Sam Discusses Lunacy

Table of contents

Installing Lunacy 2
Running Lunacy 3
Desktop calculator 4
Numbers 5
Comments 8
Strings 9
Binary numbers 12
Tables 16
Metatables (Classes) 19
Loops (While and repeat) 24
For loops with functions 27
List iteration 30
Multiple inheritance 31
Regular expressions 34

Installing Lunacy

Lunacy is my personal fork of Lua 5.1. *Lua* is a tiny scripting language which is suitable for embedding in other applications. While there are newer versions of Lua, Lua 5.1 is the most widely deployed and used. Lua 5.1 is the most recent version supported by LuaJIT, a very fast implementation of Lua; Lua 5.1 is also the basis for Luau, the scripting language used by the very popular gaming platform Roblox, as well as the Scribunto MediaWiki extension which allows the Wikipedia to run Lua scripts.

To install Lunacy depends on the operating system one is using. In Windows (Windows XP, Windows 7, Windows 8, Windows 10, and Windows 11) one can simply use the pre-compiled Lunacy binary. In Linux and MacOS, if one has a C compiler and an implementation of *make*, it is possible to compile Lunacy.

The simplest way to download Lunacy is to get it from GitHub using the *git* command. From a command line, run this:

```
git clone https://github.com/samboy/lunacy
```

One option to obtain Lunacy without git is to go to `https://github.com/samboy/lunacy`, click on “Code” then “download zip”. Another option is to go to `https://maradns.samiam.org/lunacy/` and download Lunacy there.

Should GitHub be down, Lunacy is available at other locations on the internet, e.g. `https://codeberg.org/samboy/lunacy` or `https://gitlab.com/maradns/lunacy/`

Once Lunacy is downloaded, Windows users can use the `lunacy.exe` 32-bit x86 compile of Lunacy in the `bin/` folder of the download. People on other platforms can compile Lunacy with `make` and a C compiler (`gcc`, `clang`, etc.) in the `src/` folder. The simplest way to compile Lunacy is to simply run `make` in the `src/` folder of Lunacy; if readline support is desired (to have up arrow history) and the, one can compile Lunacy via `make -f Makefile.readline` If one has the editline library instead (similar to readline, but without the GPL license), one can compile Lunacy via `make -f Makefile.editline`

Once Lunacy is compiled, one can copy it over to one’s local path (e.g. `/bin/` or `/usr/bin/` or another directory in one’s usual path) as root or admin via the appropriate copy command (e.g. `cp lunacy /usr/bin/`).

The Windows binary of Lunacy has up arrow history support if it’s run in Cygwin (available at `https://cygwin.com/`) or PowerShell.

Lunacy is a command line application, so it is best run from a terminal window, such as Cygwin, PowerShell, any Linux terminal application, or the terminal in MacOS.

That said, the `lunacy.exe` binary runs just fine when double clicked on in Windows 10, and even has up arrow history support.

Running Lunacy

Once Lunacy is installed, running the `lunacy` binary (e.g. `lunacy.exe` on Windows) will show something that looks like this:

```
Lunacy 2022-12-06  Copyright 1994-2012 Lua.org PUC-Rio; 2020-2022 Sam Trenholme
>
```

Here we can begin to type in commands. In the examples on this page, text in bold **`like this`** is typed by the user; text not in bold `like this` is returned from the Lunacy binary.

Let's do the "Hello, world!" example:

```
> print("Hello, world!")
Hello, world!
```

We can also use Lunacy as a basic desktop calculator:

```
> 1 + 1
2
```

Note that the trick that makes the answer to an expression appear only works if the first character in the line is a number or the left parenthesis "`(`" character. Other characters need `=` at the beginning of the line, e.g.

```
> = math.pi
3.1415926535898
```

Lunacy has support for functions, e.g.

```
> function hi()
>> print("Hello, world!")
>> end
> hi()
Hello, world!
```

If Lunacy is being run from the command shell, it is possible to tell Lunacy to run a given file as its script contents. For example, let us suppose we have a file `foo.lua` with the following in it:

```
print("Hello, world!")
```

Running Lunacy in the same directory as `foo.lua` gives us a result like this:

```
$ cat foo.lua      This would be "type foo.lua" in a Windows cmd shell terminal
print("Hello, world!")
$ lunacy foo.lua
Hello, world!
```

Where `$` is a Cygwin, Linux terminal, MacOS terminal, or similar prompt.

Desktop Calculator

Lunacy can be used as a desktop calculator. To do so, Lunacy can be started from the command line in a terminal window. For example, anything in bold is typed by the user:

```
$ lunacy
Lunacy 2022-12-06  Copyright 1994-2012 Lua.org PUC-Rio; 2020-2022 Sam Trenholme
> 1 + 1
2
> (1 + 2) * 3
9
```

`+` is addition; `-` is subtraction; `*` is multiplication; `/` is division.

Lines which begin with a number of the `(` character are calculated. Should an expression who which starts with another character is desired, the line needs to start with the `=` character.

```
> =math.pi
3.1415926535898
```

It's possible to assign and use variables in the Lunacy command line:

```
> a = (34 * 3) + 2
> = a * 2
208
```

It's also possible to define functions (or do pretty much anything Lunacy can do when run as a script) when being run as a desktop calculator:

```
> function addOne(x)
>> print(x + 1)
>> end
> =addOne(3)
4
```

A desktop calculator will usually have arrow history support; in other words, if one hits the up arrow character while using Lunacy in desktop calculator mode, the previously entered command will appear again and can be edited with the left and right arrow keys. The 32-bit Windows Lunacy has arrow support when run in Cygwin, PowerShell, or from the Windows cmd shell; the source code to Lunacy has support for both readline and editline which also can give Lunacy this support.

Lunacy does not store the state of its virtual machine; every single time Lunacy is started as a fresh slate; functions and variables stored in previous invocations of Lunacy are lost.

NUMBERS

The version of Lua I use is Lunacy, which is a fork of Lua 5.1.

In Lunacy, all numbers are IEEE 64-bit floats. This means:

- Numbers between -9,007,199,254,740,992 and 9,007,199,254,740,992 have a precision of 1 or higher; numbers in this range can be represented as integers.
- Numbers can be fractional; the closer the number is to zero, the more precision the fraction has.
- The type of floating point number Lunacy uses is binary. This means that fractions which are not a power of two are imprecise in Lua.

Let's show an issue with these floats by looking at a loop in Lua.

To do a simple loop in Lua, we use this form:

```
for counter = start, end, increment do
  print(counter)
end
```

The default value for increment is `1`.

