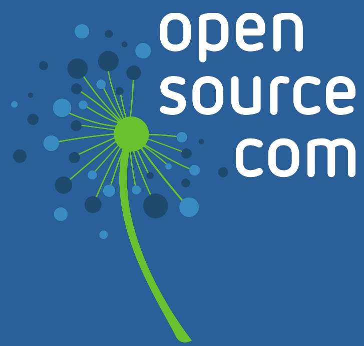


Learn Lua

by Seth Kenlon

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

Supported by
Red Hat

Table of Contents

5 steps to learn any programming language....4
Is Lua worth learning?....12
Learn Lua by writing a "guess the number" game....17
Trick Lua into becoming an object-oriented language....20
How to iterate over tables in Lua....24
Manipulate data in files with Lua....28
Parse arguments with Lua....31
Make Lua development easy with Luarocks....34
Parsing config files with Lua....39
Parsing command options in Lua....45

Seth Kenlon

Seth Kenlon is a UNIX geek, free culture advocate, independent multimedia artist, and tabletop RPG nerd. He has worked in the film and computing industry, often at the same time. He is one of the maintainers of the Slackware-based multimedia production project Slackermedia.

5 steps to learn any programming language

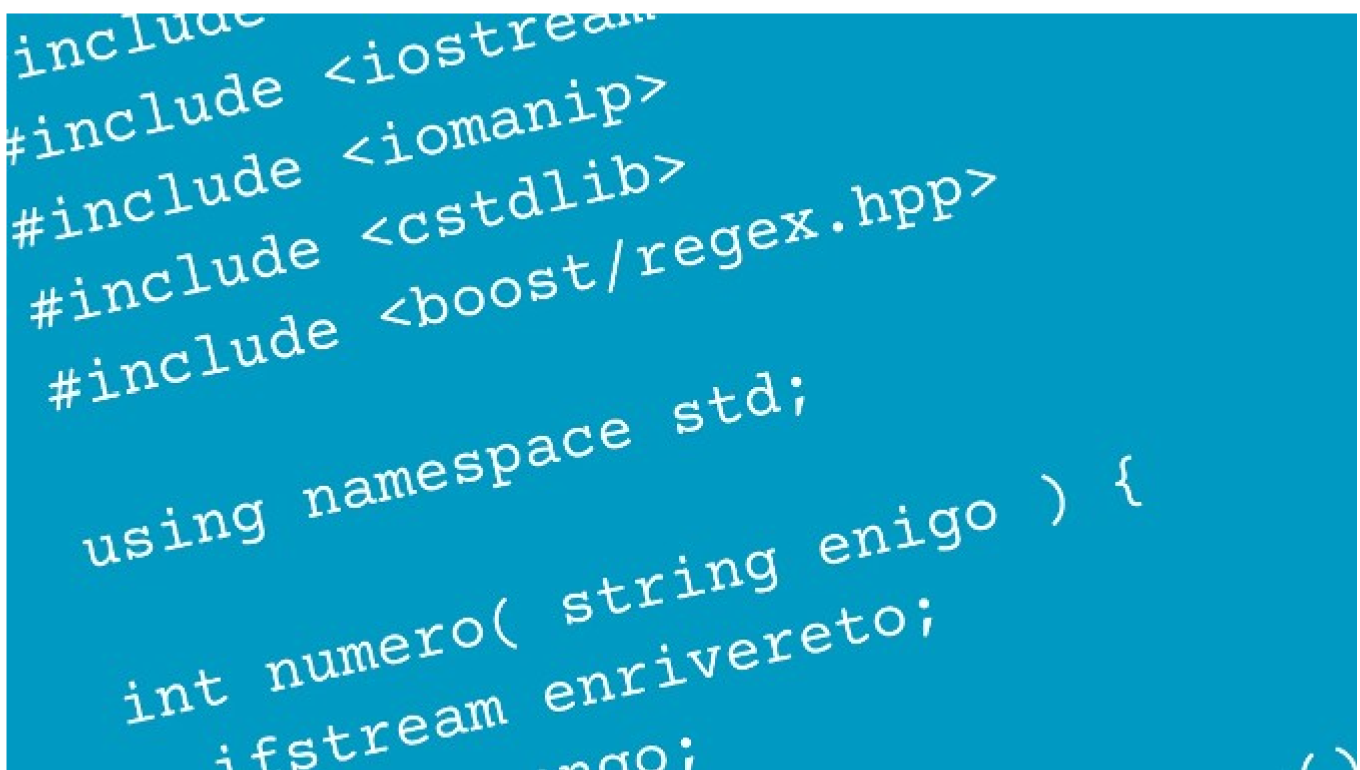
Some people love learning new programming languages. Other people can't imagine having to learn even one. In this article, I'm going to show you how to think like a coder so that you can confidently learn any programming language you want.

The truth is, once you've learned how to program, the language you use becomes less of a hurdle and more of a formality. In fact, that's just one of the many reasons educators say to teach kids to code early. Regardless of how simple their introductory language may be, the logic remains the same across everything else children (or adult learners) are likely to encounter later.

With just a little programming experience, which you can gain from any one of several introductory articles here on Opensource.com, you can go on to learn *any* programming language in just a few days (sometimes less). Now, this isn't magic, and you do have to put some effort into it. And admittedly, it takes a lot longer than just a few days to learn every library available to a language or to learn the nuances of packaging your code for delivery. But getting started is easier than you might think, and the rest comes naturally with practice.

When experienced programmers sit down to learn a new language, they're looking for five things. Once you know those five things, you're ready to start coding.

1. Syntax



(Seth Kenlon, CC BY-SA 4.0)

The syntax of a language describes the structure of code. This encompasses both how the code is written on a line-by-line basis as well as the actual words used to construct code statements.

