An Introduction to Lua

This introduction assumes that you have some familiarity with basic programming.

We recommend that you get a good programmer's editor which understands Lua syntax and can highlight it appropriately. This is not just because it's more attractive, but such an editor can immediately show you common mistakes like forgetting the closing quote on a string. Such editors can also match up parentheses and braces. They will often automatically *indent* code for you, which is important for readability..

It is better to lay out code like this:

```
for i = 1,10 do
    if i < 5 then
        print("lower",i)
    else
        print("upper",i)
    end
end
```

Than this:

```
for i = 1,10 do
if i < 5 then
print("lower",i) else
print("upper",i)
end
end
```

The logical structure of the code is now harder to see. By making indenting a habit, you will write code that will be more readable to another person. (After several months, you will be that other person.)

A good start is the Lua wiki. If you are on Windows, then the Lua for WIndows distribution comes with SciTE, which also has Lua debugging support.

Basics

Expressions

It is traditional to start with "Hello, world". Like all scripting languages, this is straightforward:

```
print("Hello world")
```

An important point is that `print` is a function which takes a number of values and prints out these values to standard output. It is not a statement, as in Python 2.X or Basic.

To run this example, save this line as 'hello.lua' and run from the command line:

```
$> lua hello.lua
Hello world!
```

(Or, use an editor that knows how to run Lua, like SciTE in Lua for Windows. The F5 key will run the current program.)

Arithmetic expressions use standard programming notation:

```
print("result",2 + 3 * 4 ^ 2)
```

There is an exponentiation ('power of') operator `^`, unlike in C or Java.

The order of evaluation follows the usual rules, and you can use parentheses to group expressions - for example, `(1 + 2)*(3 + 4)`. The original expression could be written like:

```
(2 + (3 * (4 ^ 2))
```

which makes the order obvious. When in doubt, use parentheses, but knowing when you have to use them is an important part of learning a programming language.

Operations like `+` and `*` are called *operators*, and the values they operate on are called *arguments*.

There is a remainder operator, `%` which gives the integer remainder from a division:

```
print(1 % 2) --> 1
print(2 % 2)  --> 0
print(3 % 2) --> 1
```

There are the usual standard mathematical functions available in the `math` namespace (or *table*) like `math.sin`, `math.sqrt`,etc. The useful constant `math.pi` is also available.

```
print('sin', math.sin(0.5*math.pi), 'cos', math.cos(0.5*math.pi))
```

This is more readable when spread over two lines using a *variable* x:

```
x = 0.5*math.pi
print('sin',math.sin(x),'cos',math.sin(x))
```

Of course, you could say that x is not a variable, but Lua does not make any distinction between variables and named constants.

Even if the command-line is not your strong point, I recommend learning Lua interactively.

```
$> lua
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
> print(10 + 20)
30
> = 10 + 20
30
> = math.sin(2.3)
0.74570521217672
> p = math.pi
> = math.cos(p)
-1
```

Starting a line with = is a shortcut for print(). Interactive Lua is a very useful scientific calculator !

With Lua for Windows, it is not even necessary to open a command prompt. On the SciTE toolbar, there is a prompt icon with the tooltip 'Launch Interactive Lua'. It will open an interactive session in the output pane, and you can evaluate Lua statements. As with a console prompt, the up and down arrows can be used to select and re-evaluate previous statements.

Variables and Assignment

Lua variables may have letters, digits and underscores ('_'), but they cannot start with a digit. So check_number, checkNumber, and catch21 are fine, but 21catch is not. There is no limit to the length of variable names, except the patience of you and your readers.

Variables are *case-sensitive*. ERROR,Error and error are all different variables.

They may contain any Lua type:

```
x = "hello"
x = 1
x = x + 1
```

Here, the value of `x` becomes "hello", and then it becomes 1, and finally 2 (which is the value of `x + 1`). This process is commonly called *assignment*.

Variables have no fixed type, but always contain a definite type of value, even if it is just `nil`.

Assigning multiple variables works like this:

```
x, y = 1, 2
```

This sets `x` to 1 and `y` to 2. This is different from the usual `x=1, y=2` style in other languages. It is possible to swap the values of two variables in one line:

```
x, y = y, x
```

The *assignment* statement `x = 1` is not an expression - it does not return a value.

Numeric `for` loops

Programming is more than calculation. The simplest way to do something a number of times is the *numeric for* statement:

```
for i = 1, 5 do
    print("hello",i)
end
-->
hello   1
hello   2
hello   3
hello   4
hello   5
```

Any statements in the *block* from `do` to `end` are repeated with different values of the variable `i`.

Spacing and line ends (often called 'whitespace') is usually not important. This code could also be written like so:

```
for i = 1, 5 do print("hello", i) end
```

or

```
for i = 1, 5
do
print("hello",i)
end
```

However, the last example would be considered bad style. The logic of a program is much easier to follow if statements are laid out at the right columns.

The numeric `for` statement loops from a start to a final value, inclusive. There can be a third number, which is the 'step' used in calculating the next value. To print the greetings out backwards:

```
for i = 5, 1, -1 do print("hello",  i) end
```

So to print a little table of sine values from 0 to π with steps of 0.1:

```
for x = 0, math.pi, 0.1 do
    print(x, math.sin(x))
end
```

@ note [There is something you need to know about `for` variables; they only exist inside the block.

```
for i = 1,10 do print(i) end
print(i)
```

You might expect that the last value printed would be 11, but the last `print` will show just `nil`; `i` is *not defined* outside the loop.]

Conditions

Programs often have to make choices.

```
age = 40
if age > 30 then
   print("too old to play games!")
end
```

(Well, that's rude, but computers often are. You can't please everyone.)

The expression in the `if` statement is called a *condition.* For instance, `10 == 10` is always `true`; `2 >= 3` (meaning 'greater or equal') is always `false`.

