Title Case, and stores it in the variable name.

Demonstrated here is the ability to get a guaranteed return value of the correct dataType:

• string	`kb.GetString(prompt, withTitle, minValue, maxValue)`
• integer	`kb.GetInteger(prompt, minValue, maxValue)`
• float	`kb.GetFloat(prompt, minValue, maxValue)`
• boolean	`kb.GetBoolean(prompt)`

Also a Menu option which guarantees an integer index value from a list (table) supplied to the library

```lua
    local options = {"Brilliant", "Not bad", "Could do better", "Rubbish"}
    local choice = kb.Menu(title, options)
    Get the option from the index -> options[choice]
```

The kboard.lua code is shown here for interest only. There are parts that you may find difficult to understand at this stage, but you do not need to know in order to use it.

```lua
--[[
kboard static class returns string, integer, float, boolean and menu choices.
Use:

name = Kboard.GetString("What is your name?", true, 1, 10, row);
age = Kboard.GetInteger("How old are you", 5, 110, row);
height = Kboard.GetRealNumber("How tall are you?", 0.5, 2.0, row);
likesPython = Kboard.GetBoolean("Do you like C#? (y/n)", row);

options = {"Brilliant", "Not bad", "Could do better", "Rubbish"};
choice = Kboard.Menu("What do think of this utility?", options, row)
]]
local Kboard = {}
local blank = string.rep(" ", 79)
local delay = 2

--[[from http://lua-users.org/wiki/StringRecipes
    Change an entire string to Title Case (i.e. capitalise the first letter of each word)
    Add extra characters to the pattern if you need to. _ and ' are
    found in the middle of identifiers and English words.
    We must also put %w_' into [%w_'] to make it handle normal stuff
    and extra stuff the same.
    This also turns hex numbers into, eg. 0Xa7d4

    str = str:gsub("(%a)([%w_']*)", tchelper)
]]
local function tchelper(first, rest)
   return first:upper()..rest:lower()
end

--[[ local functions are private, not called directly from other files ]]

local function processInput(prompt, minValue, maxValue, dataType)
    --[[ validate input, raise error messages until input is valid ]]
    local validInput = false
    local userInput
    while not validInput do
        io.write(prompt.."_")
        userInput = io.read()
        local output = userInput
        if dataType == "string" then
            if userInput:len() == 0 and minValue > 0 then
                print("Just pressing the Enter key or spacebar doesn't work")
            else
                if userInput:len() < minValue or userInput:len() > maxValue then
                    print("Try entering text between "..minValue.." and "..maxValue.." characters")
                else
                    validInput = true
                end
            end
        else
            if userInput:len() == 0 then
                print("Just pressing the Enter key or spacebar doesn't work")
            else
                if dataType == "bool" then
                    if userInput:sub(1, 1):lower() == "y" then
                        userInput = true
                        validInput = true
                    elseif userInput:sub(1, 1):lower() == "n" then
                        userInput = false
                        validInput = true
                    else
                        print("Only anything starting with y or n is accepted")
                    end
                else
                    if dataType == "int" or dataType == "float" then
                        local input = tonumber(userInput)
                        if input == nil then
                            print("Try entering a number - "..userInput.." does not cut it")
                        else
                            if input >= minValue and input <= maxValue then
                                if math.floor(input / 1) ~= input and dataType == "int"  then
                                    print("Try entering a whole number - "..userInput.." does not cut it")
                                else
                                    validInput = true
                                    userInput = input
                                end
                            else
                                print("Try a number from "..minValue.." to "..maxValue)
                            end
                        end
                    end
                end
            end
        end
    end
    return userInput
end

function Kboard.GetString(prompt, withTitle, minValue, maxValue)
    --[[ Return a string. withTitle, minValue and maxValue are given defaults if not passed ]]
    withTitle = withTitle or false
    minValue = minValue or 1
    maxValue = maxValue or 20
    local userInput = processInput(prompt, minValue, maxValue, "string")
    if withTitle then
        userInput = Kboard.ToTitle(userInput)
    end
    return userInput
end

function Kboard.GetFloat(prompt, minValue, maxValue)
    --[[ Return a real number. minValue and maxValue are given defaults if not passed ]]
    minValue = minValue or -10000000000
    maxValue = maxValue or  10000000000

    return processInput(prompt, minValue, maxValue, "float")
end

function Kboard.GetInteger(prompt, minValue, maxValue)
    --[[ Return an integer. minValue and maxValue are given defaults if not passed ]]
    minValue = minValue or 0
    maxValue = maxValue or 65536
    return processInput(prompt, minValue, maxValue, "int")
end

function Kboard.GetBoolean(prompt)
    --[[ Return a boolean. Based on y(es)/ n(o) response ]]
    return processInput(prompt, 1, 3, "bool", row)
end

function Kboard.ToTitle(Text)
    --[[ converts any string to Title Case ]]
    return Text:gsub("(%a)([%w_']*)", tchelper)
end

function Kboard.Sleep(s)
    --[[ Lua version of Python time.sleep(2) ]]
    local sec = tonumber(os.clock() + s);
    while (os.clock() < sec) do end
end

function Kboard.Menu(title, list)
    --[[ displays a menu using the text in 'title', and a list of menu items (string) ]]
    print(title)
    local index = 1
    for _, item in ipairs(list) do
        if index < 10 then
            print("     "..index..") "..item)
        else
            print("    "..index..") "..item)
        end
        index = index + 1
    end
    return Kboard.GetInteger("Type the number of your choice (1 to "..index-1 ..")", 1, #list)
end

return Kboard
```