Python, for instance, is known for using indentation to indicate where one block ends and another one starts:

```
while j < rows:
    while k < columns:
        tile = Tile(k * w)
        board.add(tile)
        k += 1
    j += 1
    k = 0
```

Lua just uses the keyword `end`:

```
for i,obj in ipairs(hit) do
  if obj.moving == 1 then
```

```
    obj.x,obj.y = v.mouse.getPosition()
  end
end
```

Java, C, C++, and similar languages use braces:

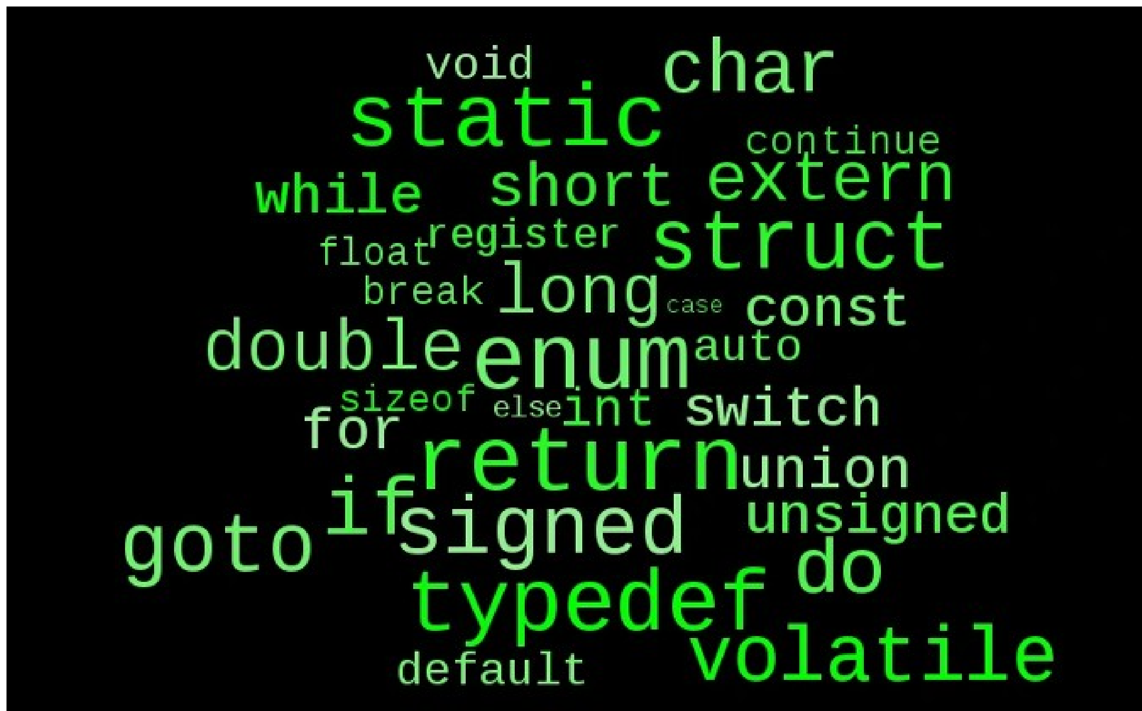
```
while (std::getline(e,r)) {
  wc++;
  }
```

A language's syntax also involves things like including libraries, setting variables, and terminating lines. With practice, you'll learn to recognize syntactical requirements (and conventions) almost subliminally as you read sample code.

Take action

When learning a new programming language, strive to understand its syntax. You don't have to memorize it, just know where to look, should you forget. It also helps to use a good IDE, because many of them alert you of syntax errors as they occur.

2. Built-ins and conditionals



(Seth Kenlon, CC BY-SA 4.0)

A programming language, just like a natural language, has a finite number of words it recognizes as valid. This vocabulary can be expanded with additional libraries, but the core language knows a specific set of keywords. Most languages don't have as many keywords as you probably think. Even in a very low-level language like C, there are only 32 words, such as `for`, `do`, `while`, `int`, `float`, `char`, `break`, and so on.

Knowing these keywords gives you the ability to write basic expressions, the building blocks of a program. Many of the built-in words help you construct conditional statements, which influence the flow of your program. For instance, if you want to write a program that lets you click and drag an icon, then your code must detect when the user's mouse cursor is positioned over an icon. The code that causes the mouse to grab the icon must execute only *if* the mouse cursor is within the same coordinates as the icon's outer edges. That's a classic if/then statement, but different languages can express that differently.

Python uses a combination of `if`, `elif`, and `else` but doesn't explicitly close the statement:

```
if var == 1:
    # action
elif var == 2:
    # some action
else:
    # some other action
```

Bash uses `if`, `elif`, `else`, and uses `fi` to end the statement:

```
if [ "$var" = "foo" ]; then
   # action
elif [ "$var" = "bar" ]; then
   # some action
else
   # some other action
fi
```

C and Java, however, use `if`, `else`, and `else if`, enclosed by braces:

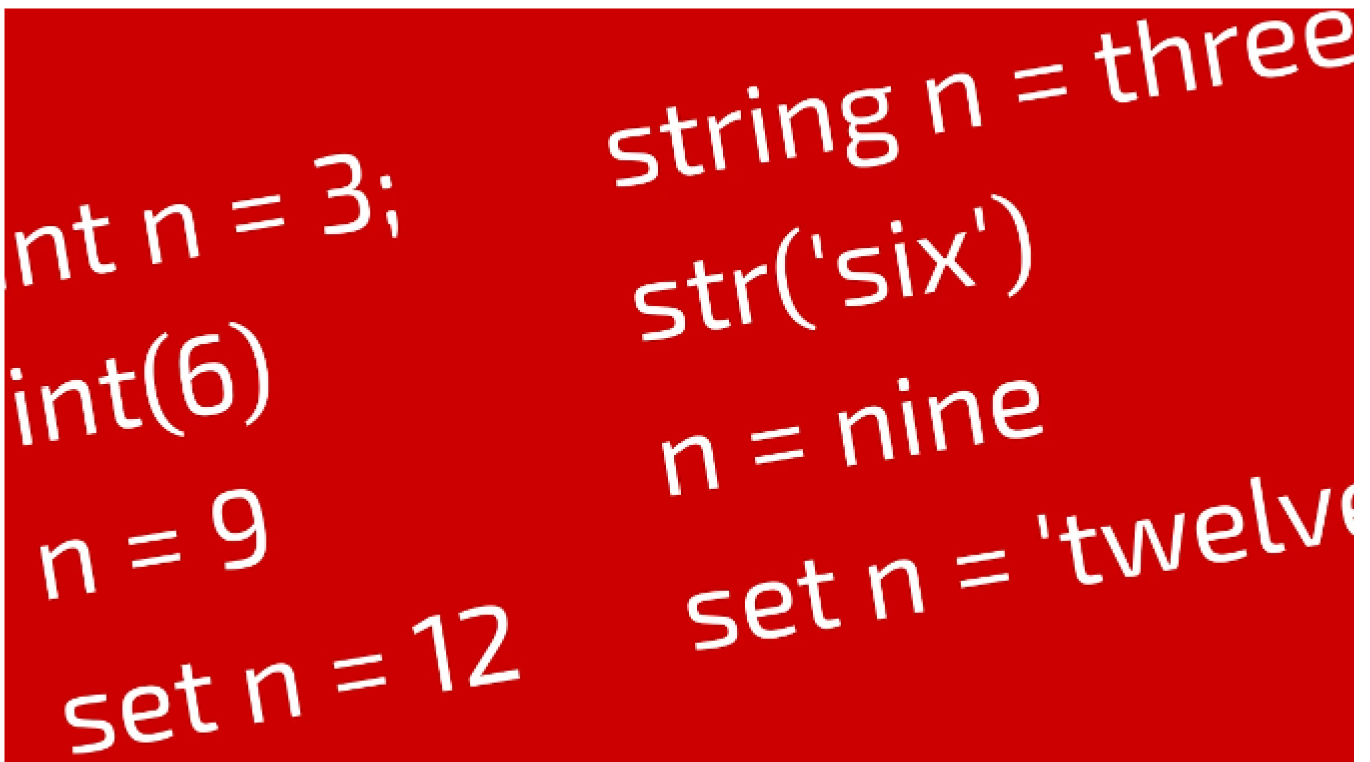
```
if (boolean) {
   // action
} else if (boolean) {
   // some action
} else {
   // some other action
}
```