Like Python and the C-like languages `==` is used for 'is equal', since a single `=` has a very different meaning. Unlike them, 'not equal' is `~=` (the tilde `~` is usually the key on the far left side, next to `!`)

It is common to have two different actions based on the condition:

```
if age > 30 then
    print("still too old")
else
    print("hello, kiddy!")
end
```

And there may be multiple choices:

```
age = 18
if age == 18 then
    print("just right")
else
    if age > 30 then
        print("too old")
    else
        print("too young")
    end
end
```

This style gets irritating if there are more than two conditions. `if-else` statements can be combined together using the single word `elseif` (not the two words `else if`!)

```
age = 18
if age == 18 then
    print("just right")
elseif age > 30 then
    print("too old")
else
    print("too young")
end
```

Conditional expressions can be combined with `and` or `or`:

```
a > 0 and b > 0 or a == 0 and b < 0
```

In this expression, `>` and `==` evaluate first, followed by `and`, and then `or`. So the explicit form would be:

```
((a > 0) and (b > 0)) or ((a == 0) and (b < 0))
```

The `not` operator turns `true` into `false`, and vice versa. Instead of saying `age > 30` you can say `not(age <= 30)`. The parenthesis is needed here because `not` has a higher precedence than `<=`; in fact it has the highest precedence of the logical operators.

Lua will 'short-circuit' logical expressions, For example,

```
if type(n) == 'number' and n > 0 then .... end
```

`n > 0` will give an error if `n` is not a number, but Lua knows that if the first argument of `and` is `false`, then there is no point in evaluating the second argument.

In general, if a condition is `f() and g()`, then if the result of calling `f` is `false` there is no need to call `g`. Simulary, in `f() or g()` Lua will not call `g` if `f` returns `true`.

`and` and `or` in Lua do not only return `true` or `false`. `and` always returns its second argument if both arguments pass, or the argument that does not pass. (Only `nil` and `false` fail a condition.)

```
print (10 and "hello") --> hello
print (false and 42) --> false
print (42 and false) --> false
print(nil and 1) --> nil
```

`or` will return the argument that succeeds (i.e not `nil` or `false`)

```
print (10 or "hello") --> 10
print(false or 42)  --> 42
```

This leads to some common shortcuts in Lua code. For example, this explicit statement for setting a default value:

```
if x == nil then
    x = 'default'
end
```

is often written like this:

```
x = x or 'default'
```

A common pattern for choosing one of two values looks like this:

```
a = 2
print(a > 0 and 'positive' or 'zero or negative')  -- positive
```

`if` statements can be used inside the `for` statement:

```
for i = 1,10 do
    if i < 5 then
        print("lower",i)
    else
        print("upper",i)
    end
end
```

Generally any statement can be 'nested' inside any other statement.

Conditional Loops

This produces the same output as the simple `for` statement example:

```
i = 1
while i < 5 do
    print("hello",i)
    i = i + 1
end
```

The body of the `while` loop is repeatedly executed *while* a condition is true; each time one is added to the variable `i` , until it becomes 6, and the condition fails.

The other loop statement is `repeat`, where we loop *until* a value is true.

```
i = 1
repeat
    print("hello",i)
    i = i + 1
until i > 5
```

For these simple tasks, it is better to use a `for` statement, but the condition in these loops can be more complex.

Tables

Often we need to store a number of values in order, usually called arrays or lists.

Arrays in Lua are done using *tables*, which is a very general data structure. A simple array is easy to define:

```
arr = {10,20,30,40}
```

and then the first element will be `arr[1]`, etc; the length of the array is `#arr`. So to print out this array:

```
for i = 1, #arr do print(arr[i]) end
```

Arrays start with index 1; it's best to accept this and learn to live with the fact.

These arrays are resizable; we can add new elements:

```
arr[#arr + 1] = 50
```

and insert elements at some position using the standard function `table.insert`; the second argument is the index *before* where the value is inserted.

```
table.insert(arr, 1, 5)
```

After these two operations the array now looks like `{5,10,20,30,50,50}`.

Arrays can be constructed efficiently using a for-loop. This creates an array containing the squares of the first ten integers:

```
arr = {}  -- an empty table
for i = 1, 10 do
    arr[i] = i ^ 2
end
```

Please notice that it is not necessary to specify the size of the array up front; a table will automatically increase in size.

To complete the picture, there is `table.remove` which removes a value at a given index.

The function `table.sort` will sort an array of numbers in ascending order.

Lua tables can contain any valid Lua value:

```
t = {10,'hello',{1,2}}
```

So `t[1]` is a number, `t[2]` is a string, and `t[3]` is itself another table. (But do not expect `table.sort` to know what to do with `t` !)

What about trying to access an element that does not exist, e.g. `arr[20]`? It will not raise an error, but return the value `nil`. So it is wise to carefully check what is returned from an arbitrary tab;e access. Since `nil` always indicates 'not found', it follows that you should not put `nil` into an array. Consider:

```
arr = {1,2,nil,3,4}
```

What is `#arr`? It may be 2, but it will definitely not be 5. In other words, it is undefined. Since inserting `nil` into an array causes such a breakdown of expected behavior, it is also called 'inserting a hole'.

The same caution applies to creating arrays that start at 0.

```
arr = {}
for i = 0,9 do arr[i] = i end
```

Well, `arr[0]` will be 0, and `arr[9]` will be 9, as expected. But `#arr` will be 9, not 10. And `table.sort` will only operate on indexes between 1 and 9. (So yes, it is possible to arrange arrays from 0, but the standard table functions will not work as expected.)

It is possible to compare tables for equality, as we did above:

```
a1 = {10,20}
a2 = {10,20}
print(a1 == a2) --> false !
```

The result is `false` because `a1` and `a2` are different *objects*. Table comparison does not compare the elements, it just checks whether the arguments are in fact the same tables.

```
a1 = {10,20}
a2 = a1
print(a1 == a2) --> true !
a1[2] = 2
print(a2[2]) --> 2
```

`a1` and `a2` are merely names for the same thing - `a2` is just an *alias* for `a1`.

Newcomers to Lua are often surprised by the lack of 'obvious' functionality, like how to compare arrays 'properly' or how to print out an array. It helps to think of Lua as 'the C of dynamic languages' - lean and mean. It gives you a powerful core and you either build what you need, or reuse what others have provided, just as with C. The whole purpose of the Lua cookbook is to provide good solutions so you do not have to reinvent the wheel.

