Tables Tutorial

You can read about tables in sections 2.2 and 2.5.6 of the Reference Manual [1]. Tables are a very flexible and useful data type in Lua. This page is an introduction to various features.

Lua has a *table library* to complement its table data type. Examples on this page use functions from the standard table library. You can find more information about the Lua table library in section 5.4 of the Reference Manual [1]. There is more information in the TableLibraryTutorial.

Creating tables

Tables are created using *table constructors*, which are defined using curly brackets, e.g. `{ }` . To define an empty table we can do the following.

```
> t = {}    -- construct an empty table and assign it to variable "t"
> print(t)
table: 0035AE18
```

Notice when the value of a table is displayed only the type and unique id of the object are displayed. In order to print out the contents of a table we must do so explicitly. We'll learn how to do this later.

Tables as arrays

Tables can be used to hold arrays of information. Table constructors can contain a comma separated list of objects to create an array. The array elements can be accessed using square brackets, *table[index]*. e.g.:

```
> t = { 1,1,2,3,5,8,13 }
> print( t[1] )
1
> print( t[0] )
nil
> print( t[4] )
3
```

Notice that the indexing into the array starts at 1, not at zero. `t[0]` has the value `nil`, i.e. there is no element at position 0. The `t = { 1,1,2,3,5,8,13 }` line is equivalent to the following:

```
> t = {}
> t[1]=1 t[2]=1 t[3]=2 t[4]=3 t[5]=5 t[6]=8 t[7]=13
```

Using table constructors is a little more readable and less error prone though.

We can find the size of a table using the standard table library function `table.getn()` (i.e. get number of elements)

```
> = table.getn(t)
7
```

We can append new elements to the table array using the `table.insert(table,value)` function.

```
> table.insert(t,21)
> = table.getn(t)
8
> = t[7], t[8]
13      21
```

We can also insert elements at a chosen point in the list. We can use the table library to insert elements into the table array without having to shuffle the other elements around. To do this we supply 3 arguments to the `table.insert(table,position,value)` function. We can also use the `table.remove (table,position)` to remove elements from a table array.

```
> table.insert(t,4,99)
> = t[3], t[4], t[5]
2       99      3
> table.remove(t,4)
> = t[3], t[4], t[5]
2       3       5
```

We can show the elements contained in an array using the `table.foreachi(table,function)` function. This applies the function passed to each element of the array, in index order. For example, if we pass the above table and the `print()` function we get the following output.

```
> table.foreachi(t,print)
1       1
2       1
3       2
4       3
5       5
6       8
7       13
8       21
```

Note that we are not restricted to storing one type of data in an array. We can insert numbers, strings, functions, or other tables, e.g.

```
> t[4] = "three"
> t[6] = "eight"
> t[2] = { "apple", "pear", "banana" }
> table.foreachi(t,print)
1       1
2       table: 0035DFE8
3       2
4       three
5       5
6       eight
7       13
8       21
```

Tables as dictionaries

Tables can also be used to store information which is not indexed numerically, or sequentially, as with arrays. These storage types are sometimes called *dictionaries*, *associative arrays* or *mapping types*. We'll use the term *dictionary* where an element *pair* has a *key* and a *value*. The key is used to set and retrieve a value associated with it. Note that, just like arrays we can use the *table[key] = value* format to insert elements into the table. A key need not be a number, it can be a string, or for that matter, any other Lua object. Let's construct a table with some key-value pairs in it:

```
> t = { apple="green", orange="orange", banana="yellow" }
> table.foreachi(t,print)
> table.foreach(t,print)
apple   green
orange  orange
banana  yellow
```

Notice that we have to use the `table.foreach(table,function)` function, rather than the `table.foreachi(table,function)` to output the values. This is because the keys are no longer numbers and the `table.foreachi()` function iterates over the *indices* in a table, whereas the `table.foreach()` function iterates over the *keys* in a table. Note, there is no guarantee as to the order in which keys will be stored in a table when using dictionaries so the order of retrieval of keys using `table.foreach()` is not guaranteed. e.g.

```
> t.melon = "green"
> t["strawberry"] = "red"
> table.foreach(t,print)
melon   green
strawberry      red
apple   green
orange  orange
```

```
banana  yellow
```