For example:

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

Which gives us 1,2,3,4,5,6,7,8,9, and finally 10.

Likewise:

```
for counter = 1,11,2 do
  print(counter)
end
```

Will give us 1,3,5,7,9,11

We can count backwards:

```
for counter = 10,1,-1 do
  print(counter)
end
```

This gives us 10,9,8,7,6,5,4,3,2,1

Note that we must have the increment as a negative number to count backwards.

Note that, since Lunacy uses 64-bit IEEE floats, counters really should only use integers. This can give unexpected behavior:

```
for counter = .9,1,.05 do
  print(counter)
end
```

With infinite precision numbers, or heck with decimal floating point numbers, we would get 0.90, 0.95, 1.00, but instead we get 0.90 and 0.95 without the 1.00 (when running as 32-bit x86 code; the behavior is as expected on ARM64 and x86_64). This is caused because .05 is a number ever so slightly higher when represented as a 64-bit IEEE binary floating point number, causing the loop to act in unexpected ways.

The solution to avoid these kinds of issues is to always use integers for Lua loops, then divide the number as needed, e.g.:

```
for counter = 90,100,5 do
  print(counter / 100)
end
```

This will give the expected behavior.

How can we see a large integer in Lunacy?

The simple `print()` command will, with Lunacy, print numbers as high as 99,999,999,999,999. Higher numbers show the number of as "e+XXX" format, e.g. 100,000,000,000,000 is shown as `1e+014`.

```
print(99999999999999)
print(100000000000000)
```

This shows `99999999999999` then `1e+014`

In a 64-bit compile of Lunacy, this will correctly show `100000000000000`:

```
print(string.format("%d",100000000000000))
```

However, in a 32-bit compile of Lunacy, the above code incorrectly shows `-2147483648`. To work around this issue requires a function (see the next page):

```
function numberToString(n)
  out = ""
  isNeg = false
  if(n < 0) then
    n = -n
    isNeg = true
  end
  if(n == 0) then
    return "0"
  end
  while n > 0 do
    out = string.format("%d",n%10) .. out
    n = n - (n % 10)
    n = n / 10
  end
  if isNeg then
    out = "-" .. out
  end
  return out
end
print(numberToString(100000000000000))
```

This will correctly show `100000000000000` in both the 32-bit and 64-bit versions of Lunacy.

Note that this function only works correctly for integers, and only for numbers small enough to correctly divide by 10.

COMMENTS

Lunacy has support for comments. Comments are blocks of code ignored by the Lunacy parser. Unlike traditional UNIX scripts, Lua uses two dashes to indicate the beginning of the comment.

Here is an example comment:

```
-- Here is a comment. It ends at the end of the line
foo = 1
-- Another comment
```

A comment which starts with two dashes ends at the end of a line.

Multi-line comments

Lua also has support for multi-line comments. A multi line comment starts with two dashes, immediately followed by two left square brackets. A multi-line comment ends with two right square brackets.

Here is an example multi-line comment:

```
foo = 1
--[[ Here is a multi-line comment.  It ends with two right square
     brackets, like this: ]]
bar = 2
```

It's also possible to use square bracket comment notation to put a comment in the middle of a line:

```
foo --[[ Here is a mid-line comment ]] = 1
```

In case we want `]]` in a comment, we can use this syntax:

```
foo = 1
--[=[ Here is another multi-line comment.  Since this comment has an equal
      sign between the brackets, ]] no longer ends a comment.  Instead,
      the comment ends with an equals sign between the brackets: ]=]
bar = 2
```

If both `]]` and `]=]` are desired in a comment, we can use `--[==[` or `--[===[` or `--[====[` to begin a comment; the comment is ended by the corresponding number of equals signs between the brackets closing the comment (e.g. `]==]` or `]===]` or `]====]`):

```
foo = 1
--[===[ Comment with three equals between the brackets
  ]===]
bar = 2
```

Strings

Lunacy has support for strings. For example:

```
a = "Hello, world!"
print(a)
```

This will output "`Hello, world!`".

Single quotes

Strings can be quoted with either single quotes or double quotes; here's single quotes:

```
a = 'Hello, world!'
print(a)
```

The only difference between single and double quotes is that it changes whether a single quote or a double quote needs to be escaped; unlike other languages, Lunacy does not by default expand variables in strings.

Strings are binary

A string is a binary stream; unlike C's default string type, the NUL (0 value) character can be part of a string. Lunacy does not have UTF-8 or any other locale support; high bit strings are stored as is and strings do not have locale information (i.e. what character encoding a given string uses or even what type of line feed is used) attached to them.

Escaping characters

It's possible to escape characters with strings:

- `\"` always returns a double quote without terminating a string. This escape sequence is not needed if the string is single quoted; e.g. `'" is a double quote'` is equivalent to the string `"\" is a double quote"` or `'\" is a double quote'`. In all cases, the resulting string is `" is a double quote`
- `\'` likewise always returns a single quote.
- `\\` returns a backslash
- `\` followed by a newline allows a multi-line string; the `\` is not part of the resulting string.
- `\0`, `\00`, or `\000` represents a NUL character
- Likewise *nnn*, where *nnn* is a one digit, two digit, or three digit decimal number between 0 and 255 represents a character with the corresponding ASCII (or high bit) numerical value. E.g. `\33` or `\033` is an exclamation point ("bang") character and is equivalent to `!`
- `\r`, `\n`, `\a`, `\b`, `\f`, `\t`, and `\v` generate the corresponding character (In order shown here: carriage return, newline, bell, backspace, form feed, tab, and vertical tab).

String concatenation

It's possible to concatenate strings using the `..` operator. For example:

```
a = "Hello, "
b = a .. "world!"
print(b)
```

returns `Hello, world!`

String manipulation functions

Let's look at some of Lunacy's string manipulation methods. String methods can be called either as `string.`*functionName* (the first argument is the string to run the method against) or as *string*`:`*functionName* (where the string name is already given); e.g. given that `a` is a string, `string.sub(a,1,3)` is equivalent to `a:sub(1,3)`. Note that strings in Lunacy are immutable and the functions which generate strings return newly created strings.

string.sub

`string.sub` returns a substring. Its arguments are `string.sub(`*string, start, end*), where *start* is the first character of the resulting string we put in the resulting substring (since Lua tends to be 1-indexed, 1 is the first character in the source string, 2 is the second character, and so on) and *end* is the final character. If *start* or *end* are negative, we count characters from the end of the string; e.g. `-1` is the final character in the string, `-2` is the second final character, and so on. If *end* is not specified, the resulting substring ends with the source string ends.