Section 5

Lua Tables

https://github.com/Inksaver/LuaForSchools/tree/main/Beginners/Section5

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:

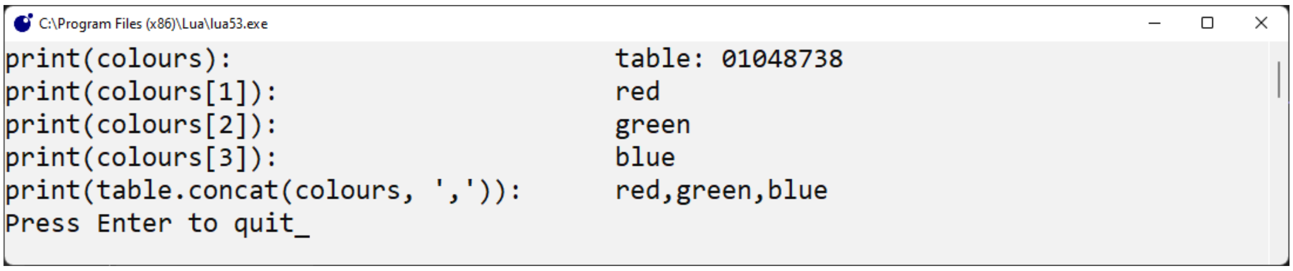
```
myFriends = [“Fred”, “Alice”, “Jim”, “Karen”]

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

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section5/01-Tables1.lua

```lua
-- introduction to tables

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

function main()
    -- your code here
    -- 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
input("Press Enter to quit")
```



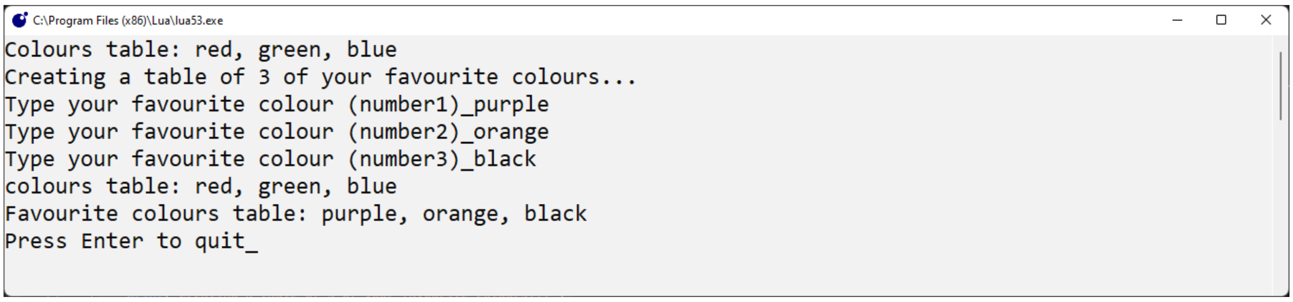
```
C:\Program Files (x86)\Lua\lua53.exe
print(colours):                         table: 01048738
print(colours[1]):                      red
print(colours[2]):                      green
print(colours[3]):                      blue
print(table.concat(colours, ',')):      red,green,blue
Press Enter to quit_
```

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

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section5/02-TableInsert.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
input("Press Enter to quit")
```


