GAME.DEV

Lua – part 2

BITS Pilani

Pilani Campus

Tables

- Associative arrays (key => value)
- only compound data structure
- Empty table declaration:

  T={}

- Most general format of Tables – literal notaion

T= { [“my_name”] = “text1” , [6.78] = “text3” , [“text4”] = 3.14 }

- String notation: -- number keys can’t use this notation

T ={ my_name = “text1” , text4 = 3.14 }

Accessing/Modifying Table entries

- Literal notation

  print(T[“my_name”])

  print(T[6.78])

  T[6.78] = 34

  T[454] = 34.5

- String notation

  print(T.my_name)

  T.text4 = 78

Using Tables as Arrays

- List literals implicitly set up integer keys starting form 1:

```
v = {'value1', 'value2', 1.21, 'gigawatts'}
print(v[1]) --  value1
print(v[3]) --  1.21
```

- Array length given by #operator

```
for i = 1, #v do -- #v is the size of v
 print(v[i])
end
```

Initialize Arrays

```
a = {} -- new array
for i=1, 1000 do
a[i] = 0
end
```

print(#a) --> 1000

Be careful with the Length Operator!

- Try length on this :

```
a = {}
a[10000] = 1
print(#a)
```

- Better option

```
a = {}
a[10000] = 1
print(table.maxn(a)) --> 10000
```

Iterating through Tables:

- print all values of table 't‘

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

- Iterate through the following table:

T = { “23” ,23 ,{ 45,”deep” } ,89,”text” }

Iterate through the following table:

innovate achieve lead

T = { “23” ,23 ,{ 45,”deep”,”lets go” , { “further deeper” , {“deepest”} } } ,89,”text” }

Numbers re-visited

1| First a function called ‘generateTable()’ which will create a table of all Narcissistic numbers up to 1000000.

- Create a global empty table T
- Fill this table using the generateTable() function.

2| Now a simple program, which after generating such a table will , ask the user for an Armstrong function, and will simply look up this table for answers. Write it as a function *checkTable(n)*

Functions as first-class members

- Ex1

```
function foo(a,b,c)
        print("foo does something")
end
```

- Ex2 -- same as Ex1

```
foo = function (a,b,c)
        print("foo does something")
end
```

Functions as first-class members (cont'd)

```
foo2 = foo  -- will also work

function foo3()
return "foo3"
end

T = { foo() , foo2() ,foo3() , foo}
--[[
    the above statement will result in a table equivalent to
    T={nil,nil,"foo3" , function_reference }
    and will print "foo does something" to the screen twice when file is
    executed
    try printing the table T (via iterating it )
--]]
```

Tables with functions

- the example as shown before
  T= { foo,foo2,foo3 }

- Build dynamically: (anonymous functions – implicity)

```
T = {
        fun1 = function (x) return x^2 end ,
        [3.14] = function () print("Approx Value of PI")
    }
```

- Build dynamically: (named functions – explicitly)

```
T={}

function T.fun1(x)
  return x^2
end

T[3.14] = function ()
  print("Approx Value of PI")
end
```

Modules

- Defining a Module:
  - Declare a local Table
  - Define functions, variables as part of this local table
    - Make sure all variables, etc defined in these functions are all local
  - Return the Table
  - Save the file as *module_name.*lua
- Loading a module:
  - Variable = require “*module_name*”

Modular Programming

- Write a module containing all the functions and variables you have defined till now, namely:

1. Empty table T
2. countDigits(n)
3. isArmstrong(n)
4. isNarcissistic(n) – manual version
5. generateTable()
6. checkTable(n)

IO Model

- By default io.read() and io.write() reads and writes to the standard input/output. (stdout/stdin)
- To change, set the filename using:
  - io.input(*filename*)
  - io.output(*filename*)

- io.read() already done before
- io.write(*args*) gives more control than print()
  - *args* are any number of comma delimited arguments which are outputted to the output file, one after the other
  - newlines, spaces all have to be explicity specified in io.write()

- Example on next slide…

io.write() example

- Note the differing output from print()

```
> print("hello", "Lua"); print("Hi")
--> hello Lua
--> Hi

> io.write("hello", "Lua"); io.write("Hi", "\n")
--> helloLuaHi
```