Examples:

```
a = "1234567890"
print(a:sub(2,5)) -- Output is 2345
print(string.sub(a,3,8)) -- Output is 345678
print(a:sub(8)) -- Output is 890
print(a:sub(7,-2)) -- Output is 789
print(string.sub(a,-5,8)) -- Output is 678
```

string.len

`string.len` returns how long a string is. A string can be 0 or more bytes long. Examples:

```
a = "12345"
print(a:len()) -- Returns 5
a = ""
print(a:len()) -- Returns 0
```

string.find

`string.find` looks for a given substring in a string. Its form is `string.find(`*string*, *pattern*, *start*, *plain*`)`. *start* and *plain* are optional. *pattern* is the substring we are looking for; by default *pattern* is a Lunacy regular expression pattern (a regular expression allows various different strings with certain characteristics to match; regular expressions will be discussed elsewhere). *start* is the first character we look at in the string for the matching pattern; if not specified, *start* has a value of `1` which is the start of a string. *plain*, if set to `true`, looks for the literal string in *pattern* instead of performing regular expression matching.

If the pattern is found, `string.find` returns two numbers: Where the matching string starts (as a numeric string index; e.g. `1` is the first character), followed by where the matching string ends. If the pattern is not found, `string.find` returns `nil`.

For example, `string.find(a, "345")` will look for an occurrence of the substring `345` in the string `a`. Some other examples:

```
a = "1234567890"
print(a:find("567")) -- Returns 5,7
print(a:find("111")) -- Returns nil
a = "LoveLoveLove"
print(a:find("Love")) -- Returns 1,4
print(a:find("Love",3)) -- Returns 5,8
print(a:find("Love",7,true)) -- Returns 9,12
print(a:find("Love",11)) -- Returns nil
```

string.gsub

`string.gsub` replaces occurrences of a pattern with another string. It is called as `string.gsub(`*string*`, `*pattern*`, `*newString*`, `*count*`)` or *string*`:gsub(`*pattern*, *newString*, *count*`)`

string is the string we are creating an altered copy of. *pattern* is the regular expression pattern we are looking for (again, regular expressions will be discussed elsewhere). *newString* is the string we replace occurrences of *pattern* with. *count*, which is an optional argument, specifies the maximum number of times we replace *pattern*; if *count* is not specified, we replace all occurrences of *pattern* in *string*.

`string.gsub` returns the altered string as well as the number of times it found *pattern*.

Some examples:

```
a = "1234567890"
print(a:gsub("456","axy")) -- Returns "123axy7890",1
print(a:gsub("111","axy")) -- Returns "1234567890",0
a = "LifeLifeLife"
print(a:gsub("Life","Love")) -- Returns "LoveLoveLove",3
print(a:gsub("Life","Love",1)) -- Returns "LoveLifeLife",1
print(a:gsub("Life","Love",2)) -- Returns "LoveLoveLife",2
```

Binary Numbers

Numbers in Lunacy are *binary numbers*, which mean they are internally represented as numbers with only 0s and 1s in them. The number 0 is represented as `0`, the number 1 is `1`, but the number 2 is `10`, and the number 3 is `11`, 4 is `100`, and so on (to be pedantic, the representation for numbers in Lunacy are actually IEEE 754 64-bit floating point numbers which have a different representation in binary, but for the purposes of this lesson we can treat the numbers as if they are binary integers). Each digit in a binary number is called a *bit*.

Hexadecimal

While it's not possible in Lunacy to directly represent numbers in binary, we can represent numbers as hexadecimal (sometimes called hex) numbers. This is a base 16 notation and we tell Lunacy the number is hexadecimal by prefixing it with `0x` so `0x9 + 0x1` is `0xA` or equivalently 10. Let's look at numbers in decimal (normal numbers as taught in elementary school math), binary (base 2, as discussed above), and hexadecimal.

Decimal	`0`	`1`	`2`	`3`	`4`	`5`	`6`	`7`
Binary	`0000`	`0001`	`0010`	`0011`	`0100`	`0101`	`0110`	`0111`
Hexadecimal	`0x0`	`0x1`	`0x2`	`0x3`	`0x4`	`0x5`	`0x6`	`0x7`

Decimal	`8`	`9`	`10`	`11`	`12`	`13`	`14`	`15`
Binary	`1000`	`1001`	`1010`	`1011`	`1100`	`1101`	`1110`	`1111`
Hexadecimal	`0x8`	`0x9`	`0xA`	`0xB`	`0xC`	`0xD`	`0xE`	`0xF`

Note forms like `0xa` or `0xc` are also acceptable; either upper case or lower case or even mixed case (e.g. `0xfF`) is acceptable.

Viewing numbers in hexadecimal can be done with the `string.format` call as follows:

```
function viewAsHex(n)
  return string.format("%x",n)
end
print(viewAsHex(0xBeadBabe))
```

If we wish to view the hexadecimal number in upper case, use `%X` for the string format:

```
function viewAsHexUpper(n)
  return string.format("%X",n)
end
print(viewAsHexUpper(0xBeadBabe))
```

32-bit numbers

There are certain binary operations Lunacy can perform on the numbers. Since the float point format Lunacy uses to represent numbers allows numbers with up to 53 bits of precision, we can not use 64-bit binary integers in Lunacy. 32-bit is the largest common representation of

binary numbers smaller than the 53 bits Lunacy can actually store as a number, so the functions which this book is about to show representing binary math are done with 32-bit numbers. `bit32.bor(0,foo)` will return the value of `foo` as long as `foo` can fit in 32 bits. That in mind:

```
print(string.format("%x",bit32.bor(4294967295,0)))
```

returns `ffffffff`, the largest number we can represent with 32 bits, but

```
print(string.format("%x",bit32.bor(4294967297,0)))
```

returns `1` (two more than the largest number one can fit in 32 bits; its hex form would be `0x100000001` but 32-bit Lunacy can only handle numbers inputted as hex between `0x00000000` and `0xffffffff`) because the `bit32` operations work on return the lowest 32 bits of the input number.

Binary operations

While Lua 5.1 does not have direct support for bitwise binary operations, Lunacy (as well as other common ports of Lua 5.1 such as LuaJIT, Luau used in Roblox, and the version of Lua 5.1 used by the Wikipedia) includes a library of functions which allow binary operations. The number of this library in Lunacy is `bit32`.