C:\Program Files (x86)\Lua\lua53.exe

```
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
Press Enter to quit_
```

A new empty table is created

The user is asked for a favourite colour

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

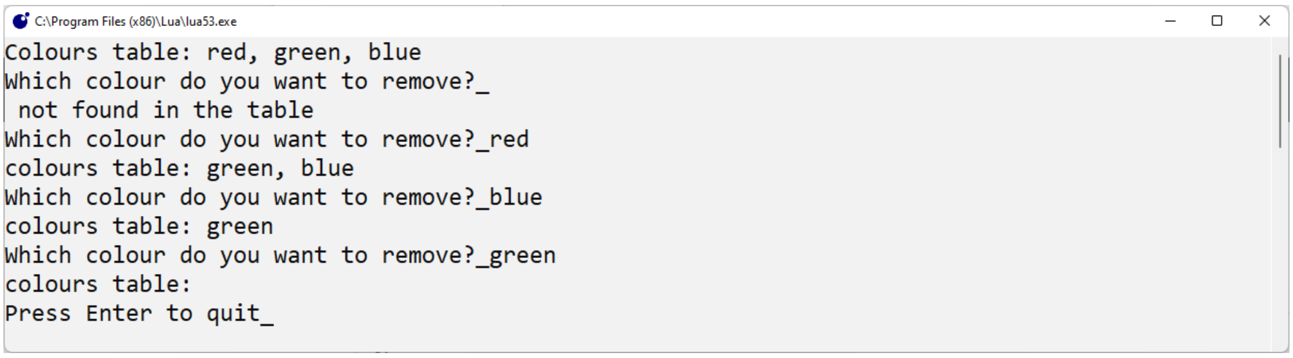
Note: there is no check to see if the same colour is inserted more than once.

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

```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
input("Press Enter to quit")
```



```
Colours table: red, green, blue
Which colour do you want to remove?_
 not found in the table
Which colour do you want to remove?_red
colours table: green, blue
Which colour do you want to remove?_blue
colours table: green
Which colour do you want to remove?_green
colours table:
Press Enter to quit_
```

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.

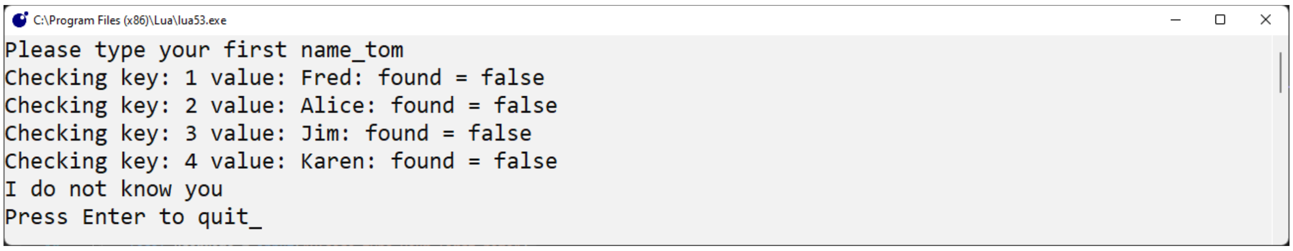
The while loop ends when the table is empty: `#colours > 0 is false`

