Lua Types Tutorial

This is a basic introduction to the variable types used by Lua when scripting. Each section introduces a different variable type. Please look at TutorialExamples for notes about the examples.

Numbers

Lua allows simple arithmetic using the usual operators to add, subtract, multiple and divide. We'll use the `print()` function to print out the results of some calculations. The brackets around the arguments are important and will cause an error if missed out.

```
> print(2+2)
4
> print(2-7)
-5
> print(7*8)
56
> print(7/8)
0.875
```

Notice that the numbers are not rounded into integers. They are floating point, or real numbers. We can assign values to variables using the `=` operator.

```
> x = 7
> print(x)
7
```

The variable `x` is created when the number 7 is assigned to it. We use the `print()` function again to print out the value of `x`. We can now use the value in `x` for other calculations.

```
> x = x * 9
> print(x)
63
> print(x*2) -- will not change the value of x
126
> print(x)
63
```

Notice how `print(x*2)` does not change the value of `x` because it was not assigned using the `=` but `x = x * 9` multiples the current value of `x` by 9 and stores the new value in `x` again.

For more information on Lua's number type you can look at the NumbersTutorial.

Strings

Lua also uses strings, or text variable types:

```
> print("hello")
hello
```

We can assign strings to variables just like we can numbers:

```
> who = "Lua user"
> print(who)
Lua user
```

We can concatenate (join together) strings together using the `..` operator between two strings.

```
> print("hello ")
hello
> print("hello " .. who) -- the variable "who" was assigned above
hello Lua user
> print(who)
Lua user
```

Notice that the `..` operator does not change the value of `who` unless the `=` assignment operator is used, just like numbers.

```
> message = "hello " .. who
> print(message)
hello Lua user
```

Unlike some other languages, you can not use the `+` operator to concatenate strings. i.e.:

```
> message = "hello " + who
stdin:1: attempt to perform arithmetic on a string value
stack traceback:
        stdin:1: in main chunk
        [C]: ?
```

Tables

Lua also has a general purpose data type called a table. Tables can be used to store groups of objects. You can store numbers, or strings, or other tables in tables. Tables are created using a pair of curly brackets `{}` . Let's create an empty table:

```
> x = {}
> print(x)
table: 0035C910
```

When we display the value of a table variable using the built in `print` function Lua just displays the fact that variable is a table, and unique identifier for that table (i.e. its address in memory). We can print out the contents of a table but that comes in the TablesTutorial.

We can construct tables containing other objects, such as the numbers and strings described above, e.g.

```
> x = { value = 123, text = "hello" }
> print(x.value)
123
> print(x.text)
hello
```

We can print the values out using the notation: *table.item*. We can also put tables inside other tables.

```
> y = { const={ name="Pi", value=3.1415927 }, const2={ name="light speed", value=3e8 } }
> print(y.const.name)
Pi
> print(y.const2.value)
300000000
```

Boolean

Boolean values have either the value `true` or `false`. If a value is not true, it must be false and vice versa. The `not` operator can be placed before a boolean value to invert it. i.e. `not true` is equal to `false`.

```
> x = true
> print(x)
true
> print(not x)
false
> print(not false)
```

```
true
```

Boolean values are used to represent the results of logic tests. The equals `==`, and not equals `~=` operators will return boolean values depending on the values supplied to them.

```
> print(1 == 0) -- test whether two numbers are equal
false
> print(1 == 1)
true
> print(1 ~= 0) -- test whether two numbers are not equal
true
> print(true ~= false) -- is true not equal to false?
true
```

Functions

In Lua, functions are assigned to variables, just like numbers and strings. Functions are created using the `function` keyword. Here we create a simple function which will print a friendly message.

```
> function foo() print("hello") end -- declare the function
> foo() -- call the function
hello
> print(foo) -- get the value of the variable "foo"
function: 0035D6E8
```

Notice we can print the value of the variable `foo` and it displays (like tables) that the variable is a function, and has unique identifier for that particular function. So, being a variable just like any other, we should be able to assign functions to variables, just like the other variables, and we can.

```
> x = function() print("hello") end
> x()
hello
> print(x)
function: 0035EA20
```

The ability to do this is because Lua has *first class values*. This means that all values are treated the same way. This is a very powerful and useful feature of Lua.

Dynamic typing

You might have noticed that whilst we created the above variables, we did not have to specify which type of variable we were creating. For example,

```
a = 1
b = "hello"
c = { item1="abc" }
```

In other languages, such as C, we have to specify the type of a variable when we create it. As well as not having to specify which type a particular variable is we can also assign different value types to the same variable, e.g.

```
a = 1
a = "hello"
a = { item1="abc" }
```

This is called *dynamic typing*. This means that you don't have to specify what type a variable is. The variable knows what type it is from the value, or object, assigned to it.

nil values

This is a special value which means a variable has no value. If a variable has the value `nil` then it has no value assigned to it and therefore will no longer exist (or doesn't exist yet). By setting a variable to `nil` you can delete a variable. e.g.

```
> x = 2.5
> print(x)
2.5
> x = nil
> print(x)
nil
```

You can test to see if a variable exists by checking whether its value is `nil`.

```
> print(x == nil)
true
> x = 7
> print(x == nil)
false
> print(x)
7
```