Common binary operations are *and*, called `bit32.band` in Lunacy, *or* called `bit32.bor`, and *exclusive or* called `bit32.bxor`. These operations work on each bit independently. Let's look at the four possible one bit results for these operations, where the input is `0` or `1`

Operation	Result
`bit32.band(0,0)`	`0`
`bit32.band(0,1)`	`0`
`bit32.band(1,0)`	`0`
`bit32.band(1,1)`	`1`
`bit32.bor(0,0)`	`0`
`bit32.bor(0,1)`	`1`
`bit32.bor(1,0)`	`1`
`bit32.bor(1,1)`	`1`
`bit32.bxor(0,0)`	`0`
`bit32.bxor(0,1)`	`1`
`bit32.bxor(1,0)`	`1`
`bit32.bxor(1,1)`	`0`

As mentioned before, these operations work on each bit independently. So, if the two inputs have 3 bits, we perform the relevant operation on the lowest bit of the input numbers and have the determine the lowest bit in the output number, then perform the relevant operation on the second lowest bit of the input numbers to determine the second lowest bit in the output number, and finally perform the operation on the third lowest bit of the input numbers which affects the third lowest bit in the output number.

So, for example, to exclusive or the binary numbers `100` (four) and `110` (six), we do the

following steps:

1. The highest (leftmost) bits in the input numbers are `1` and `1`. The exclusive or of `1` and `1` is `0`. So the leftmost bit of the output is `0`.
2. The second (middle) bits in the input numbers are `0` and `1`. The exclusive or of `0` and `1` is `1`, so the middle bit in the output is `1`.
3. The lowest (rightmost) bits in the input numbers are `0` and `0`. The exclusive or of `0` and `0` is `0`, so the lowest bit in the output is `0`.

Let's look at this another way:

```
100    input one
110    input two
010    result
```

Since binary `100` is `4` in decimal and binary `110` is `6` in decimal, the result is `2` (`010`).

And, indeed:

```
print(bit32.bxor(4,6))
```

Returns `2`.

Other binary operations

Another binary operation is `bit32.lshift(n,x)`, which shifts a number `n` to the left `x` number of bits. For example, shifting the binary number `110` one bit to the left yields `1100`; shifting `101` two bits to the left results in `10100`. A `lshift` of one bit is equivalent of multiplying a number by 2; a two bit shift is like multiplying a number by 4; a three bit shift like multiplying by 8; and so on.

The other binary operation in the example xoshiro code is `bit32.rrotate(n,x)`, which rotates a number `n` by `x` bits to the right. Since this is an operation on 32-bit numbers, this means that if the low (rightmost) bit is `1`, the high (32nd from left) bit becomes `1` when rotating one bit to the right.

One example of `bit32.rrotate`:

```
Binary form of n          00000010 00000111 10100101 10000001 In hex: 0x0207a581
bit32.rrotate(n,2)        01000000 10000001 11101001 01100000 In hex: 0x4081e960
```

Here, the two low bits of the input become the two high bits in the output, and the rest of the bits move to the right by two spots.

A real world example

One real-world algorithm which generates *pseudo-random numbers* (a pseudo-random number is a number which looks random, but is actually generated in a non-random way) using 32-bit number bitwise math is *xoshiro128***. Here is an implementation of the algorithm in Lua:

```
xoshiro128 = {}
xoshiro128.__index = xoshiro128
function xoshiro128:init()
  out = {}
  setmetatable(out, xoshiro128) -- Instance of xoshiro128 class
  out.s1 = 1
  out.s2 = 2
  out.s3 = 3
  out.s0 = 4
  return out
end
function xoshiro128:next()
  local result = self.s1 * 5
  result = result % 4294967296 -- Keep number 32-bit
  result = bit32.rrotate(result,25) -- 7-bit left shift in reference code
  result = result * 9
  result = result % 4294967296
  local t = bit32.lshift(self.s1, 9)
  self.s2 = bit32.bxor(self.s2,self.s0)
  self.s3 = bit32.bxor(self.s3,self.s1)
  self.s1 = bit32.bxor(self.s1,self.s2)
  self.s0 = bit32.bxor(self.s0,self.s3)
  self.s2 = bit32.bxor(self.s2,t)
  self.s3 = bit32.rrotate(self.s3,21)
  return result
end
-- This seeding is not part of the xoshiro128** spec
function xoshiro128:seed(s)
  s = s % 4294967296
  self.s0 = bit32.bxor(s,0x55555555)
  self.s1 = (s * 37 + s * 3 + 1009) % 4294967296
  self.s2 = (s * 223 + s * 7 + 1229) % 4294967296
  self.s3 = (s * 947 + bit32.bxor(s,0xAAAAAAAA)) % 4294967296
  return true
end
x = xoshiro128:init()
x:seed(12345)
for a=1,8 do
  print(string.format("%08x",x:next()))
end
```

This code uses classes, which are discussed in the chapter on metatables.

While stock Lua 5.1 uses a low-quality pseudo random number generator for `math.random()` and the above code generates higher quality numbers, Lunacy's `math.random()` generates high quality random numbers using RadioGatún[32].

Tables

Let's look at Lunacy's tables:

```
foo = {}
foo["bar"] = "Hello, "
foo["baz"] = "world!"
foo["bozzle"] = function() print("Hello, world!") end
```

Then, we can do stuff like this:

```
print(foo["bar"])
print(foo["baz"])
print(foo["bozzle"])
```

Since `foo["bozzle"]` is a function (table members can be numbers, strings, functions, tables, booleans, among other data types, but not `nil`), we can call it directly:

```
foo["bozzle"]()
```

or even

```
print(foo["bozzle"]())
```

We can define the above table like this in Lua:

```
foo = {
   bar = "Hello, ",
   baz = "world!",
   bozzle = function() print("Hello, world!") end
}
```

Tables, as noted above, can have members which are themselves tables:

```
foo = {}
foo["bar"] = { oink = "Hello, world!",
               mister = function() print("Hello, world!") return 1 end
             }
foo["baz"] = "Cool!"

print(foo["bar"]["oink"])
print(foo["bar"]["mister"]())
```

The brackets and quotes can get unwieldy, so this also works:

```
print(foo.bar.oink)
print(foo.bar.mister())
```

We can combine brackets and dot notation:

```
print(foo["bar"].oink)
```

Which is userful when using a variable to determine the table index:

```
bozzle = "bar"
print(foo[bozzle].oink)
```

We can even have a sub-table refer to the parent table:

```
foo.bar.loop = foo
```