There is another use of Lua tables, constructing 'dictionaries' or 'maps'. Generally a Lua table associates values called 'keys' with other values:

```
M = {one=1,two=2,three=3}
```

`M` operates like an array with string indexes, so that we have:

```
print(M['one'],M['two'],M['three'])
--> 1   2   3
```

An unknown key always maps to `nil`, without causing an error.

To iterate over all the keys and values requires the generic `for` loop:

```
for key, value in pairs(M) do print(key, value) end
--> one     1
--> three   3
--> two     2
```

The `pairs` function creates an *iterator* over the map key/value pairs which the generic `for` loops over.

This example illustrates an important point; the *actual* order of iteration is not the *original* order. (In fact the original order is lost, and extra information needs to be stored if you want the keys in a particular order.)

Functions

It is straightforward to create your own functions. Here is a sine function which works in degrees, not radians, using the standard function `math.rad` for converting degrees to radians:

```
function sin(x)
    return math.sin(math.rad(x))
end
```

That is, after the keyword `function` there is the function name, an *argument list* and a group of statements ending with `end`. The keyword `return` takes an expression and makes it the value returned by the function.

Functions are the building blocks of programs; any programmer collects useful functions like cooks collect tasty recipes. Lua does not provide a standard function to print out arrays, but it is easy to write one:

```
function dump(t)
    for i = 1, #t do
        print( t[ i ] )
    end
end
```

A function does not *have* to return a value; in this case we are not interested in the result, but the action.

This `dump` is not so good for longer arrays, since each value is on its own line. The standard function `io.write` writes out text without a line feed:

```
function dumpline (t)
    for i = 1, #t do
        io.write( t [ i ] )
        if i < #t then io.write( "," ) end
    end
    print()
end

dumpline({10,20,30})
--->
10,20,30
```

True to its name, most of this Cookbook is dedicated to giving you functions to do useful things.

Types

The standard function `type` returns the type of any Lua value:

```
type("") == "string"
type(0) ==  "number"
type({})) == "table"
type(print) == "function"
type(nil) == "nil"
```

`nil`

`nil` is a special case; other languages call it `NULL`, `nil`,`Nothing`,`null`,etc. It is however a valid value and has a definite type, which is 'nil'.

Only `nil` and the boolean `false` values make conditions fail.

Otherwise, `nil` is special when used with tables; table elements cannot be `nil`.

Be particularly careful when putting `nil` values into an array; these will become 'holes' and the value of `#arr` will become invalid.

Booleans

A boolean value is either `true` or `false`. The conditional operators, like `>` and `==` return this value. The standard boolean operators `and` and `or` will usually also return 'boolean'.

Only `false` and `nil` make a condition fail.

```
a = 5
print(a > 4)  --> true
print(a == 4) --> false
print(a > 4 and a < 10) --> true
```

Numbers

Lua has only one type of number, which is usually double-precision floating-point on desktop machines. This means you do not usually have to worry about integers and floats; this is one of the simplifications which makes Lua so small and fast. If you worry about storing integers as double-precision floats, remember that integers can be *exactly* represented up to about 1e16.

There are two rounding functions, `math.floor` and `math.ceil`; so given 3.1, the first gives the integer part (3), and the second gives the first integer that is larger (4). `math.ceil(x)==x` is only true if `x` is an integer.

```
function is_integer (x)
    return math.ceil(x) == x
end
```

0 does not fail a condition, unlike C or Python.

Strings

A Lua *string* is usually quoted text, but can actually include any byte value, including the so-called null byte. (You can for instance read a binary file into a Lua string without any corruption.)

Both single and double quotes can be used, with no change in meaning - this is useful if you wish to quote a string inside a string:

message = "cannot find: 'dolly'"

The best advice is: be consistent in your quoting.

Strings compare as you would expect using the locale, so that `s1==s2` behaves sensibly, unlike in Java. In Lua, identical strings are *identical* objects.

There are cases where strings will be automatically considered to be numbers. For instance, `"2"+"3"` will be the number 5. But if any of the strings cannot be converted into a number, then there will be an error: "attempt to perform arithmetic on a string value". It isn't a good idea to depend on this default behaviour, because your program will crash on bad input.

The length operator `#` returns the number of bytes in a string, which is not necessarily the number of characters.

Tables

Lua tables are *associative arrays*; indexing a table is giving it a *key* and getting a value. Any Lua value can be stored in a table, except for `nil`.

They are very efficient at *behaving* like arrays, that is, consecutive integer keys starting at one. The length operator `#` only gives the number of these keys, not the total number of keys.

Functions

In Lua, functions are *first-class values* - that is, they can be passed around like any other value.

Coroutines

Userdata

Strings

The `tonumber` function will explicitly convert a string to a number; it will return `nil` if the conversion is not possible. It can also be used to convert hexadecimal numbers like so:

```
val = tonumber('FF',16) -- result is 255
```

How about converting numbers to strings? `tostring` does the general job of converting any Lua value into a string. (The `print` function calls `tostring` on its arguments.) If you want more control, then use the `string.format` function:

```
string.format("%5.2f",math.pi) == '"3.14"
```

These `%` format specifiers will be familiar to C and Python programmers, but basic usage is straightforward: the 'f' specifier has a total field with (5) and a number of decimal places (2) and gives fixed floating-point format; the ‘e’ specifier gives scientific notation. ’s’ is a string, ‘d’ is an integer, and ‘x’ is for outputing numbers in hex format.

```
print(string.format("The answer to the %s is %d", "universe", 42) )
-->
The answer to the universe is 42
```

There is a set of standard operations on strings. We saw that ‘adding’ strings would try to treat them as numbers. To join strings together (*concatenate*) there is the `..` operator:

```
"1".."2" == "12"
```

Most languages use `+` to mean this, so note the difference. Using a different operator makes it clear, for instance, that `1 .. 2` results in the *string* ‘12’.

As with arrays, `#s` is the length of the string `s`. (This is the number of *bytes*, not the number of characters.)

The opposite operation is extracting substrings.

```
string.sub("hello",1,4) == "hell"
string.sub("hello",4) == "lo"
```

The first number is the start index (starting at *one*, as with arrays) and the second number is the final index; the result includes the last index, so that `sub(s,1,1)` gives the first ‘character’ in the string:

```
-- printing out the characters of a string
for i = 1,#s do
   print(string.sub(s,i,i))
end
```

It is not possible to treat a string as an array - `s[i]` is not meaningful. (It will just silently return `nil`) A Lua string is not a sequence of characters, but a read-only lump of bytes; it is not very efficient to process a string by iterating over its bytes and in fact Lua provides much more powerful techniques for string manipulation.