While there are small variations in word choice and syntax, the basics are always the same. Learn the ways to define conditions in the programming language you're learning, including `if/then`, `do...while`, and `case` statements.

Take action

Get familiar with the core set of keywords a programming language understands. In practice, your code will contain more than just a language's core words, because there are almost certainly libraries containing lots of simple functions to help you do things like print output to the screen or display a window. The logic that drives those libraries, however, starts with a language's built-in keywords.

3. Data types



(Seth Kenlon, CC BY-SA 4.0)

Code deals with data, so you must learn how a programming language recognizes different kinds of data. All languages understand integers and most understand decimals and individual characters (a, b, c, and so on). These are often denoted as `int`, `float` and `double`, and

`char`, but of course, the language's built-in vocabulary informs you of how to refer to these entities.

Sometimes a language has extra data types built into it, and other times complex data types are enabled with libraries. For instance, Python recognizes a string of characters with the keyword `str`, but C code must include the `string.h` header file for string features.

Take action

Libraries can unlock all manner of data types for your code, but learning the basic ones included with a language is a sensible starting point.

4. Operators and parsers

```
ind = mystring.index(' ')

mystring.upper()
mystring.count("t")
mystring[:ind].strip()
```

(Seth Kenlon, CC BY-SA 4.0)

Once you understand the types of data a programming language deals in, you can learn how to analyze that data. Luckily, the discipline of mathematics is pretty stable, so math operators are often the same (or at least very similar) across many languages. For instance, adding two integers is usually done with a + symbol, and testing whether one integer is greater than

another is usually done with the > symbol. Testing for equality is usually done with == (yes, that's two equal symbols, because a single equal symbol is usually reserved to *set* a value).

There are notable exceptions to the obvious in languages like Lisp and Bash, but as with everything else, it's just a matter of mental transliteration. Once you know *how* the expression is different, it's trivial for you to adapt. A quick review of a language's math operators is usually enough to get a feel for how math is done.

You also need to know how to compare and operate on non-numerical data, such as characters and strings. These are often done with a language's core libraries. For instance, Python features the `split()` method, while C requires `string.h` to provide the `strtok()` function.

Take action

Learn the basic functions and keywords for manipulating basic data types, and look for core libraries that help you accomplish complex actions.

5. Functions

```
46
47 public class CamView extends JFrame implements Runnable, Webc
48     WindowListener, UncaughtExceptionHandler, ItemListener,
49     WebcamDiscoveryListener, ActionListener, WebcamImageTrans
50
51     private int counter = 0;
52     private static final long serialVersionUID = 1L;
53     private static BufferedImage onion = null;
54     private Webcam webcam = null;
55     private WebcamPanel panel = null;
56     private WebcamPicker picker = null;
57     private File dir_images;
58
59     @Override
60     public void run() {
61         JMenuBar menu_main = new JMenuBar();
62         //ImageIcon exitIcon = new ImageIcon("src/resources/e
63
64         JMenu menu_main_file = new JMenu("File");
```

(Seth Kenlon, CC BY-SA 4.0)

Code usually isn't just a to-do list for a computer. Typically when you write code, you're looking to present a computer with a set of theoretical conditions and a set of instructions for actions that must be taken when each condition is met. While flow control with conditional

statements and math and logic operators can do a lot, code is a lot more efficient once functions and classes are introduced because they let you define subroutines. For instance, should an application require a confirmation dialogue box very frequently, it's a lot easier to write that box *once* as an instance of a class rather than re-writing it each time you need it to appear throughout your code.

You need to learn how classes and functions are defined in the programming language you're learning. More precisely, first, you need to learn whether classes and functions are available in the programming language. Most modern languages do support functions, but classes are specialized constructs common to object-oriented languages.

Take action

Learn the constructs available in a language that help you write and use code efficiently.

You can learn anything

Learning a programming language is, in itself, a sort of subroutine of the coding process. Once you understand the theory behind how code works, the language you use is just a medium for delivering logic. The process of learning a new language is almost always the same: learn syntax through simple exercises, learn vocabulary so you can build up to performing complex actions, and then practice, practice, practice.

Is Lua worth learning?

Lua is a scripting language used for procedural programming, functional programming, and even object-oriented programming. It uses a C-like syntax, but is dynamically typed, features automatic memory management and garbage collection, and runs by interpreting bytecode with a register-based virtual machine. This makes it a great language for beginners, but also a powerful tool for experienced programmers.

Lua has been somewhat eclipsed from the public view by languages like Python and JavaScript, but Lua has several advantages that make it popular in some major software projects. Lua is easily embedded within other languages, meaning that you can include Lua files in the code base of something written in (for instance) Java and it runs as if it were native Java code. It sounds like magic, but of course there are projects like luaj working to make it possible, and it's only possible because Lua is designed for it. It's partly because of this flexibility that you're likely to find Lua as the scripting language for video games, graphic applications, and more.

As with anything, it takes time to *perfect*, but Lua is easy (and fun) to learn. It's a consistent language, a friendly language with useful error messages, and there's lots of great support online. Ready to get started?

Installing Lua

On Linux, you can install Lua using your distribution's package manager. For instance, on Fedora, CentOS, Mageia, OpenMandriva, and similar distributions:

```
$ sudo dnf install lua
```

On Debian and Debian-based systems:

```
$ sudo apt install lua
```

For Mac, you can use MacPorts or Homebrew.

```
$ sudo port install lua
```

For Windows, install Lua using Chocolatey.

To test Lua in an interactive interpreter, type `lua` in a terminal.