Lua Tables and for loops

04-TableIteration.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section5/04-TableIteration.lua

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

function main()
    local found = false
    local myFriends = {"Fred", "Alice", "Jim", "Karen"}
    local userName = input("Please type your first name")
    for key, value in ipairs(myFriends) do
        print("Checking key: "..key.." value: "..value..": found = "..tostring(found))
        if userName:lower() == myFriends[key]:lower() then
            found = true
        end
    end
    if found then
        print("Hello friend")
    else
        print("I do not know you")
    end
    input("Press Enter to quit")
end

main()
```



C:\Program Files (x86)\Lua\lua53.exe

```
Please type your first name_tom
Checking key: 1 value: Fred: found = false
Checking key: 2 value: Alice: found = false
Checking key: 3 value: Jim: found = false
Checking key: 4 value: Karen: found = false
I do not know you
Press Enter to quit_
```

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:

```
print("Checking key: "..key.." value: "..value..": found = "..tostring(found))
```

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

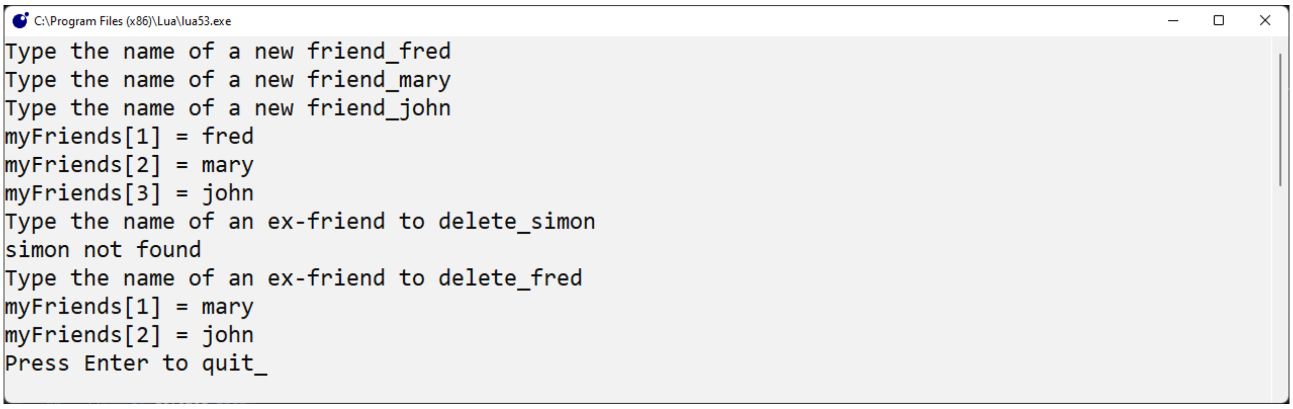
```
if userName:lower() == myFriends[key]:lower() then
    found = true
end
```

05-AssignmentTables.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section5/05-AssignmentTables.lua

```lua
 --[[
    tables assignment:
    Complete the skeleton code below to do the following:
    1. Add 3 names to the empty table myFriends
    2. Ask the user to select a name to be deleted from the table
    ]]

local myFriends = {}    -- table created here so is available to all functions

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

local function addToTable()
    --[[ Add a new friend to the table myFriends ]]
    local newName = input("Type the name of a new friend")
    local added = true     -- assume new name will be added to the table
    for key,value in ipairs(myFriends) do
        -- check if newName is already in the table
        -- set value of added as appropriate
    end
    if added then
        -- insert newName into table
    else
        -- tell user that name already exists
    end
    return added    -- true/false
end

local function displayTable()
    --[[ Display the contents of the table myFriends ]]
    for key,value in ipairs(myFriends) do
        -- display each key/value pair
    end
end