For instance, a naive solution to the problem of finding a character in a string involves looking at one character at a time; the `string.find` function is faster and less trouble.

```
string.find('hello','e') == 2
```

In general, this function will return *two* values, the index of the start and the finish of the matched substring:

```
print(string.find('hello','lo'))
--> 4       5
```

(Which are exactly the numbers you need to feed to `string.sub`.)

This may not seem so very useful, because we knew the length of the substring. However, `string.find` is much more powerful than a simple string matcher.

In general, the ‘substring’ is a Lua string pattern. If you have previously met regular expressions, then string patterns will seem familiar. For instance, the string pattern ‘l+’ means ‘one or more’ repetitions of ‘l’.

```
print(string.find('hello','l+'))
--> 3       4
```

‘Character classes’ make string patterns much more powerful. The pattern ‘[a-z]+’ means ‘one or more letter in the range ’a’ to ‘z’:

```
print(string.find('hello','[a-z]+'))
--> 1       5
```

That is, it matches the whole string. So we could write a function `is_lower` like so:

```
function is_lower(s)
    i1,i2 = string.find(s,'[a-z]+')
    return i1 == 1 and i2 == #s
end
```

But there is a neater way. The pattern ‘[1]+′*doesthejob*, *sinceitsaysthatthesequenceofoneormorelettersmuststa*
So `string.find` will return `nil` for ’ hello’.

Lua provides names for common character classes; ‘%a’ is short for ‘[a-zA-Z]’ and ‘%d’ is short for ‘[0–9]’. ‘%s’ stands for any whitespace, i.e. ‘[⁀]’. The capital letter versions stand for any characters *not* in the set, so ‘%S’ stands for anything that is not a space. So the pattern ‘^%S+$’ will match any sequence of characters that does not contain a space. (These are different from the usual regular expression syntax, which is to use a backslash. So Lua patterns tend to be easier to read than regular expressions. However, they are more limited.)

String patterns are an important part of learning Lua well, and we will return to them in this Cookbook. But you should always be aware of them, because

[1] a-z

`string.find` normally assumes that the match is a pattern that contains ‘magic’ characters. For instance, ‘$'standsfor'endofstring';ifyouwantedtofindanactual'$’ in a string then you have two options:

- *escape* the magic character like so: ‘%$’
- use `string.find(s,sub,1,true)`; the last argument means ‘plain match’.

`string.match` is similar to `string.find`, except that it does not return the index range, but rather the match itself.

```
print(string.match('hello dolly','%a+'))
--->
hello
```

Here the pattern means ‘one or more alphabetic characters’, so the match gives us the first word. You could do this with a combination of `string.find` and `string.sub`, but `string.match` is more general and efficient. Consider:

```
print(string.match('hello dolly','(%a+)%s+(%a+)'))
--->
hello     dolly
```

Here `match` returns *two* matches, which are indicated using parentheses in the string pattern. These are often called *captures*. So the pattern would read like this ‘capture some letters, skip some space, and capture some more letters’.

A very powerful function for modifying strings is `string.gsub` (for *global substitute*):

```
print(string.gsub('hello help','e','a'))
--> hallo halp     2
```

It replaces *each* match of the pattern with a given string, and returns the resulting string and the number of substitutions. There can also be a fourth argument which lets you set the maximum number of substitutions:

```
print(string.gsub('hello help','e','a',1))
--> hallo help      1
```

(There is no form that does a ‘plain match’ like `string.find` so you will have to be careful to escape magic characters.)

Finally, there is `string.gmatch` which iterates over all the matches in a string. A common task is finding all the words in a string, separated by spaces. The pattern ‘%S+’ means ‘one or more non-space character’, but `string.match` will only give you a fixed number of matches.

```
local str = 'one  two   three'
for s in string.gmatch(str,'%S+') do
    print('"'..s..'"')
end
-->
"one"
"two"
"three"
```

This suggests the following useful function, which breaks up a string into a table of words:

```
function split(str)
    local t = {}
    for s in string.gmatch(str.'%S+') do
        t]#+1] = s
    end
    return t
end
```

To split a string with other delimiters is just a matter of choosing the right pattern. For instance, '[%S,]+' matches 'one or more characters from the set of non-space and comma'. You could use this to split 'one, two, three' into `{'one','two','three'}`.

The special pattern '.' matches *one* arbitrary byte. So

```
for c in string.gmatch('.') do print(string.byte(c)) end
```

prints all the byte codes in a string.

Tables

It is commonly said that a Lua table has both an array and a map part. It is more correct to say that it can be used both ways.

Here is a more general table - note that the length operator `#` does not see any non-array keys:

```
T = {10,20,30,40,50; sorted=true}
print(t [1], #t, t [#t])  --> 10    5    50
print(t["sorted"],  t.sorted) --> true     true
```

In Lua, `M.one` is *defined* to be `M['one']`. This gives us a way to do 'objects' or 'structures' with tables:

```
-- 'point' objects
s1 = {x = 1, y = 2}
s2 = {x = 0, y = 4}
-- adding two points
sum = { x = s1.x + s2.x, y = s1.y + s2.y }
print(sum.x, sum.y) --> 1    6
```

Something like `t.function` is a syntax error, because `function` is a keyword. In this case you must say `t["function"]`. The general way to construct such table is

```
t = {
  ["function"] = 1,
  ["end"] = 2
}
```

In Lua, tables are used to represent arrays. But inserting a `nil` value creates a hole, and the length operator `#` is no longer defined.

You must be careful not to insert any holes by accident. Consider a task in which you want to get a list of objects from a list of names using a function `getObject` which may return `nil`. You furthermore want the resulting list to have the same length as the list of strings:

```
local null = {}
local objects = {}
for i = 1,#strings do
    local obj = getObject(strings[i])
    if obj then
        objects[i] = obj
    else
        objects[i] = null
    end
end
```

That is, put some distinct and unique value into the array that stands for `nil`. Then when using the list of objects, test for this unique value:

```
for i = 1,#objects do
    local obj = objects[i]
    if obj == null then
        print(i,'nada')
    else
        print(i,tostring(obj))
    end
end
```

The list of objects remains a perfectly good array.