Functions

As with many programming languages, Lua syntax generally involves a built-in function or keyword, followed by an argument. For instance, the `print` function displays any argument you provide to it.

```
$ lua
Lua 5.4.2  Copyright (C) 1994-2020 Lua.org, PUC-Rio
> print('hello')
hello
```

Lua's `string` library can manipulate words (called "strings" in programming.) For instance, to count the letters in a string, you use the `len` function of the `string` library:

```
> string.len('hello')
5
```

Variables

A variable allows you to create a special place in your computer's memory for temporary data. You can create variables in Lua by inventing a name for your variable, and then putting some data into it.

```
> foo = "hello world"
> print(foo)
hello world
> bar = 1+2
> print(bar)
3
```

Skip to content

Tables

Second only to the popularity of variables in programming is the popularity of *arrays*. The word "array" literally means an *arrangement*, and that's all a programming array is. It's a specific arrangement of data, and because there is an arrangement, an array has the advantage of being structured. An array is often used to perform essentially the same purpose as a variable, except that an array can enforce an order to its data. In Lua, an array is called a `table`.

Creating a table is like creating a variable, except that you set its initial content to two braces (`{}`):

```
> mytable = {}
```

When you add data to a table, it's also a lot like creating a variable, except that your variable name always begins with the name of the table, and is separated with a dot:

```
> mytable.foo = "hello world"
> mytable.bar = 1+2
> print(mytable.foo)
hello world
> print(mytable.bar)
3
```

Scripting with Lua

Running Lua in the terminal is great for getting instant feedback, but it's more useful to run Lua as a script. A Lua script is just a text file containing Lua code, which the Lua command can interpret and execute.

The eternal question, when just starting to learn a programming language, is how you're supposed to know what to write. This article has provided a good start, but so far you only know two or three Lua functions. The key, of course, is in documentation. The Lua language isn't that complex, and it's very reasonable to refer to the Lua documentation site for a list of keywords and functions.

Here's a practice problem.

Suppose you want to write a Lua script that counts words in a sentence. As with many programming challenges, there are many ways to go about this, but say the first relevant function you find in the Lua docs is `string.gmatch`, which can search for a specific

character in a string. Words are usually separated by an empty space, so you decide that counting spaces + 1 ought to render a reasonably accurate count of the words they're separating.

Here's the code for that function:

```
function wc(words,delimiter)
  count=1
  for w in string.gmatch(words, delimiter) do
    count = count + 1
  end
 return count
end
```

These are the components of that sample code:

- `function`: A keyword declaring the start of a function. A custom function works basically the same way as built-in functions (like `print` and `string.len`.)
- `words` and `delimiter`: *Arguments* required for the function to run. In the statement `print('hello')`, the word `hello` is an argument.
- `counter`: A variable set to 1.
- `for`: A loop using the `string.gmatch` function as it iterates over the `words` you've input into the function, and searches for the `delimiter` you've input.
- `count = count +1`: For each `delimiter` found, the value of `count` is re-set to its *current value* plus 1.
- `end`: A keyword ending the `for` loop.
- `return count`: This function outputs (or *returns*) the contents of the `count` variable.
- `end`: A keyword ending the function.

Now that you've created a function all your own, you can use it. That's an important thing to remember about a function. It doesn't run on its own. It waits for you to call it in your code.

Type this sample code into a text file and save it as `words.lua`:

```
function wc(words,delimiter)
  count=1
  for w in string.gmatch(words, delimiter) do
    count = count + 1
  end
```

```
  return count
end
result = wc('zombie apocalypse', ' ')
print(result)
result = wc('ice cream sandwich', ' ')
print(result)
result = wc('can you find the bug? ', ' ')
print(result)
```

You've just created a Lua script. You can run it with Lua. Can you find the problem with this method of counting words?

```
$ lua ./words.lua
2
3
6
```

You might notice that the count is incorrect for the final phrase because there's a trailing space in the argument. Lua correctly detected the space and tallied it into `count`, but the word count is incorrect because that particular space happens not to delimit a word. I leave it to you to solve that problem, or to find other ways in which this method isn't ideal. There's a lot of rumination in programming. Sometimes it's purely academic, and other times it's a question of whether an application works at all.

Learning Lua

Lua is a fun and robust language, with progressive improvements made with each release, and an ever-growing developer base. You can use Lua as a utilitarian language for personal scripts, or to advance your career, or just as an experiment in learning a new language. Give it a try, and see what Lua brings to the table.

Learn Lua by writing a "guess the number" game

If you're a fan of scripting languages like Bash, Python, or Ruby, you might find Lua interesting. Lua is a dynamically typed, lightweight, efficient, and embeddable scripting language with an API to interface with C. It runs by interpreting bytecode with a register-based virtual machine, and it can be used for everything from procedural programming to functional programming to data-driven programming. It can even be used for object-oriented programming through the clever use of arrays, or *tables*, used to mimic classes.

A great way to get a feel for a language is by writing a simple application you're already familiar with. Recently, some Opensource.com correspondents have demonstrated how to use their favorite languages to create a number-guessing game. Lua is one of my favorites, so here's my Lua version of the guessing game.

Install Lua

If you're on Linux, you can install Lua from your distribution's software repository. On macOS, you can install Lua from MacPorts or Homebrew. On Windows, you can install Lua from Chocolatey.

Once you have Lua installed, open your favorite text editor and get ready to code.

Lua code

First, you must set up a pseudo-random number generator, so your player has something unpredictable to try to guess. This is a two-step process: first, you start a random seed based on the current time, and then you select a number within the range of 1 to 100:

```
math.randomseed(os.time())
number = math.random(1,100)
```

Creative Commons Attribution Share-alike 4.0

Next, create what Lua calls a *table* to represent your player. A table is like an [array in Bash] or an ArrayList in Java. You can create a table and then assign child variables associated with that table. In this code, `player` is the table, and `player.guess` is an entry in that table:

```
player = {}
player.guess = 0
```

For the purpose of debugging, print the secret number. This isn't good for the game, but it's great for testing. Comments in Lua are preceded by double dashes:

```
print(number) --debug
```

Next, set up a `while` loop that runs forever upon the condition that the value assigned to `player.guess` is not equal to the random `number` established at the start of the code. Currently, `player.guess` is set to 0, so it is not equal to `number`. Lua's math operator for inequality is ~=, which is admittedly unique, but you get used to it after a while.

The first thing that happens during this infinite loop is that the game prints a prompt so that the player understands the game.