local function deleteFromTable()
    --[[ Delete a friend from the table myFriends ]]
    local oldName = input("Type the name of an ex-friend to delete")
    local deleted = false
    local index = 0
    for key,value in ipairs(myFriends) do
        -- check if oldName is in the table
        -- set deleted and index as appropriate
    end
    if deleted then
        -- remove the value at index
    else
    -- tell user that name not found
    end
    return deleted    -- true/false
end

local function main()
    --[[ Add 3 names to empty table and delete 1 ]]
    local count = 0
    -- do not use a for loop. Why not?
    repeat
        if addToTable() then -- true = name added, false = entered name exists
            count = count + 1
        end
    until count == 3
    displayTable()
    -- remove 1 name
    while not deleteFromTable() do end -- return value of function deleteFromTable() is true/false
    displayTable()
    input("Press Enter to quit")
end

main()
```

Using the above skeleton code, complete the functions to produce the following output:



```
Type the name of a new friend_fred
Type the name of a new friend_mary
Type the name of a new friend_john
myFriends[1] = fred
myFriends[2] = mary
myFriends[3] = john
Type the name of an ex-friend to delete_simon
simon not found
Type the name of an ex-friend to delete_fred
myFriends[1] = mary
myFriends[2] = john
Press Enter to quit_
```

06-TableRectangle.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section5/06-TableRectangle.lua

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

The table represents a rectangle and contains it's X and Y coordinates, Width and Height.

It also contains a nested table of floats representing the rgb colour values and a string containing its draw mode (line or fill)

```lua
local rect = {}                -- create an empty table
rect.X = 0                     -- set X value to 0
rect.Y = 0                     -- set Y value to 0
rect.Width = 10                -- set Width to 10
rect.Height = 4                -- set Height to 4
rect.Colour = {0.2, 0.5, 0.9} -- set colour to relative values of red / green / blue 0 to 1
rect.Mode = "fill"             -- set draw mode to "fill" (alternative is "line")
```

This could also have been created in a more compact, direct form:

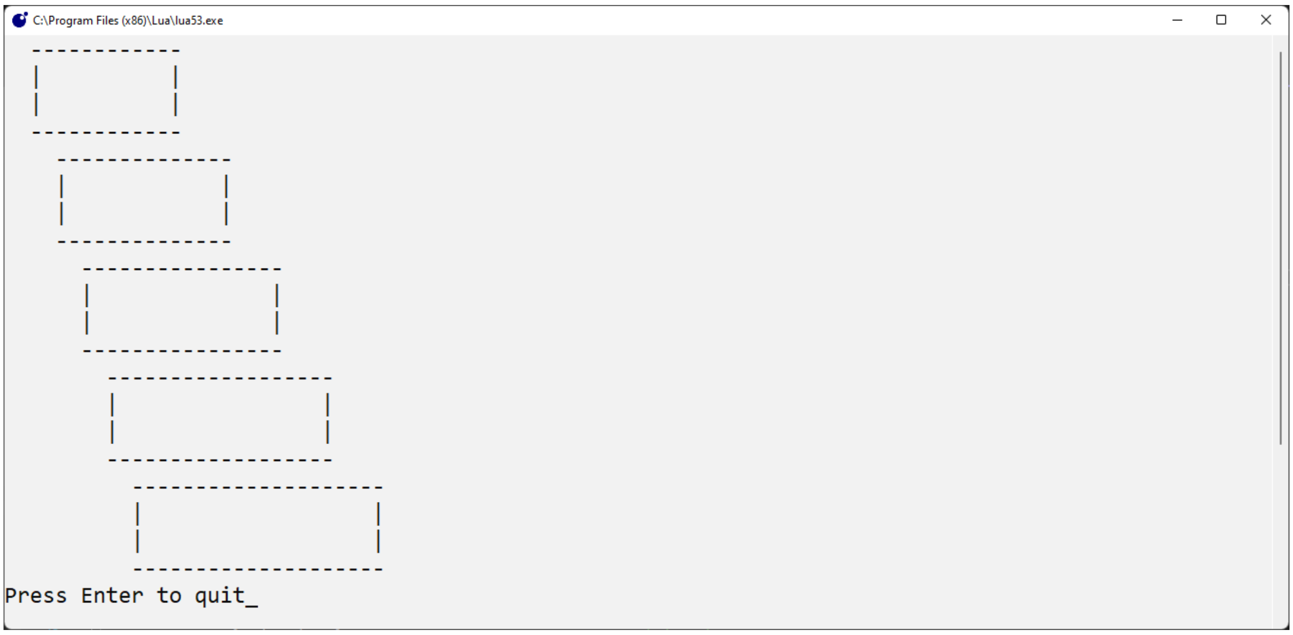
```lua
local rect =
{
    X = 0,
    Y = 0,
    Width = 10,
    Height = 4,
    Colour = {0.2, 0.5, 0.9},
    Mode = "fill"
}
```

This table can be referred to directly to assign and read values to/from the various rectangle properties:

```lua
--[[ increase the X position and Width ]]
rect.X = rect.X + 2                     -- add 2 to X position
rect.Width = rect.Width + 2             -- add 2 to Width


local width = ("-"):rep(rect.Width)      -- string of - equivalent to Width "----------"
```

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

In this file they are called just 5 times, and the resulting rectangle printed out:



```
  ------------
  |          |
  |          |
  ------------
    --------------
    |            |
    |            |
    --------------
      ----------------
      |              |
      |              |
      ----------------
        ------------------
        |                |
        |                |
        ------------------
          --------------------
          |                  |
          |                  |
          --------------------
Press Enter to quit_
```

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

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

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

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

local function draw()
    local x = (" "):rep(rect.X)             -- string of spaces equivalent to position of X "" -> "     "
    local width = ("-"):rep(rect.Width)     -- string of - equivalent to Width "----------" -> "---------------"
    local empty = (" "):rep(rect.Width - 2) -- string of spaces equivalent to inside of rectangle 8 -> 13
    print(x..width)                         -- print top of rectangle "----------" -> "---------------"
    for rows = 1, rect.Height - 2 do        -- loop: height of rectangle minus top and bottom
        print(x.."|"..empty.."|")           -- print sides of rectangle "|        ||" -> "||             |"
    end
    print(x..width)                         -- print bottom of rectangle "----------" -> "---------------"
end

local function main()
    rect.X = 0                    -- set X value to 0
    rect.Y = 0                    -- set Y value to 0
    rect.Width = 10               -- set Width to 10
    rect.Height = 4               -- set Height to 4
    rect.Colour = {0.2, 0.5, 0.9} -- set colour to relative values of red / green / blue 0 to 1
    rect.Mode = "fill"            -- set draw mode to "fill" (alternative is "line")
    for frames = 1, 5 do
        update()                  -- update values of rect.X and rect.Width
        draw()                    -- draw updated rectangle
    end
    input("Press Enter to quit")
end

main()
```

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

Functions within Tables

07-TableWithFunctions.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section5/07-TableWithFunctions.lua

Following on from the rectangle table used above, there is one more step you can take.

Tables can hold entire functions within them….

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

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

local function update()
    --[[ increase the X position and Width ]]
    --rect.X = rect.X + 2                    -- add 2 to X position
    --rect.Width = rect.Width + 2            -- add 2 to Width
    rect.moveHorizontal(2)
    rect.changeWidth(2)
end

function rect.draw()
    local x = (" "):rep(rect.X)             -- string of spaces equivalent to position of X ""
    local width = ("-"):rep(rect.Width)     -- string of - equivalent to Width "----------"
    local empty = (" "):rep(rect.Width - 2) -- string of spaces equivalent to inside of rectangle
    print(x..width)                         -- print top of rectangle "----------"
    for rows = 1, rect.Height - 2 do        -- loop: height of rectangle minus top and bottom
        print(x.."|"..empty.."|")           -- print sides of rectangle "|        ||"
    end
    print(x..width)                         -- print bottom of rectangle "----------"
end