(In this code, variables are *explicitly* declared as being local; we'll see why this is such a good idea later.)

Most of the `table` functions are meant to operate on arrays; you can sort them with `table.sort`. In the simple case the table must only contain numbers (or sortable objects), but a sort function can be provided.

Arrays of strings can be joined together with `table.concat`. This is the most efficient way to build up large strings:

```
local res = {}
for i = 1,1000 do
    res[i] = "hello "..i
end
str = table.concat(res)
```

(Note that `table.concat` is picky about the array elements; they must either be numbers or strings. If in doubt, use `tostring` to convert values first.)

The most commonly used function with tables is `pairs` which allows you to iterate over all the elements. There is a corresponding function `ipairs` which works for arrays:

```
for i,v in ipairs{10,20,30; x=1} do print(i,v) end
--> 1   10
--> 2   20
--> 3   30
```

Unlike `pairs`, `ipairs` will give you the keys in the correct order, and it will only access the array elements.

Coroutines

Although coroutines are sometimes called threads (and `type` returns 'thread' as their type) they are not operating system threads as usually understood. (Lua sometimes runs on machines that don't even have the luxury of an operating system.)

Rather, a coroutine is another kind of 'function with memory'. Once a coroutine is created from a regular Lua function, it can be resumed; the coroutine yields and waits for the caller to resume it again. Unlike a function which is repeatedly called, it resumes exactly at the point where it last yielded.

```
function coco()
    print '1'
    coroutine.yield(1)
    print '2'
    coroutine.yield(2)
    print '3'
    coroutine.yield(3)
end

c = coroutine.create(coco)
print(coroutine.resume(c))
print(coroutine.resume(c))
print(coroutine.resume(c))

----> output:
1
true    1
2
true    2
3
true    3
```

The first `resume` call causes the first `yield` call (after printing 1) and we get the value that was passed to `yield`. The second `resume` call makes the `yield` return and we print out 2, etc. The key idea is that the coroutine *remembers where it was* and resumes where it last was executing.

So each coroutine preserves its complete state, and is sleeping when not explicitly resumed. This is often called 'cooperative multitasking' because one coroutine has to yield for another coroutine to resume.

`coroutine.wrap` will construct a function which resumes a coroutine.

```
f = coroutine.wrap(coco)
print(f()) -- 1
print(f()) -- 2
print(f()) -- 3
print(f()) -- nil
```

(This is convenient, but if there is a problem an error will be raised and will need to be handled. With `coroutine.resume` the first returned value is the status; if it's `false` then the second value is the error, just like `pcall`.)

This is exactly what Lua considers a simple iterator: a function which can be repeatedly called, returning new values and indicating the end with a `nil`.

Coroutines help in making iterators where the logic is not so straightforward as the previous examples. Consider the following data structure:

T = { value = 10, left = { value = 20, left = {value = 40}, right = {value = 50}, }, right = { value = 30 } }

This kind of structure is often called a *tree*, in this case a binary tree because there are at most two branches at each note. The end nodes are often called *leaves* because they have no branches.

A recursive function for printing all the values s easy to write

```
function traverse(t)
    if t.left then traverse(t.left) end
    print (t.value)
    if t.right then traverse(t.right) end
end

traverse(T)
-->
40
20
50
10
30
```

This function will keep visiting the left branches until there isn't any, so it first visits `left = {value = 20..` and then `left = {value = 40}`; this node has no `left` so it prints out `value` (40), and so on.

Now comes the key part: replace the `print` with `coroutine.yield` and turn it into a coroutine:

```
function traverse(t)
    if t.left then traverse(t.left) end
    coroutine.yield (t.value)
    if t.right then traverse(t.right) end
end

c = coroutine.create(traverse)
print(coroutine.resume(c,T)) -- true   40
print(coroutine.resume(c,T)) -- true    20
```

This can be made into an iterator if we make a function of no arguments from this coroutine:

```
function tree_iter(t)
    return coroutine.wrap(function()
        return traverse(t)
    end)
end

for v in tree_iter(T) do print(v) end
 -->
40
20
50
10
30
```

`tree_iter` returns the iterator. Each time it is called, the coroutine yields, returning the value of the node. We keep yielding until the function finishes, when it returns `nil`.

Programs

Lua is often called a 'scripting langauge' which implies that it is only suitable for bashing out little scripts to do specific tasks. This is not true; many commercial games have much of their functionality written in Lua, as do other products like Adobe Lightroom. It has always been the fastest of the dynamic languages, typically several times faster than Python or Ruby, and the LuaJIT just-in-time compiler can give performance equivalent to conventional compiled languages.

However, there are certain habits that are essential in writing robust Lua code which can be safely used to build large applications. The difference between a 'script' and a 'program' is fairly arbitrary. A script is often defined as a small program that does a specific task that is meant as a tool for knowledgeable users. Compilation does not magically transform a script into a program, however the built-in discipline of statically-typed languages helps to debug faults before actually running them. Everything is declared, and everything has a definite type. With a dynamically-typed language, you have to provide that discipline.

For example, a script can use only global variables, and work fine. But experience shows that you will enter a zone of frustrating debugging if you write non-trivial applications in this way.

Lua is in fact compiled before it executes, although commonly the generated 'bytecode' is not saved, since the compiler is so fast as to be practically instantaneous on everything except the largest programs. This means that syntax errors (like forgetting to say `then` after `if`) will be caught immediately.

However, these are easy errors to avoid and you will find yourself making fewer with time, since Lua has a simple syntax and there are not too many rules to remember. But misspelling variables is not a compiler error, and not a run-time error either, as we will see.

For a larger application, it is important to spread it out over multiple files. So it is important to know how to build and use libraries of dependable code. By knowing what libraries are available in the Lua universe, you can avoid re-inventing wheels and get on with building the car.

Modules

The global function `dofile` will evaluate a Lua file. However, it is not very useful for organizing libraries and large programs. It takes a full path to the Lua file, which is a problem for programs that need to be portable, and it will always load the source file each time.