Next, Lua pauses and waits for the player to enter a guess. Lua reads from files and standard in (stdin) using the `io.read` function. You can assign the results of `io.read` to a variable that is dynamically created in the `player` table. The problem with the player's input is that it is read as a string, even if it's a number. You can convert this input to an integer type using the `tonumber()` function, assigning the result back to the `player.guess` variable that initially contained 0:

```
while ( player.guess ~= number ) do
  print("Guess a number between 1 and 100")
  player.answer = io.read()
  player.guess = tonumber(player.answer)
```

Now that `player.guess` contains a new value, it's compared to the random number in an `if` statement. Lua uses the keywords `if`, `elseif`, and `else` and terminates the statement with the keyword `end:`

```
  if ( player.guess > number ) then
    print("Too high")
  elseif ( player.guess < number) then
    print("Too low")
  else
```

```
    print("That's right!")
    os.exit()
  end
end
```

At the end, the function `os.exit()` closes the application upon success and the keyword `end` is used twice: once to end the `if` statement and again to end the `while` loop.

Run the application

Run the game in a terminal:

```
$ lua ./guess.lua
96
Guess a number between 1 and 100
1
Too low
Guess a number between 1 and 100
99
Too high
Guess a number between 1 and 100
96
That's right!
```

That's it!

Intuitive and consistent

As you may be able to tell from this code, Lua is sublimely consistent and fairly intuitive. Its table mechanism is a refreshing way of associating data, and its syntax is minimalistic and efficient. There are few wasted lines in Lua code; in fact, at least one pair of lines in this example could be optimized further, but I wanted to demonstrate data conversion as its own step (maybe you can find the two lines I'm referring to and restructure them).

Lua is a pleasure to use, and its documentation is a pleasure to read, mostly because there's just not that much to it. You'll learn the core language in no time, and then you'll be free to explore LuaRocks to discover all the great libraries others have contributed to make your time with Lua even easier. "Lua" means "moon" in Portuguese, so give it a try tonight.

Trick Lua into becoming an object-oriented language

Lua isn't an object-oriented programming language, but a scripting language utilizing C functions and a C-like syntax. However, there's a cool hack you can use within Lua code to make Lua act like an object-oriented language when you need it to be. The key is in the Lua *table* construct, and this article demonstrates how to use a Lua table as a stand-in for an object-oriented class.

What is object-oriented programming?

The term "object-oriented" is a fancy way of describing, essentially, a templating system. Imagine you're programming an application to help users spot and log zombies during a zombie apocalypse. You're using an object-oriented language like C++, Java, or Python. You need to create code objects that represent different types of zombies so the user can drag them around and arrange them on a map of the city. Of course a *zombie* can be any number of things: dormant, slow, fast, hungry, ravenous, and so on. That's just textual data, which computers are good at tracking, and based on that data you could even assign the virtual "object" a graphic so your user can identify which general type of zombie each widget represents.

You have a few options for how you can resolve this requirement for your application:

- Force your users to learn how to code so they can program their own zombies into your application
- Spend the rest of your life programming every possible type of zombie into your application
- Use a code template to define the attributes of a *zombie* object, allowing your users to create just the items they need, based on what zombie they've actually spotted

Obviously, the only realistic option is the final one, and it's done with a programming construct called a *class*. Here's what a class might look like (this example happens to be Java code, but the concept is the same across all object-oriented languages):

```
public class Zombie {
  int height;
  int weight;
  String speed;
  String location;
}
```

Whether or not you understand the code, you can probably tell that this is essentially a template. It's declaring that when a virtual "object" is created to represent a zombie, the programming language assigns that object four attributes (two integers representing height and weight, and two words representing the movement speed and physical location). When the user clicks the (imaginary) **Add item** button in the (imaginary) application, this class is used as a template (in programming, they say "a new instance" of the class has been created) to assign values entered by the user. Infinite zombies for the price of just one class. That's one of the powers of object-oriented programming.

Lua tables

In Lua, the *table* is a data type that implements an associative array. You can think of a table in Lua as a database. It's a store of indexed information that you can recall by using a special syntax. Here's a very simple example:

```
example = {}

example.greeting = "hello"
example.space = " "
example.subject = "world"

print(example.greeting ..
  example.space ..
  example.subject)
```

Run the example code to see the results:

```
$ lua ./example.lua
hello world
```

As you can tell from the sample code, a table is essentially a bunch of keys and values kept within a single collection (a "table").

Lua metatable

A metatable is a table that serves as a template for a table. You can designate any table as a metatable, and then treat it much as you would a class in any other language.

Here's a metatable to define a zombie:

```
Zombie = {}
function Zombie.init(h,w,s,l)
  local self = setmetatable({}, Zombie)
  self.height = h
  self.weight = w
  self.speed = s
  self.location = l
  return self
end
-- use the metatable
function setup()
  z1 = Zombie.init(176,50,'slow','Forbes & Murray Avenue')
end
function run()
  print(z1.location .. ": " ..
  z1.height .. " cm, " .. z1.weight .. " kg, " .. z1.speed)
end
setup()
run()
```

To differentiate my metatable from a normal table, I capitalize the first letter of its name. That's not required, but I find it a useful convention.

Here's the results of the sample code:

```
$ lua ./zombie.lua
Forbes & Murray Avenue: 176 cm, 50 kg, slow
```

This demonstration doesn't entirely do metatables justice, because the sample code creates just a single object with no interaction from the user. To use a metatable to satisfy the issue of creating infinite items from just one metatable, you would instead code a user input function (using Lua's `io.read()` function) asking the user to provide the details of the zombie they spotted. You'd probably even code a user interface with a "New sighting" button, or

something like that. That's beyond the scope of this article, though, and would only complicate the example code.

Creating a metatable

The important thing to remember is that to create a metatable, you use this syntax:

```
Example = {}
function Example.init(args)
  local self = setmetatable({}, Example)
  self.key = value
  return self
end
```

Using a metatable

To use a metatable once it's been created, use this syntax:

```
my_instance = Example.init("required args")
```

Object-oriented Lua

Lua isn't object-oriented, but its table mechanism makes handling and tracking and sorting data as simple as it possibly can be. It's no exaggeration to say that once you're comfortable with tables in Lua, you can be confident that you know at least half of the language. You can use tables (and, by extension, metatables) as arrays, maps, convenient variable organization, close-enough classes, and more.

How to iterate over tables in Lua

In the Lua programming language, an array is called a table. A table is used in Lua to store data. If you're storing a lot of data in a structured way, it's useful to know your options for retrieving that data when you need it.