The *table.key = value* format is shorthand for *table["key"] = value* when the key is a string, i.e. `t.apple` is a little more readable than `t["apple"]`. This *syntactic sugar* makes Lua more readable. In the above example:

```
> t = { apple="green", orange="orange", banana="yellow" }
```

is the same as:

```
> t = { ["apple"]="green", ["orange"]="orange", ["banana"]="yellow" }
```

Note, where our key contains a space we have to use the *["key"]=value* format:

```
> t = { ["keys can contain more than one word"] = "as a string can contain any characters" }
> t["another string"] = 99
```

Mixed table constructors

You are not restricted to using table constructors as just sequentially indexed lists, or as dictionaries, you can mix the two together, e.g.,

```
> t = { 2,4,6, language="Lua", version="5" }
```

Here we have an array of numbers followed by some dictionary values. We can use our Lua table library functions to output the contents of the table.

```
> table.foreach(t,print)
1       2
2       4
3       6
language        Lua
version 5
> table.foreachi(t,print)
1       2
2       4
3       6
```

Notice how the outputs differ. `table.foreachi()` has only output the numerically indexed content and `table.foreach()` has output all of the content. We can switch backwards and forwards between the two construction styles at will:

```
> t = { 2,4,6, language="Lua", version="5", 8,10,12, web="www.lua.org" }
```

Notes about table keys

Lua stores all elements in tables generically as *key-value* pairs. Lua does not differentiate between arrays and dictionaries. All Lua tables are actually dictionaries. In an array the keys are just number object keys.

```
> t = { 3,6,9 }                 -- is the same as...
> t = { [1]=3, [2]=6, [3]=9 }   -- is the same as...
> t = {} t[1]=3 t[2]=6 t[3]=9
```

Keys are references

Note that keys are *references to objects* so you must use the same reference to get the same key into the table. You can use any Lua object as a key in a table. We can demonstrate that keys are references to objects by using a table as a key:

```
> a = { [{1,2,3}]="yadda" }  -- construct a table where the element has a table key
> table.foreach(a,print)     -- display the table contents: key-value pairs
table: 0035BBC8 yadda
> = a[{1,2,3}]               -- display the element value
nil
```

This didn't work because when we tried to retrieve the value of `a[{1,2,3}]` we constructed another table, i.e. the table that we used as a key in the table constructor is not the same one as we used to retrieve the value. If we use the same table in both cases then it will work:

```
> tablekey = {1,2,3}           -- create a table with a variable referencing it
> a = { [tablekey]="yadda" }   -- construct a table using the table as a key
> table.foreach(a,print)       -- display the table elements
table: 0035F2F0 yadda
> = a[tablekey]                -- retrieve a value from the table
yadda
> print(tablekey)              -- the table value is the same as the key above
table: 0035F2F0
```

Strings as references

Aside: The fact that keys are references, and strings can be used, tells us something about Lua's internals. We must be referring to the same string when we get and set the value or table string keys wouldn't work! Strings are not duplicated in Lua internally, so when you reference a string with the same value you are refering to the same string.

```
> t = { apple=5 }
```

```
> a = "apple"
> b = "apple"
> print(t.apple, t[a], t[b])  -- all the same value because there is only one "apple"!
5       5       5
```

Anything as a key

Just to demonstrate that we can use all Lua objects here is a function as a key:

```
> t = { [function(x) print(x) end] = "foo" }
> for key,value in t do key(value) end
foo
```

The anonymous function takes a single argument, which it prints out, and has the value "foo". We iterate over the table executing the key (i.e. function) and passing it the value of the function, which it print out: "foo".

FindPage · RecentChanges · preferences
edit · history
Last edited January 14, 2004 5:29 pm PDT (diff)