`require` solves these problems by looking for the Lua file in standard locations, and only loading it once.

`require` is given a *module name*, without an extension. To understand how the modules are found, it is useful to look at the error message you get when a module is *not* found:

```
require 'alice'
---> a Linux machine
lua: module 'alice' not found:
    no field package.preload['alice']
    no file './alice.lua'
    no file '/usr/local/share/lua/5.1/alice.lua'
    no file '/usr/local/share/lua/5.1/alice/init.lua'
    no file './alice.so'
    no file '/usr/local/lib/lua/5.1/alice.so'
    no file '/usr/local/lib/lua/5.1/loadall.so'
--> a Windows machine
lua: module 'alice' not found:
    no field package.preload['alice']
    no file '.\alice.lua'
    no file 'C:\Program Files\Lua\5.1\lua\alice.lua'
    no file 'C:\Program Files\Lua\5.1\lua\alice\init.lua'
    no file 'C:\Program Files\Lua\5.1\alice.lua'
    no file 'C:\Program Files\Lua\5.1\alice\init.lua'
    no file 'C:\Program Files\Lua\5.1\lua\alice.luac'
    no file '.\alice.dll'
    no file '.\alice51.dll'
    no file 'C:\Program Files\Lua\5.1\alice.dll'
    no file 'C:\Program Files\Lua\5.1\clibs\alice.dll'
    no file 'C:\Program Files\Lua\5.1\loadall.dll'
    no file 'C:\Program Files\Lua\5.1\clibs\loadall.dll'
```

The first place where `require` looks is the current directory; if there was a file `alice.lua` in the same directory as this program, it would be loaded. Then it will look in a few standard places - for Unix this is usually the directory `/usr/local/share/lua/5.1` and for Lua for Windows it is the `lua` subdirectory of the installation directory.

Then things get interesting; it starts looking for `.so` files on Unix and `.dll` files on Windows - these are *binary Lua extensions* which are shared libraries, usually written in C. So `require` will load both pure Lua and binary modules.

How does Lua know which locations to search? The system variable `package.path` contains the locations where Lua modules will be found. it is a string containing *patterns* separated by semi-colons: (I've printed these patterns on separate lines for clarity)

```
print(package.path)
-->
./?.lua;
/usr/local/share/lua/5.1/?.lua;
/usr/local/share/lua/5.1/?/init.lua;
```

The procedure to find module `alice` is simple; take each of these patterns, and replace `?` with 'alice'; if the result is an existing file, load it.

(Remember that Lua has *no* built-in concept of a file system; this search is done by string substitution followed by an attempt to open the result.)

If this search fails, then Lua will use `package.cpath` to find a binary extension in a similar way. So you can see that the error message is detailing exactly what happens in this process.

So, to summarize, `require`:

- takes a module name, not a file name
- looks for both Lua files and shared libraries (binary extensions)
- only loads the module once
- looks on the Lua and extension paths

A simple way to try this out is to create a Lua file `mod1.lua`:

```
-- mod1.lua
print("hello")
```

and require it from `usemod1.lua`, in the same directory:

```
-- usemod1.lua
require("mod1")
print("dolly")
-->
hello
dolly
```

Which shows that the file loaded by `require` can be any Lua code. Actually, the word 'module' is just a word we use to describe Lua files that are generally useful for different programs to use - there is no formal concept of 'module' in the language itself.

Modules are usually used to make functions and data available to a program. It is better to put these into their own table:

```
-- mod2.lua
mod2 = {}
mod2.answer = function()
    return 42
end
```

Now, after `require("mod2")` our code can call `mod2.answer()`. There can now be a number of functions called `answer` in the program and they will not interfere with each other.

If you move `mod2.lua` to a location on your Lua path (such as '/usr/local/share/lua/5.1/') then a Lua program anywhere on the system can load the module `mod2`.

Of course, to copy files into the system-wide directory often requires superuser privileges. These standard directories are built into Lua and may not be convenient for your purposes.

You can add other directories to the Lua module path by setting the environment variable `LUA_PATH`. For a Bash shell on Unix this would look like:

```
export LUA_PATH=";;/home/steve/lualibs/?.lua"
```

On Windows:

```
set LUA_PATH=;;c:\lua\lualibs\?.lua
```

Please note that the semi-colon is used to separate patterns on all systems. The double semicolon at the start means "append this to the existing Lua module path" so Lua will first look in the system-wide directory for modules.

So, there is nothing special about a Lua file loaded by `require`. Generally, it should create a table containing the functions. A common pattern looks like this:

```
-- mod3.lua
local mod3 = {}

function mod3.fun1()
  return 42
end

function mod3.fun2()
  return mod3.fun1()
end

return mod3
```

Note that it *returns the table.* A user of the module may now say:

```
local mod3 = require 'mod3'
```

This way of writing modules does not create a global table. Consider the effect of leaving out the `local` when declaring `mod3`: then the table *is* available globally. You can then use it like follows:

```
require 'mod3'
print(mod3.fun1())
```

It's *always* a good idea to return the value of a module. If you do not, then `require` simply returns `true`, which is not very useful. Lua programmers expect `require` to return a table containing the functions.

Please note that functions in a module may depend on each other, but they must reference them using the table explicitly.

Using global tables of functions is going out of fashion in the Lua world, because it leads to the problems discussed in the section on variables and scope. Particularly in a big program, the aim is to keep the number of globals floating around to the absolute minimum.

There is a common style introduced in Lua 5.1 for creating modules, which many libraries use. The `module` function creates a named table of functions and sets the environment of the file so that these functions may be declared and without explicit reference to the table:

```
-- mod4.lua
module("mod4")

function fun1()
  return 42
```

```
end

function fun2()
  return fun1()
end
```

This has the same affect as `mod3` without the `local` declaration; `require` returns the table, and a global variable `mod4` is created which refers to the table.

It is convenient that the module functions can call each other directly, but there is a tricky little issue:

```
-- mod5.lua
module("mod5")

function dump(msg)
  print(msg)
end
```

Calling `mod5.dump` leads to a run-time error:

```
.\mod5.lua:5: attempt to call global 'print' (a nil value)
stack traceback:
        .\mod5.lua:5: in function 'dump'
...
```

This is because we are not inside the global environment, but inside the environment created by `module`. The recommended fix is to insert the following line *before* the call to `module`:

```
local print = print
```

The other fix is to make the module call look like this:

```
module("mod5",package.seeall)
```

This makes all the global functions available to the module. It's beter however to define what you need explicitly at the top of the module, since then readers can see at a glance what the dependencies are. (Last, but not least, using locals is significantly faster.)