And then:

```
print(foo.bar.loop.bar.oink)
print(foo.bar.loop.bar.loop.bar.oink)
print(foo.bar.loop.bar.loop.bar.loop.bar.oink)
```

To see the members of a table, we have to iterate through them:

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

Which gives us output like this:

```
bar     table: 00F49F08
baz     Cool!
```

We can use recursion to look at subtables, but be careful of loops!

```
function tableView(t, name, depth)
  -- If we call a function with too few arguments, the extra parameters
  -- have the value nil
  if name == nil then
    name = ""
  end
  if depth == nil then
    depth = 0
  end

  -- Infinite loop protection
  if depth > 10 then
    return 0
  end

  -- View this table and recurse with subtables
  for k,v in pairs(t) do
    if type(v) ~= "table" then -- ~= is != in most other languages
      print(name,k,v)
    else
      tableView(v, name .. " " .. k, depth + 1)
    end
  end
end
```

Note that Lua uses "`--`" to start a comment, unlike the "`#`" most UNIX scripting languages use.

We can now invoke it:

```
tableView(foo)
```

Note the function takes three arguments; since we call the function with only one argument, the other two values are set by Lua to be `nil`.

While the above works, the issue is that when we have an element which loops back to the top of the table, we will show that element multiple times. There *is* a way to solve that problem, but it's a little tricky to understand.

The trick is this: Table keys can *also* be tables (well, pointers to tables; Lua internally stores the memory address of the table as the key and a note the pointer is to a table).

Let's check this out:

```
foo[foo] = "Table pointer key example"
```

Here, the *key* is a pointer to a table. This can be useful, because we can use that to know right away if we're going in to a loop with `tableView`:

```
function tableView(t, name, seen)
  -- If we call a function with too few arguments, the extra parameters
  -- have the value nil
  if name == nil then
    name = ""
  end
  if seen == nil then
    seen = {t = true}
  end

  seen[t] = true

  -- View this table and recurse with subtables
  for k,v in pairs(t) do
    if type(v) ~= "table" then -- ~= is != in most other languages
      print(name,k,v)
    else
      if seen[v] then
        print(name, k, "LOOP!")
      else
        tableView(v, name .. " " .. k, seen)
      end
    end
  end
end
```

METATABLES

Let's make two tables in Lua:

```
foo = { bar = "Hello", baz = "World" }
someTable = { life = "is", good = "yes" }
print(someTable.bar)
```

This returns "`nil`". When we look for `someTable.bar` (i.e. `someTable["bar"]`), we will get `nil` because the element "`bar`" is not in `someTable`, even though it's in the table `foo`.

Now, let's have it so that, if an element is not in `someTable`, we look in the table `foo` for that element:

```
someTableMeta = {__index = foo}
setmetatable(someTable, someTableMeta)
print(someTable.bar)
```

Now, looking up `someTable.bar` returns the value "`Hello`":

1. We look in `someTable` for the element "`bar`"
2. Since it's not found, we see if we have an index metatable. We do, so we look in the index metatable, in this case the table `foo`, for the element.
3. Since `foo` has the element `foo.bar`, we use the element in the index metatable for `someTable` to have it so asking for `someTable.bar` gives us "`Hello`".

If we don't want a separate metatable, a table can be its own metatable:

```
foo = { bar = "Hello", baz = "World" }
someTable = { life = "is", good = "yes", __index = foo }
setmetatable(someTable, someTable)
print(someTable.bar)
```

`someTable.bar` is now "`Hello`", since `foo.bar` is "`Hello`" and `someTable`'s index metatable is the `foo` table.

Likewise, the index metatable can also be the metatable:

```
foo = { bar = "Hello", baz = "World" }
foo.__index = foo
someTable = { life = "is", good = "yes" }
setmetatable(someTable, foo)
print(someTable.bar)
```

Again, this returns "`Hello`".

Note that the "`__index`" element needs to be set *after* the table is initially declared.

Index metatables can also be functions:

```
someTable = { life = "is", good = "yes",
              __index = function() return 404 end }
setmetatable(someTable, someTable)
print(someTable.bar)
```

This returns the number `404`. Here, any element which is not in the table will return, instead of `nil`, the number `404`.

We can use a function to have the same behavior we have when an index metatable is a table:

```
foo = { bar = "Hello", baz = "World" }
someTable = { life = "is", good = "yes",
              __index = function(self, i) return foo[i] end }
setmetatable(someTable, someTable)
print(someTable.bar)
```

This will return "`Hello`". The second argument to an index metatable function is the element we tried to access in the table.

The first argument, `self`, is a reference to the table calling the function, e.g. this will return "`yes`" for any element not otherwise in `someTable`:

```
someTable = { life = "is", good = "yes",
              __index = function(self, i) return self.good end }
setmetatable(someTable, someTable)
print(someTable.bar)
```

This returns "`yes`".

All of this in mind, we can use metatables to have classes:

```
simpleClass = {}
simpleClass.__index = simpleClass
function simpleClass.init(self, x, y)
  self.x = x
  self.y = y
end
function simpleClass.add(self, x, y)
  self.x = self.x + x
  self.y = self.y + y
end
function simpleClass.values(self)
  return self.x, self.y
end
```

Now that we have created a simple class, we can make instances of that class via metatables. Give the above class, let's make an instance of it:

```
objectInstance = {}
setmetatable(objectInstance, simpleClass)
objectInstance.init(objectInstance, 1, 2)
objectInstance.add(objectInstance, 3, 3)
print(objectInstance.values(objectInstance))
```

This will return `4` then `5`.

This works but the grammar is a bit clunky. Lua has some synatic sugar to make things look a little nicer. This is equivalent to the above example class code:

```
simpleClass = {}
simpleClass.__index = simpleClass
function simpleClass:init(x, y)
  self.x = x
  self.y = y
end
function simpleClass:add(x, y)
  self.x = self.x + x
  self.y = self.y + y
end
function simpleClass:values()
  return self.x, self.y
end
objectInstance = {}
setmetatable(objectInstance, simpleClass)
objectInstance:init(1,2)
objectInstance:add(3,3)
print(objectInstance:values())
```

This also returns `4` then `5`.

"`self`" is a magic word in Lua used with Lua's colon syntax.