Creating a table in Lua

To create a table in Lua, you instantiate the table with an arbitrary name:

```
mytable = {}
```

There are different ways you can structure your data in a table. You could fill it with values, essentially creating a list (called a *list* in some languages):

```
mytable = { 'zombie', 'apocalypse' }
```

Or you could create an associated array (called a *map* or *dictionary* in some languages). You can add arbitrary keys to the table using dot notation. You can also add a value to that key the same way you add a value to a variable:

```
myarray = {}
myarray.baz = 'happy'
myarray.qux = 'halloween'
```

You can add verification with the `assert()` function:

```
assert(myarray.baz == 'happy', 'unexpected value in myarray.baz')
assert(myarray.qux == 'halloween', 'unexpected value in myarray.qux')
```

You now have two tables: a list-style `mytable` and an associative array-style `myarray`.

Iterating over a table with pairs

Lua's `pairs()` function extracts key and value pairs from a table.

```
print('pairs of myarray:')
for k,v in pairs(myarray) do
  print(k,v)
end
```

Here's the output:

```
pairs of myarray:
baz     happy
qux     halloween
```

If there are no keys in a table, Lua uses an index. For instance, the `mytable` table contains the values `zombie` and `apocalypse`. It contains no keys, but Lua can improvise:

```
print('pairs of mytable:')
for k,v in pairs(mytable) do
  print(k,v)
end
```

Here's the output:

```
1   zombie
2   apocalypse
```

Iterating over a table with ipairs

To account for the fact that tables without keys are common, Lua also provides the `ipairs` function. This function extracts the index and the value:

```
print('ipairs of mytable:')
for i,v in ipairs(mytable) do
  print(i,v)
end
```

The output is, in this case, the same as the output of `pairs`:

```
1   zombie
2   apocalypse
```

However, watch what happens when you add a key and value pair to `mytable`:

```
mytable.surprise = 'this value has a key'
print('ipairs of mytable:')
for i,v in ipairs(mytable) do
  print(i,v)
end
```

Lua ignores the key and value because `ipairs` retrieves *only* indexed entries:

```
1   zombie
2   apocalypse
```

The key and value pair, however, have been stored in the table:

```
print('pairs of mytable:')
for k,v in ipairs(mytable) do
  print(k,v)
end
```

The output:

```
1          zombie
2          apocalypse
surprise   this value has a key
```

Retrieving arbitrary values

You don't have to iterate over a table to get data out of it. You can call arbitrary data by either index or key:

```
print('call by index:')
print(mytable[2])
print(mytable[1])
print(myarray[2])
print(myarray[1])

print('call by key:')
print(myarray['qux'])
print(myarray['baz'])
print(mytable['surprise'])
```

The output:

```
call by index:
apocalypse
zombie
nil
nil
call by key:
halloween
happy
this value has a key
```

Data structures

Sometimes using a Lua table makes a lot more sense than trying to keep track of dozens of individual variables. Once you understand how to structure and retrieve data in a language, you're empowered to generate complex data in an organized and safe way.

Manipulate data in files with Lua

Some data is ephemeral, stored in RAM, and only significant while an application is running. But some data is meant to be persistent, stored on a hard drive for later use. When you program, whether you're working on a simple script or a complex suite of tools, it's common to need to read and write files. Sometimes a file may contain configuration options, and other times the file is the data that your user is creating with your application. Every language handles this task a little differently, and this article demonstrates how to handle data files with Lua.

Installing Lua

If you're on Linux, you can install Lua from your distribution's software repository. On macOS, you can install Lua from MacPorts or Homebrew. On Windows, you can install Lua from Chocolatey.

Once you have Lua installed, open your favorite text editor and get ready to code.

Reading a file with Lua

Lua uses the `io` library for data input and output. The following example creates a function called `ingest` to read data from a file and then parses it with the `:read` function. When opening a file in Lua, there are several modes you can enable. Because I just need to read data from this file, I use the `r` (for "read") mode:

```
function ingest(file)
   local f = io.open(file, "r")
   local lines = f:read("*all")
   f:close()
   return(lines)
end

myfile=ingest("example.txt")
```

```
print(myfile)
```

In the code, notice that the variable `myfile` is created to trigger the `ingest` function, and therefore, it receives whatever that function returns. The `ingest` function returns the lines (from a variable intuitively called `lines`) of the file. When the contents of the `myfile` variable are printed in the final step, the lines of the file appear in the terminal.

If the file `example.txt` contains configuration options, then I would write some additional code to parse that data, probably using another Lua library depending on whether the configuration was stored as an INI file or YAML file or some other format. If the data were an SVG graphic, I'd write extra code to parse XML, probably using an SVG library for Lua. In other words, the data your code reads can be manipulated once it's loaded into memory, but all that's required to load it is the `io` library.

Writing data to a file with Lua

Whether you're storing data your user is creating with your application or just metadata about what the user is doing in an application (for instance, game saves or recent songs played), there are many good reasons to store data for later use. In Lua, this is achieved through the `io` library by opening a file, writing data into it, and closing the file:

```
function exgest(file)
   local f = io.open(file, "a")
   io.output(f)
   io.write("hello world\n")
   io.close(f)
end

exgest("example.txt")
```

To read data from the file, I open the file in `r` mode, but this time I use `a` (for "append") to write data to the end of the file. Because I'm writing plain text into a file, I added my own newline character (`\n`). Often, you're not writing raw text into a file, and you'll probably use an additional library to write a specific format instead. For instance, you might use an INI or YAML library to help write configuration files, an XML library to write XML, and so on.

File modes

When opening files in Lua, there are some safeguards and parameters to define how a file should be handled. The default is `r`, which permits you to read data only:

- **r** for read only
- **w** to overwrite or create a new file if it doesn't already exist
- **r+** to read and overwrite
- **a** to append data to a file or make a new file if it doesn't already exist
- **a+** to read data, append data to a file, or make a new file if it doesn't already exist

There are a few others (`b` for binary formats, for instance), but those are the most common. For the full documentation, refer to the excellent Lua documentation on Lua.org/manual.

Lua and files

Like other programming languages, Lua has plenty of library support to access a filesystem to read and write data. Because Lua has a consistent and simple syntax, it's easy to perform complex processing on data in files of any format. Try using Lua for your next software project, or as an API for your C or C++ project.

Creative Commons Attribution Share-alike 4.0

Parse arguments with Lua

Most computer commands consist of two parts: The command and arguments. The command is the program meant to be executed, while the arguments might be command options or user input. Without this structure, a user would have to edit the command's code just to change the data that the command processes. Imagine rewriting the printf command just to get your computer to greet you with a "hello world" message. Arguments are vital to interactive computing, and the Lua programming language provides the {…} expression to encapsulate varargs given at the time of launching a Lua script.