local function main()
    rect.X = 0                    -- set X value to 0
    rect.Y = 0                    -- set Y value to 0
    rect.Width = 10               -- set Width to 10
    rect.Height = 4               -- set Height to 4
    rect.Colour = {0.2, 0.5, 0.9} -- set colour to relative values of red / green / blue 0 to 1
    rect.Mode = "fill"            -- set draw mode to "fill" (alternative is "line")
    function rect.moveHorizontal(pixels)
        rect.X = rect.X + pixels  -- add no of pixels to X position
    end
    function rect.changeWidth(pixels)
        rect.Width = rect.Width + pixels -- add no of pixels to Width
    end
    for frames = 1, 5 do
        update()                  -- update values of rect.X and rect.Width
        rect.draw()               -- draw updated rectangle
    end
    input("Press Enter to quit")
end

main()
```

The first part of function main() is unchanged, then you get:

```lua
    function rect.moveHorizontal(pixels)
        rect.X = rect.X + pixels  -- add no of pixels to X position
    end
    function rect.changeWidth(pixels)
        rect.Width = rect.Width + pixels -- add no of pixels to Width
    end
```

These 2 start with `function rect.` Which makes them part of the rect table.

They do exactly the same job as the original lines in the update() function:

```
    rect.X = rect.X + 2                     -- add 2 to X position
    rect.Width = rect.Width + 2             -- add 2 to Width
```

now changed to:

```
    rect.moveHorizontal(2)
    rect.changeWidth(2)
```

Things have been taken even further.
The original:

```
   local function draw()
```

has been changed to:

```
    function rect.draw()
```

The contents of the function have not been changed.

As `draw()` no longer exists, one othe change is in `main()` again:

from:

```
for frames = 1, 5 do
     update()                    -- update values of rect.X and rect.Width
     draw()                      -- draw updated rectangle
end
```

to:

```
for frames = 1, 5 do
    update()                    -- update values of rect.X and rect.Width
    rect.draw()                 -- draw updated rectangle
end
```

Those of you who have already started using Object Oriented Programming (OOP) will see a pattern here.

This 'rect' table is acting like a class. It has properties AND now methods.

This is indeed the equivalent of a C# *static* class. (you cannot make rectangle objects, there is only one.)

It can become a real class, where rectangle objects can be made from it.

This only takes a few extra lines involving metatables, but that is outside the scope of this tutorial.

Variable Scope

08-VariableScope.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section5/07-VariableScope.lua
This is a subject usually tackled at a much later date when learning Python, C# or Java, but it is important, so best introduced at an early stage.

```lua
--[[this file will cause an error
    07-VariableScope.lua:21: attempt to concatenate global 'myString' (a nil value)
]]
local function input(prompt)
    --[[ get input from user ]]
    io.write(prompt .. "_")
    return io.read()
end

local function main()
    local myString = "Hello"

    print("The variable myString contains: "..myString)
    print()

    myString = myString.." Goodbye"
    print("The variable ‘myString’ contains: "..myString)
end

main()
print("The variable ‘myString’ contains: ".. myString) -- error!
input("Enter to quit")
io.read()
```

If you read the code through, the line after main()

```lua
print("The variable ‘myString’ contains: ".. myString)  -- error!
```

should print out the contents of the variable again, just like it did on the exact same line inside the `main()` function.

Run it. Whoops!:

```
Program starting as '"C:\Users\Public\PortableApps\ZeroBraneStudio\App\bin\lua.exe" -e
"io.stdout:setvbuf('no')" "C:\Users\graha\Dropbox\Computer Club\Lua\Beginners\Section4\07-
VariableScope.lua"'.
Program 'lua.exe' started in 'C:\Users\graha\Dropbox\Computer Club\Lua\Beginners\Section4' (pid:
21292).
The variable myString contains: Hello