Let's change the class so that, for addition, we have a method which returns a new object which is the sum of two other objects:

```
simpleClass = {}
simpleClass.__index = simpleClass
function simpleClass:init(x, y)
  self.x = x
  self.y = y
end
function simpleClass:add(other)
  newObject = {}
  setmetatable(newObject, simpleClass)
  newObject.x = self.x + other.x
  newObject.y = self.y + other.y
  return newObject
end
function simpleClass:values()
  return self.x, self.y
end
```

```
objectInstance = {}
setmetatable(objectInstance, simpleClass)
anotherObject = {}
setmetatable(anotherObject, simpleClass)
objectInstance:init(1,2)
anotherObject:init(3,3)
thirdObject = objectInstance:add(anotherObject)
print(thirdObject:values())
```

This again returns `4` then `5`.

This in mind, we can use metatables to define what "`+`" does with a given table:

```
simpleClass = {}
simpleClass.__index = simpleClass
function simpleClass:init(x, y)
  self.x = x
  self.y = y
end
function simpleClass:add(other)
  newObject = {}
  setmetatable(newObject, simpleClass)
  newObject.x = self.x + other.x
  newObject.y = self.y + other.y
  return newObject
end
simpleClass.__add = simpleClass.add
function simpleClass:values()
  return self.x, self.y
end
objectInstance = {}
setmetatable(objectInstance, simpleClass)
anotherObject = {}
setmetatable(anotherObject, simpleClass)
objectInstance:init(1,2)
anotherObject:init(3,3)
thirdObject = objectInstance + anotherObject
print(thirdObject:values())
```

Like the other examples, this returns `4` then `5`.

As we can see, Lua has operator overloading.

We can make it a little easier to make new objects. In addition, the `__add` metatable can be a function instead of a pointer to a function (see the next page):

```
simpleClass = {}
simpleClass.__index = simpleClass
function simpleClass:init(x, y)
  out = {}
  setmetatable(out, simpleClass)
  out.x = x
  out.y = y
  return out
end
function simpleClass:__add(other)
  newObject = {}
  setmetatable(newObject, simpleClass)
  newObject.x = self.x + other.x
  newObject.y = self.y + other.y
  return newObject
end
function simpleClass:values()
  return self.x, self.y
end
objectInstance = simpleClass:init(1, 2)
anotherObject = simpleClass:init(3, 3)
thirdObject = objectInstance + anotherObject
print(thirdObject:values())
```

This also returns `4` then `5`.

Note that while the `init` method gets a "`self`" argument, we ignore that so that we can retain the colon syntax for object methods.

Loops (While and Repeat)

In this chapter we will look at `while` loops, `repeat` loops, the `break` keyword, working around Lunacy's lack of a "continue" keyword, and show a basic `for` loop.

The While loop

In a `while` loop, we go through a loop while *condition* is true. A `while` loop starts with the `do` keyword and ends with the `end` keyword. The general form is:

```
while condition do
  actions
end
```

When Lunacy hits the `end` keyword, we see if *condition* is true; if so, Lunacy goes back to the `do` keyword which begins the `while` loop.

Let's this look at this code:

```
a = 1
while a < 10 do
  print(a)
  a = a + 1
end
```

The output is 1, 2, 3, 4, 5, 6, 7, 8, and 9.

While loops can nest, which means we can have one while loop in another while loop:

```
a = 1
while a < 4 do
  b = 1
  while b < 3 do
    print(a * b)
    b = b + 1
  end
  a = a + 1
end
```

Output is 1, 2, 2, 4, 3, 6.

Note that it's possible to have a `while` loop which never runs, e.g.:

```
a = 10
while a < 4 do
  print("This will never run")
end
print("We are here")
```

This will only output `We are here`.

The repeat loop

A `repeat` loop is similar to a `while` loop, but it will always goes through the loop once. For example:

```
a = nil
repeat
  if a == nil then
    a = 1
  end
  print(a)
  a = a + 1
until a > 9
print("Hello")
```

This outputs 1,2,3,4,5,6,7,8,9, then the word `Hello`.

Note that a `repeat` loop, unlike other loops, does not terminate with the `end` keyword but instead with an `until` statement.

The break keyword

It's possible to break a `for`, `while`, or `repeat` loop with the `break` keyword. For example:

```
a = 1
while true do
  if a >= 10 then
    break
  end
  print(a)
  a = a + 1
end
```

This will output 1,2,3,4,5,6,7,8, and 9.

With a `repeat` loop:

```
a = 1
repeat
  if a >= 10 then
    break
  end
  print(a)
  a = a + 1
until false
```

This will also output 1,2,3,4,5,6,7,8, and 9.

No continue keyword

Some other languages, such as C, have a "continue" keyword. This keyword would cause the loop to go back to the beginning of the loop, e.g. the `while` keyword if we're in a `while` loop.

Lunacy does not have a continue keyword, but we can simulate it by having a loop inside of a loop:

```
doBreak = false
a = 1
repeat
  while a < 10 do
    doBreak = true
    a = a + 1
    if a % 2 == 0 then
      doBreak = false -- Make break act like continue
      break -- Acts like continue
    end
    if a == 9 then
      break -- Acts like break because doBreak is true
    end
    print(a)
  end
until doBreak
```

This will output 3,5, and 7.

This trick works for `while` and `repeat` loops, but not with `for` loops (a version with the `repeat/until` inside the `for` loop would work; this is left as an exercise for the reader).

The for loop

Lunacy actually has two forms of a `for` loop. One is a simple for loop where we go through numbers; the other, which we will look at elsewhere, uses a function along with `for`. Here is a simple numerical `for` loop:

```
for a=1,10,2 do
  print(a)
end
```

This will output 1,3,5,7, and 9.

The "`1`" above is the value `a` has when we run the loop for the first time; `10` is the maximum value `a` can have and have the loop still run. `2` above is the amount we increment `a` by (or decrement, should this third number be negative) after each iteration of the loop.

Note that it's best in a `for` loop of this form to use only integers; see the chapter on numbers for discussion of why.

FOR LOOPS WITH FUNCTIONS

It's possible to use a function with a `for` loop. E.g:

```
a = {z = 1, y = 2}
for k,v in pairs(a) do
  print(k,v)
end
```

This will output in varying order (z,1) and (y,2).

We can make our own function which can be used as the iterator in a `for` loop:

```
function range(low, high)
  local index = low
  return function()
    local rvalue = index
    index = index + 1
    if(rvalue > high) then
      return nil
    end
    return rvalue
  end
end
```

Here, `range` is a function which returns a function. What does the function generated by `range` do? Let's look:

```
twoToNine = range(2,9)
for a=1,11 do
  print(twoToNine())
end
```

We get this output:

```
2
3
4
5
6
7
8
9
nil
nil
nil
```

Here is what is happening: The function `twoToNine` is able to remember the value of the variable `index` between its invocations. `index` is a variable whose value gets stored in a namespace local to the `twoToNine` function; this ability is called *function closure*.