Use arguments in Lua

Almost every command given to a computer assumes an argument, even if it expects the argument to be an empty list. Lua records what's written after it launches, even though you may do nothing with those arguments. To use arguments provided by the user when Lua starts, iterate over the {…} table:

```
local args = {...}
for i,v in ipairs(args) do
    print(v)
end
```

Run the code:

```
$ lua ./myargs.lua
$ lua ./myargs.lua foo --bar baz
foo
--bar
baz
----
```

Having no arguments is safe, and Lua prints all arguments exactly as entered.

Creative Commons Attribution Share-alike 4.0

Parse arguments

For simple commands, the basic Lua faculties are sufficient to parse and process arguments. Here's a simple example:

```
-- setup
local args = {...}
-- engine
function echo(p)
   print(p)
end
-- go
for i,v in ipairs(args) do
  print(i .. ": " .. v)
end
for i,v in ipairs(args) do
  if args[i] == "--say" then
    echo("echo: " .. args[i+1])
  end
end
```

In the `setup` section, dump all command arguments into a variable called `args`.

In the `engine` section, create a function called `echo` that prints whatever you "feed" into it.

Finally, in the `go` section, parse the index and values in the `args` variable (the arguments provided by the user at launch). In this sample code, the first `for` loop just prints each index and value for clarity.

The second `for` loop uses the index to examine the first argument, which is assumed to be an option. The only valid option in this sample code is `--say`. If the loop finds the string `--say`, it calls the `echo` function, and the index of the current argument *plus 1* (the *next* argument) is provided as the function parameter.

The delimiter for command arguments is one or more empty spaces. Run the code to see the result:

```
$ lua ./echo.lua --say zombie apocalypse
1: --say
2: zombie
3: apocalypse
echo: zombie
```

Most users learn that spaces are significant when giving commands to a computer, so dropping the third argument, in this case, is expected behavior. Here's a variation to demonstrate two valid "escape" methods:

```
$ lua ./echo.lua --say "zombie apocalypse"
1: --say
2: zombie apocalypse
echo: zombie apocalypse

$ lua ./echo.lua --say zombie\ apocalypse
1: --say
2: zombie apocalypse
echo: zombie apocalypse
```

Parse arguments

Parsing arguments manually is simple and dependency-free. However, there are details you must consider. Most modern commands allow for short options (for instance, `-f`) and long options (`--foo`), and most offer a help menu with `-h` or `--help` or when a required argument isn't supplied.

Using LuaRocks makes it easy to install additional libraries. There are some very good ones, such as alt-getopt, that provide additional infrastructure for parsing arguments.

Make Lua development easy with Luarocks

Bash too basic? Too much whitespace in Python? Go too corporate?

You should try Lua, a lightweight, efficient, and embeddable scripting language supporting procedural programming, object-oriented programming, functional programming, data-driven programming, and data description. And best of all, it uses explicit syntax for scoping!

Lua is also small. Lua's source code is just 24,000 lines of C, the Lua interpreter (on 64-bit Linux) built with all standard Lua libraries is 247K, and the Lua library is 421K.

You might think that such a small language must be too simplistic to do any real work, but in fact Lua has a vast collection of third-party libraries (including GUI toolkits), it's used extensively in video game and film production for 3D shaders, and is a common scripting language for video game engines. To make it easy to get started with Lua, there's even a package manager called Luarocks.

What is Luarocks?

Python has PIP, Ruby has Gems, Java has Maven, Node has npm, and Lua has Luarocks. Luarocks is a website and a command. The website is home to open source libraries available for programmers to add to their Lua projects. The command searches the site and installs libraries (defined as "rocks") upon demand.

What is a programming library?

If you're new to programming, you might think of a "library" as just a place where books are stored. Programming libraries ("lib" or "libs" for short) are a little like a book library in the sense that both of these things contain information that someone else has already worked to discover, and which you can borrow so you have to do less work.

For example, if you were writing code that measures how much stress a special polymer can withstand before breaking, you might think you'd have to be pretty clever with math. But if there was already an open source library specifically designed for exactly that sort of calculation, then you could include that library in your code and let it solve that problem for you (provided you give the library's internal functions the numbers it needs in order to perform an accurate calculation).

In open source programming, you can install libraries freely and use other people's work at will. Luarocks is the mechanism for Lua that makes it quick and easy to find and use a Lua library.

Installing Luarocks

The **luarocks** command isn't actually *required* to use packages from the Luarocks website, but it does keep you from having to leave your text editor and venture onto the worldwide web [of potential distractions]. To install Luarocks, you first need to install Lua.

Lua is available from lua.org or, on Linux, from your distribution's software repository. For example, on Fedora, CentOS, or RHEL:

```
$ sudo dnf install lua
```

On Debian and Ubuntu:

```
$ sudo apt install lua
```

On Windows and Mac, you can download and install Lua from the website.

Once Lua is installed, install Luarocks. If you're on Linux, the **luarocks** command is available in your distribution's repository.

On Mac, you can install it with Brew or compile from source:

```
$ wget https://luarocks.org/releases/luarocks-X.Y.Z.tar.gz
$ tar zxpf luarocks-X.Y.Z.tar.gz
$ cd luarocks-X.Y.Z
$ ./configure; sudo make bootstrap
```

On Windows, follow the install instructions on the Luarocks wiki.

Search for a library with Luarocks

The typical usage of the **luarocks** command, from the perspective of a user rather than a developer, involves searching for a library required by some Lua application you want to run and installing that library.

To search for the Lua package **luasec** (a library providing HTTPS support for **luarocks**), try this command:

```
$ luarocks search luasec
Warning: falling back to curl -
install luasec to get native HTTPS support

Search results:
===============

Rockspecs and source rocks:
---------------------------

luasec
   0.9-1 (rockspec) - https://luarocks.org
   0.9-1 (src) - https://luarocks.org
   0.8.2-1 (rockspec) - https://luarocks.org
[...]
```

Install a library with Luarocks

To install the **luasec** library:

```
$ luarocks install --local luasec
[...]
gcc -shared -o ssl.so -L/usr/lib64
src/config.o src/ec.o src/x509.o [...]
-L/usr/lib -Wl,-rpath,/usr/lib: -lssl -lcrypto

luasec 0.9-1 is now installed in
/home/seth/.luarocks (license: MIT)
```

You can install Lua libraries locally or on a systemwide basis. A *local* install indicates that the Lua library you install is available to you, but no other user of the computer. If you share your computer with someone else, and you each have your own login account, then you probably want to install a library systemwide. However, if you're the only user of your computer, it's a

good habit to install libraries locally, if only because that's the appropriate method when you develop with Lua.