The variable myString contains: Hello Goodbye
C:\Users\Public\PortableApps\ZeroBraneStudio\App\bin\lua.exe: ...omputer Club\Lua\Beginners\
Section4\07-VariableScope.lua:21: attempt to concatenate global 'myString' (a nil value)
stack traceback:
        ...omputer Club\Lua\Beginners\Section4\07-VariableScope.lua:21: in main chunk
        [C]: at 0x00401b00
Program completed in 0.03 seconds (pid: 21292).
```

So what went wrong?

The script ran ok until the line after main()

Delete the word ‘`local`’ from `local myString = "Hello"` and try again. It all works perfectly.

Adding the word 'local' makes the variable local to the code block it is declared in.

When the main() function was finished, the variable 'myString' was **deleted**, as it was **local** only to main()
When the last line executed, myString did not exist, so it's value is nil, and you cannot join (concatenate) a string with nil, (or add a number to nil), so the error in red above makes sense.

Removing local declared the variable 'global' by default, so all functions can use it.

Good coding practice is to use variables carefully, and keep them as local as possible.
The main reason for this in a large project is you may inadvertently change the value of a global variable somewhere you did not expect, and nothing works predictably as a result.

So how can I use a local variable in main() and use it later? 2 methods shown in the next 2 files:

09-VariableScope2.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section5/08-VariableScope2.lua

```lua
-- using a module scope variable
local myString -- declared here but not given a value

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

local function main()
    myString = "Hello" -- module scope variable given a value

    print("The variable myString contains: "..myString)
    print()

    myString = myString.." Goodbye" -- module scope variable given a modified value
    print("The variable 'myString' contains: "..myString)
end

main()
print("The variable 'myString' contains: ".. myString) -- no error as using module scope variable.
input("Enter to quit"
```

- This is the method commonly used, especially by beginners.
- The variable myString is declared directly in the module (not inside a function). `local myString`
- You can give it a value at declaration if needed.
- Although it is preceeded by the local keyword, that indicates it is local to the whole file, so is good practice to use local most of the time.
- The value is read/write inside any function.
  ```lua
  local function main()
      myString = "Hello"
  ```
-

Another method is to pass any local variables out of a procedure such as main() by converting the procedure into a function. A function is a procedure that returns one of more values:

10-VariableScope3.lua

https://github.com/Inksaver/LuaForSchools/blob/main/Beginners/Section5/09-VariableScope3.lua

```lua
-- creating a local variable inside main() and passing it out
local function input(prompt)
    --[[ get input from user ]]
    io.write(prompt .. "_")
    return io.read()
end

local function main()
    local myString = "Hello" -- this variable is local to main() only

    print("The variable myString contains: "..myString)
    print()

    myString = myString.." Goodbye"
    print("The variable 'myString' contains: "..myString)
    return myString --note passing the value of myString out
end

local myString = main() -- this variable is different from the one created inside main()
print("The variable 'myString' contains: ".. myString)
input("Enter to quit")
```

You do not really need to use local here: `local myString = main()`, as it is declared in the script body.
It is still useable by all the functions in the script, but it does make it invisible to any other files that make use of this script.
Later you will be making multi-file programs, so it is good to get into the habit.

This method of passing data explicitly from one function to another is good coding practice, and is extensively used in most languages.

The scope of a variable declared anywhere in the script without the keyword ‘local’ has **global scope** by default. It can be accessed from any function in the script **and by other scripts** within the project.

The scope of a variable declared ‘local’ in the body of the script has **module level scope**: It can be used by all functions in the script, but cannot be accessed by other scripts in the project.

The scope of a variable declared ‘local’ in a code block (function, loop etc) has **local scope**. It can only be accessed within that code block.

Important: Once the variable has been ‘declared’ (used the first time) Do NOT use the local keyword in any subsequent uses, as this will create a new variable of the same name and could over-write the existing value.

Note for Python users:

Python has got everything the wrong way round:

Python has no equivalent of ‘local’ or ‘private’: it has a ‘global’ keyword instead

Python variables are local by default

Python variables declared in the body of the script SHOULD have **module level scope** and be available to functions within the script.

They don’t. Read only. If you want to modify them, you have to re-declare the variables inside functions using the global keyword.

This is confusing and illogical. That is Python...