That in mind, we can use this `range` function in a `for` loop:

```
for a in range(2,9) do
  print(a)
end
```

Given `range` as defined above, the above function will return this:

```
2
3
4
5
6
7
8
9
```

Under the hood, Lunacy calls the `range` function once. The `range` function returns an anonymous function which we will call *instanceOfRange*. This *instanceOfRange* function is then called at the start of the `for` loop. If *instanceOfRange* returns `nil` then the loop terminates; otherwise we go through the loop with the variable `a` set to be the output value of the *instanceOfRange* function. We do this over and over until *instanceOfRange* returns `nil`.

Sorted table keys

Let's make a `for` iterator function which will return a function that returns the key for each element in a table in sorted order:

```
function sortedTableKeys(inputTable)
  local keyList = {}
  local index = 1
  for k,_ in pairs(inputTable) do
    table.insert(keyList,k)
  end
  table.sort(keyList)
  return function()
    rvalue = keyList[index]
    index = index + 1
    return rvalue
  end
end
```

Here, `table.insert` is used to add a key in `inputTable` to `keyList`; it is run over and over again to make `keyList` a list of all of the keys in `inputTable`. Once we have this list as an array, `table.sort` sorts the list for us.

Next, the `sortedTableKeys` function creates a function which will simply return, when called over and over again, the keys to `inputTable` in sorted order. That in mind, let's make a table and compare how the `sortedTableKeys` function runs versus how `pairs` runs (on next page):

```
someTable = {a=12,b=14,c=19,d=23,e=17,f=11}

print("Unsorted:")
for a in pairs(someTable) do
  print(a) -- Letters in random order
end

print("")
print("Sorted:")
for a in sortedTableKeys(someTable) do
  print(a) -- Letters correctly sorted
end
```

This will first output the keys to `someTable` in random order (as an aside, the order changes each time Lunacy is started), then output the keys to `someTable` in sorted order, namely a, b, c, d, e, and f.

We can even have a function which determines how the keys are sorted:

```
function sortedTableKeysSF(inputTable, sortFunction)
  local keyList = {}
  local index = 1
  for k,_ in pairs(inputTable) do
    table.insert(keyList,k)
  end
  table.sort(keyList, sortFunction)
  return function()
    rvalue = keyList[index]
    index = index + 1
    return rvalue
  end
end

function revSort(a, b)
  return a > b
end

someTable = {a=12,b=14,c=19,d=23,e=17,f=11}

for a in sortedTableKeysSF(someTable, revSort) do
  print(a) -- Letters sorted in reverse
end
```

Here, the above code returns f, e, d, c, b, and a: The letters reverse sorted. The default (forwards, i.e a, b, c, d, e, and f) sorting would be done as follows:

```
function normalSort(a, b)
  return a < b
end

for a in sortedTableKeysSF(someTable, normalSort) do
  print(a)
end
```

List iteration

Lua has simple numerical iteration via `for foo=1,10 do` and Lua has function iteration via `for foo in something(bar) do` but Lua does not have built in iterators for a list the way, say, Perl or Python do.

However, we can work around this by making functions which let Lua go through a list.

First, let's have a function that, when given a table, returns its keys as a list:

```
function tableKeys(inputTable)
  local out = {}
  for k,_ in pairs(inputTable) do
    table.insert(out,k)
  end
  return out
end
```

Now, let's have a function that will allow us to use `for` to iterate through a list:

```
function iterateList(inputList)
  local index = 1
  return function()
    local rvalue = inputList[index]
    index = index + 1
    return rvalue
  end
end
```

And another function which sorts a list and returns the sorted list

```
function sorted(inputList)
  table.sort(inputList)
  return inputList
end
```

We can also have a function to reverse a list:

```
function reverse(inputList)
  local out = {}
  for a=#inputList,1,-1 do
    table.insert(out,inputList[a])
  end
  return out
end
```

With these four functions, we can iterate through a table with the keys reversed sorted:

```
someTable = {a=12,b=14,c=19,d=23,e=17,f=11}
for k in iterateList(reverse(sorted(tableKeys(someTable)))) do
  print(k, someTable[k])
end
```

MULTIPLE INHERITANCE

In the chapter on metatables we looked at simple inheritance. We can also have multiple inheritance:

- A class can have child classes
- Child classes can have multiple parents

Children

A given class can have a derived (a.k.a. child) class. Let's look at the xoshiro128** example we used in the chapter on binary numbers, but now with a child class. First, the parent class:

```
xoshiro128 = {}
xoshiro128.__index = xoshiro128

function xoshiro128:init()
  out = {}
  setmetatable(out, self)
  out.s1 = 1
  out.s2 = 2
  out.s3 = 3
  out.s0 = 4
  return out
end

function xoshiro128:next()
  local result = self.s1 * 5
  result = result % 4294967296 -- Keep number 32-bit
  result = bit32.rrotate(result,25) -- 7-bit left shift in reference code
  result = result * 9
  result = result % 4294967296
  local t = bit32.lshift(self.s1, 9)
  self.s2 = bit32.bxor(self.s2,self.s0)
  self.s3 = bit32.bxor(self.s3,self.s1)
  self.s1 = bit32.bxor(self.s1,self.s2)
  self.s0 = bit32.bxor(self.s0,self.s3)
  self.s2 = bit32.bxor(self.s2,t)
  self.s3 = bit32.rrotate(self.s3,21)
  return result
end
```

We can run this:

```
x = xoshiro128:init()
print(string.format("%x",x:next()))
```

Now that we have this class, we can make a derived class which adds the seeding method. We do this because the method to seed a xoshiro128** state is not part of its specification. On the next page is how that code will look:

```
-- Let's make a class which is the child class
-- This seeding is not part of the xoshiro128** spec
xoSeed = {}
xoSeed.__index = xoSeed
setmetatable(xoSeed,xoshiro128)

function xoSeed:seed(s)
  s = s % 4294967296
  self.s0 = bit32.bxor(s,0x55555555)
  self.s1 = (s * 37 + s * 3 + 1009) % 4294967296
  self.s2 = (s * 223 + s * 7 + 1229) % 4294967296
  self.s3 = (s * 947 + bit32.bxor(s,0xAAAAAAAA)) % 4294967296
  return true
end
```

Now that we have a child class, we can run it:

```
x = xoSeed:init()
x:seed(12345)
for a=1,8 do
  print(string.format("%08x",x:next()))
end
```

Let's look at the `init()` method which is used by both the parent and child class again:

```
function xoshiro128:init()
  out = {}
  setmetatable(out, self)
  out.s1 = 1
  out.s2 = 2
  out.s3 = 3
  out.s0 = 4
  return out
end
```

Here, the `setmetatable` function call has it so the class the object is derived from, `self` in the code, is whatever class we are calling the function in. If we're calling this method from the `xoshiro128` class, then `self` will be `xoshiro128`. If we're calling this method from the `xoSeed` class, then `self` is `xoSeed`.