Scope of Variables

There are two kinds of variables in Lua, local and global. Globals are visible throughout the *whole* program, The standard functions in Lua are mostly contained in global tables, like `math` and `table`. An assignment like this:

```
newGlobal = 'hello'
```

causes `newGlobal` to be publically available - it is not just visible in the declared file.

Global variables are contained in the global table. The variable `_G` refers to this table explicitly, so `_G.print` works as expected. But otherwise there is nothing special about `_G` and setting it to some other value has no effect on program operation.

To understand global functions you need to remember how tables and functions work. It is easy to change the behavior of the whole program by redefining a global:

```
function print(x)
    io.write(tostring(x),'\n')
end
```

This is *totally* equivalent to the following table key assignment:

```
_G["print"] = function(x) ... end
```

So you will not be surprised when this causes unexpected behavior - suddenly every call to `print` works differently (since the new version only takes one argument.) Sometimes this technique is useful and it has a name: "monkey patching". But generally it is a disaster waiting to happen, because it messes with people's expectations of how a standard function works.

Lua can not tell you that a function is undefined at compile-time. If you misspell a name `newGlbal` then it will just tell you that you have tried to call a `nil` value, because `_G["newGlbal"]` is `nil`.

Local variables, on the other hand, are only visible in the block where they have been declared.

There are two places where local variables are implicitly declared; the first is the argument list of a function, and the second is the variables defined by a `for` statement. In this function, `x`,`y` and `z` are local because they are arguments, and `k`,`v` are locals because they are `for` variables.

```
function fun(x,y,z)
    for k,v in pairs(x) do
        ....
    end -- B2
    print(k)
end -- B1
```

Local variables always have a 'scope', which is the the part of the program where they are valid and visible. In this function, `x` is visible up to the end of block B1 and `k` is visible up to the end of block B2. In particular, `k` is not visible in `print(k)`! And If a variable is undeclared, then Lua assumes it is global.

The problem then is that a misspelling is *not an error* , just `nil` !

Locals may be explicitly defined using the `local` keyword. This function makes a 'shallow copy' of a table by creating a new table and copying the key/value pairs:

```
function copy (t)
    local res = {}
    for k,v in pairs(t) do
        res[k] = v
    end
    return res
end
```

`local` is the correct way to do this, because then the variable `res` is not visible outside the function. The function is 'pure', i.e. it has no global side-effects.

Local declarations can 'nest' inside each other. Inside the for-loop, `k` has a different meaning to `k` outside the loop.

```
function dumpv( t )
    local k = 1
    for k.v in pairs(t) do
        print(k,v)
    end
    print(k) -- 1  !
end
```

Good programming practice is to make everything as local as possible. As a bonus, accessing locals is significantly faster than accessing globals. Each global access involves a table lookup, whereas locals define 'slots' which are much more efficient.

Errors

Because Lua functions may return multiple values, they can return error strings together with the status code. The classic example is `io,open` which usually returns a file object, but will return both a `nil` and an error message if the file cannot be opened:

```
> = io.open("hello.txt")
nil hello.txt: No such file or directory    2
```

However, the Lua libraries aren't consistent in error handling. The `table` functions throw errors:

```
> t = {'h',{}}
> = table.concat(t)
stdin:1: invalid value (table) at index 2 in table for 'concat'
stack traceback:
    [C]: in function 'concat'
    stdin:1: in main chunk
```

Even in the most careful programs, something will go wrong, and you do not want users to witness a crash with a stack traceback. Also, it is often possible to recover from the situation. Languages with exception handling encourage the 'throwing' or 'raising' of errors because then the error handling code is kept away from the 'normal' logic - what is sometimes called 'the happy path'.

There is no `try/except` statement in Lua, but there is a function `pcall` (or 'protected call') which will call a Lua function safely. It will return the status and anything returned by the function. If the status is `false`, then the second return value is the error message.

```
> = pcall(table.concat,t)
false   invalid value (table) at index 2 in table for 'concat'
> = pcall(table.concat,{'one','two'})
true    onetwo
```

Anonymous functions are useful for blocks that can raise errors:

```
local status,err = pcall(function()
    do_something()
    do_another()
    and_again()
end)
if not status then
    print('error',err)
end
```

You can raise your own errors, using `error`. In its simplest form it will indicate that the error came from the current line:

```
--> error is reported at this line; program will give a stacktrace
if not right then error("was not right!") end
```

There can also be a second parameter that indicates where the error came from:

```
function not_right_error()
  error("was not right",2) -- i.e, _calling_ function raised this
end

--> error still reported at this line
if not right then not_right_error() end
```

The function `assert` raises an error if a condition fails:

```
--> message 'assertion failed...."
assert(n > 0)
--> message "size must be positive"
assert(n > 0,"size must be positive")
```

The equivalent Lua version looks like this:

```
function assert(value,message)
    if not value then
        if message == nil then message = "assertion failed" end
        error(message,2)
    else
        return value
    end
end
```

Metatables

Any table or userdata may have a *metatable* associated with it. Metatables specify the behaviour of their tables, in a similar way to how classes specify the behaviour of their objects in a traditional object-orientated language. Metatables do this by defining *metamethods*. In other respects, metatables are plain tables.

`tostring` will try to convert a value into a string; it is used by `print` when presenting values. It is not particularly useful for tables,

```
obj = {label = "hello"}
print(obj) --> table: 0x80e91f8
```

`tostring` does let an object decide how to show itself. If the object has a metatable, and that metatable contains a function called `__tostring`, it will use that function to get the string:

```
local MT = {
    __tostring = function(obj)
        return obj.label
    end
}

setmetatable(obj,MT)

print(obj) --> hello
```

A more realistic example involves making tables print out their contents. In the specialized case of an array or list, it would look like this:

```
List = {
    __tostring = function(list)
        local res = {}
        for i = 1,#list do
            res[i] = tostring(list[i])
        end
        return '{'..table.concat(res,',')..'}'
    end
}

function new_list(t)
    return setmetatable(t or {},List)
end

l1 = new_list{10,20,30}
print(l2) ---> {10,20,30}
```

This is genuinely useful when working with lists. The `__tostring` implementation is a little more complicated than it should be, because `table.concat` itself does not use `tostring` and needs a table of strings or numbers. (This is actually deliberate, since `table.concat` is *the* way to build large strings for output in Lua. If you make a mistake and pass a table value or `nil`, then it is better for this to be a runtime error than to have to look through ten pages of output for the mistake.)