Multiple parents

Since the `__index` member of a metatable can be either a table or a function, we can have multiple parents for a class in Lua. Let's see how that would look. Note that the `xoshiro128` class is the same as it was before:

```
xoshiro128 = {}
xoshiro128.__index = xoshiro128

function xoshiro128:init()
  out = {}
  setmetatable(out, self)
  out.s1 = 1
  out.s2 = 2
  out.s3 = 3
  out.s0 = 4
  return out
end

function xoshiro128:next()
  local result = self.s1 * 5
  result = result % 4294967296 -- Keep number 32-bit
  result = bit32.rrotate(result,25) -- 7-bit left shift in reference code
  result = result * 9
  result = result % 4294967296
  local t = bit32.lshift(self.s1, 9)
  self.s2 = bit32.bxor(self.s2,self.s0)
  self.s3 = bit32.bxor(self.s3,self.s1)
  self.s1 = bit32.bxor(self.s1,self.s2)
  self.s0 = bit32.bxor(self.s0,self.s3)
  self.s2 = bit32.bxor(self.s2,t)
  self.s3 = bit32.rrotate(self.s3,21)
  return result
end

seeder = {}
seeder.__index = seeder
function seeder:seed(s)
  s = s % 4294967296
  self.s0 = bit32.bxor(s,0x55555555)
  self.s1 = (s * 37 + s * 3 + 1009) % 4294967296
  self.s2 = (s * 223 + s * 7 + 1229) % 4294967296
  self.s3 = (s * 947 + bit32.bxor(s,0xAAAAAAAA)) % 4294967296
  return true
end

mix={}
mix.__index = function(self,element)
  if xoshiro128[element] then
    return xoshiro128[element]
  end
  if seeder[element] then
    return seeder[element]
  end
  return nil
end
setmetatable(mix, mix)

x = mix:init()
x:seed(12345)
for a=1,8 do print(string.format("%08x",x:next())) end
```

Regular expressions

In the chapter on strings, we mentioned regular expressions. Let's look at those in more detail.

To review, here's an example of `string.gsub`:

```
a = "Hello for the world."
b = "Life is good. Life is amazing!"
a = a:gsub("Hello","Love")
b = b:gsub("Life","Love")
print(a,b)
```

At the point, the output is "`Love for the world.`" and "`Love is good. Love is amazing!`".

What if we could simply replace the first word in a given sentence with the word `Love`? That is where *regular expressions* come in; they allow a number of different strings to match a given *pattern*. For example, here's the Lunacy regular expression which means "the first word in a sentence":

```
%.%s*%w+
```

In more detail:

- "`%.`" means "the '`.`' character"
- "`%s*`" means "any number, including zero, of spaces, tabs, or other whitespace characters"
- "`%w+`" means "one or more letters and/or numbers"

In particular:

- "`%.`" means "the literal '`.`' character"
- "`%s`" means "any whitespace character (space, tab, etc.)"
- "`*`" means "zero or more instances of the character (or character type) we just saw"
- "`%w`" means "any letter or number"
- "`+`" means "one or more instances of the previous character (or character type)"

So, let's see this in action:

```
b = "Life is good. Life is amazing!"
b = b:gsub("%.%s*%w+","Love")
print(b)
```

We get this:

```
Life is goodLove is amazing!
```

This is a start but there are some problems:

- We do not replace the word at the beginning of the string
- We do not preserve the punctuation starting the sentence

So, let's see if we can account for these. Let's look at this regular expression:

```
^%w+
```

This regular expression means "the first word in the string":

- "^" means "the beginning of the string"
- "%w" means "any letter or number"
- "+" means "one or more instances of the previous character (or character type)"

```
b = "Life is good. Life is amazing!"
b = b:gsub("^%w+","Love")
print(b)
```

Gives us this:

```
Love is good. Life is amazing!
```

So now we need another regular expression which let's us capture words at the beginning of sentences, while preserving the punctuation which begins a sentence:

```
(%.%s*)%w+
```

The part of the regular expression between the parenthesis is noted and "%1" is replaced with the part between the first set of parenthesis in the second argument to gsub, e.g.

```
b = "Life is good. Life is amazing!"
b = b:gsub("(%.%s*)%w+","%1Love")
print(b)
```

Gets this:

```
Life is good. Love is amazing!
```

It would also be nice if we could have more characters than "." matched, because sentences sometimes end with "?" or "!". Let's look at how we can match multiple characters:

```
[%.%?!]
```

- The left square bracket "[" indicates we begin a set of multiple possible characters and/or character types which the regular expression can match against.
- "%." indicates the literal "." character. We need the "%" character because "." otherwise indicates that any character can match that position in a regular expression.
- "%?" likewise matches against the literal "?" character. We need "%" here because "?" matches against zero or one occurrences of the previous character (or character group) in the regular expression.

- "`!`" does not have any special meaning in Lunacy regular expressions. However, should we forget this when formulating regular expression, "`%!`" matches against a literal "`!`" as does just "`!`".
- While not used in the above regular expression, we use "`%%`" to match against "`%`".
- "`[`" ends the set of possible characters we can match against.

Let's see the above regular expression in action:

```
a = "!@*$.?%^&"
a = a:gsub("[%.%?!]","#")
print(a)
```

This outputs:

```
#@*$##%^&
```

As one can see, the "`.`", "`?`", and "`!`" character are all replaced with "`#`".

That in mind, let's replace every word at the beginning of sentences with "`Love`". Let's first look at the Lunacy code which does this:

```
a = "Life is good. Someone is nice! What is amazing? Iron is strong."
a = a:gsub("^%w+","Love") -- Make first word on line "Love"
a = a:gsub("([%.%?!]%s*)%w+","%1Love") -- Words at sentence start
print(a)
```

This outputs:

```
Love is good. Love is nice! Love is amazing? Love is strong.
```