The ‘constructor’ `new_list` takes an existing table, or makes a new table. (`t or {}` is a common Lua idiom for specifying a default value for the case when something may be `nil`.) It would be more elegant if `List` could be used as a constructor, like `List{10,20,30}`. The `__call` metamethod makes a table *callable*. When you try to ‘call a table’ then Lua looks for a function with this name in the metatable, passing the original table:

```
setmetatable(List,{
    __call = function(C,t) return new_list(t) end
})

l2 = List{10,20,30}
print(l1 == l2) --> false
```

It would be useful if lists could be compared element-by-element when using the equality operator `==`. By defining the metamethod `__eq`, the usual behaviour is *overriden*.

```
List.__eq = function(list1,list2)
    if #list1 ~= #list2 then return false end
    for i = 1,#list1 do
        if list1[i] ~= list2[i] then return false end
    end
    return true
end

print(li1 == l2)   --> true
```

`List` is starting to resemble what people call a *class* in other languages; it serves both as a factory for making new `List` objects, and defines shared behaviour in all these objects.

The basic table access operations using a key are getting and setting values. If a key is not found in the table, then `nil` is returned. Sometimes this is not an appropriate default value, or you want this case to really be an error. The `__index` metamethod is called if Lua *cannot* find a key in a table. It is passed the table and the key:

```
local ErrorMT = {}

function ErrorMt.__index(t,key)
    error("Cannot find '"..key.."' in table",2)
end

function new_strict_table(t)
```

```
    return setmetatable(t or {},ErrorMT)
end

t = new_strict_table {fred = 1}
--> case-sensitive!  Throws an error "Cannot find 'Fred' in table"
print(t.Fred)
```

Consider counting unique words in a table:

```
for i = 1,#words do
    local word = words[i]
    local count = count_of[word]
    if not count then
        count_of[word] = 0
    end
    count_of[word] = count + 1
end
```

Having to check for the first occurance of a word is irritating; if the key is not found, the value should be zero.

```
ZeroMT = {
    __index = function(t,key) return 0 end
}
```

Then tables with this metatable can be used like this:

```
for i,word = ipairs(words) do
    count_of[word] = count_of[word] + 1
end
```

Going back to the `List` example, it would be nice if lists had some methods. The `extend` method will append all the elements of another table:

```
local methods = {}

List.__index = function(list,k) -- *note*
    return methods[k]
end

function methods.extend(self,list)
    local k = #self
    for i = 1,#list do
        k = k + 1
        self[k] = list[i]
    end
    return self
end

ls = List{10,20,30}
ls:extend {40,50}
print(ls) --> {10,20,30,40,50}
```

That is, if we can't find a key such as `extend` then `__index` assumes that it will be inside a table of methods.

(This is such a common pattern that Lua allows `__index` to be a table, so you can write the marked line as:

```
List.__index = methods
```

A little less flexible, but faster and cleaner.)

It's easy to keep adding methods. For instance, this works like `string.sub`, where the second index can be negative to refer to the end of the list:

```
function methods.sub(self,i1,i2)
    if i2 == nil then
        i2 = #self
    elseif i2 < 0 then
        i2 = #self + i2 + 1
    end
    local res = List()
    for i = i1,i2 do
        res[#res + 1] = self[i]
    end
    return res
end

print(ls:sub(1)) --> {10,20,30,40,50}
print(ls:sub(2,-2)) --> {20,30,40}
```

Such a section of a list is usually called a 'slice'. Note that calling with a start index of 1 gives us a copy of the full list.

To make this list even more Python-like, we can implement *concatenation.* This is the new list made from appending the second list to the first list. `l1 .. l2` can be made to do this if the `__concat` metamethod is defined:

```
List.__concat = function(l1,l2)
    local res = self:sub(1)  --- make a copy
    return res:extend(l2)    --- and append l2
end

print(List{10,20} .. List{30,40} .. List{50}) --> {10,20,30,40,50}
```

To recap: the behaviour of any table can be changed by giving it a *metatable*, which contains *metamethods*. The most important of these is `__index` which allows you to handle the case where a key is not found in the table. It may simply point to a table which is used as the 'fallback' table which is examined after the first lookup fails. This is a common way to implement object-oriented programming in Lua, allowing the methods for all objects of a given 'class' to be stored in one place.

You could define `l1 + l2` to mean this using the metamethod `__add`, but Lua programmers would not expect addition to work like that. They would expect that `l1 + l2` is the list made by adding the corresponding elements of the two lists ('element-wise'), and it would of course only work with lists where the elements *could* be added.

The operation of setting a value can also be customized. `__newindex` works like `_index`; it is *only* called if the key is not in the table. It receives three arguments, the table itself, the key and the new value.

What if you want `__index` and `__newindex` to be *always* called? Then the keys by definition cannot be in the table itself. Instead, the table must act as a *proxy* for another table. Say we have an array and want to raise a 'out of bounds' error instead of just silently returning `nil`.

```
local ProxyListMT = {}

function new_array(arr)
    return setmetatable({_data = arr},ProxyListMT)
end

function ProxyListMT.__index(self,k)
    if k < 1 or k > #self._data then error("out of bounds",2) end
    return self._data[k]
end

function ProxyListMT.__newindex(self,k,value)
    if k < 1 or k > #self._data then error("out of bounds",2) end
    self._data[k] = value
end
```

That is, the object just contains a `_data` field which refers to the actual table; any time we access an array element, it must go through `__index` because there are no array elements in the object itself; it is acting as a proxy for the table. The object behaves like a non-resizable array. It is naturally not quite as efficient, but it is often more important to be *correct* than *fast*.