If you're *developing* a Lua application, then you probably want to install a library to a project directory instead. In Luarocks terminology, this is a *tree*. Your default tree when installing libraries locally is **$HOME/.luarocks**, but you can redefine it arbitrarily.

```
$ mkdir local
$ luarocks --tree=./local install cmark
Installing https://luarocks.org/cmark-0.YY.0-1.src.rock
gcc -O2 -fPIC -I/usr/include -c cmark_wrap.c [..]
gcc -O2 -fPIC -I/usr/include -c ext/blocks.c -o ext/blocks.o [..]
[...]
No existing manifest. Attempting to rebuild...
cmark 0.29.0-1 is now installed in
/home/seth/downloads/osdc/example-lua/./local
(license: BSD2)
```

The library (in this example, the **cmark** library) is installed to the path specified by the **--tree** option. You can verify it by listing the contents of the destination:

```
$ find ./local/ -type d -name "cmark"
./local/share/lua/5.1/cmark
./local/lib/luarocks/rocks/cmark
```

You can use the library in your Lua code by defining the **package.path** variable to point to your local rocks directory:

```
package.path = package.path .. ';local/share/lua/5.3/?.lua'

require("cmark")
```

If a library you've installed is compiled, resulting in a **.so** file (a **.dll** on Windows and .dylib on macOS), then you must add to your **cpath** instead. For instance, the **luafilesystem** library provides the file **lfs.so**:

```
package.cpath = package.cpath .. ';local/share/lua/5.3/?.lua'

require("lfs")
```

Getting information about an installed rock

You can see information about an installed rock with the **show** option:

```
$ luarocks show luasec
LuaSec 0.9-1 - A binding for OpenSSL library
to provide TLS/SSL communication over LuaSocket.

This version delegates to LuaSocket the TCP
connection establishment between
the client and server. Then LuaSec uses this
connection to start a secure TLS/SSL session.

License:        MIT
Homepage:       https://github.com/brunoos/luasec/wiki
Installed in:   /home/seth/.luarocks
[...]
```

This provides you with a summary of what a library provides from a user's perspective, displays the project homepage in case you want to investigate further, and shows you where the library is installed. In this example, it's installed in my home directory in a **.luarocks** folder. This assures me that it's installed locally, which means that if I migrate my home directory to a different computer, I'll retain my Luarocks configuration and installs.

Get a list of installed rocks

You can list all installed rocks on your system with the **list** option:

```
$ luarocks list

Installed rocks:
----------------

luasec
   0.9-1 (installed) - /home/seth/.luarocks/lib/luarocks/rocks

luasocket
   3.0rc1-2 (installed) - /home/seth/.luarocks/lib/luarocks/rocks

luce
   scm-0 (installed) - /home/seth/.luarocks/lib/luarocks/rocks

tekui
   1.07-1 (installed) - /home/seth/.luarocks/lib/luarocks/rocks
```

This displays the rocks you have installed in the default install location. Developers can override this by using the **--tree** option to redefine the active tree.

Remove a rock

If you want to remove a rock, you can do that with Luarocks using the **remove** option:

```
$ luarocks remove --local cmark
```

This removes a library (in this example, the **cmark** library) from your local tree. Developers can override this by using the **--tree** option to redefine the active tree.

If you want to remove *all* the rocks you have installed, use the **purge** option instead.

Luarocks rocks

Whether you're a user exploring exciting new Lua applications and need to install some dependencies or you're a developer using Lua to create exciting new applications, Luarocks makes your job easy. Lua is a beautiful and simple language, and Luarocks is perfectly suited to be its package manager. Give both a try today!Parsing config files with Lua

Parsing config files with Lua

Not all applications need configuration files; many applications benefit from starting fresh each time they are launched. Simple utilities, for instance, rarely require preferences or settings that persist across uses. However, when you write a complex application, it's nice for users to be able to configure how they interact with it and how it interacts with their system. That's what configuration files are for, and this article discusses some of the ways you can implement persistent settings with the Lua programming language.

Choose a format

The important thing about configuration files is that they are consistent and predictable. You do not want to dump information into a file under the auspices of saving user preferences and then spend days writing code to reverse-engineer the random bits of information that have ended up in the file.

There are several popular formats for configuration files. Lua has libraries for most of the common configuration formats; in this article, I'll use the INI format.

Installing the library

The central hub for Lua libraries is Luarocks.org. You can search for libraries on the website, or you can install and use the `luarocks` terminal command.

On Linux, you can install it from your distribution's software repository. For example:

```
$ sudo dnf install luarocks
```

On macOS, use MacPorts or Homebrew. On Windows, use Chocolatey.

Once `luarocks` is installed, you can use the `search` subcommand to search for an appropriate library. If you don't know the name of a library, you can search for a keyword, like `ini` or `xml` or `json`, depending on what's relevant to what you're trying to do. In this case, you can just search for `inifile`, which is the library I use to parse text files in the INI format:

```
$ luarocks search inifile
Search results:
inifile
 1.0-2 (rockspec) - https://luarocks.org
 1.0-2 (src) - https://luarocks.org
 1.0-1 (rockspec) - https://luarocks.org
 [...]
```

A common trap programmers fall into is installing a library on their system and forgetting to bundle it with their application. This can create problems for users who don't have the library installed. To avoid this, use the `--tree` option to install the library to a local folder within your project directory. If you don't have a project directory, create one first, and then install:

```
$ mkdir demo
$ cd demo
$ luarocks install --tree=local inifile
```

The `--tree` option tells `luarocks` to create a new directory, called `local` in this case, and install your library into it. With this simple trick, you can install all the dependency code your project uses directly into the project directory.

Code setup

First, create some INI data in a file called `myconfig.ini`:

```
[example]
name=Tux
species=penguin
enabled=false

[demo]
name=Beastie
species=demon
enabled=false
```

Save the file as `myconfig.ini` into your home directory, *not* into your project directory. You usually want configuration files to exist outside your application so that even when a user

uninstalls your application, the data they generate while using the application remains on their system. Users might remove unnecessary config files manually, but many don't. As a result, if they reinstall an application, it will retain all of their preferences.

Config file locations are technically unimportant, but each operating system (OS) has a specification or a tradition of where they ought to be placed. On Linux, this is defined by the Freedesktop specification. It dictates that configuration files are to be saved in a hidden folder named `~/.config`. For clarity during this exercise, just save the file in your home directory so that it's easy to find and use.

Create a second file named `main.lua` and open it in your favorite text editor.

First, you must tell Lua where you've placed the additional library you want it to use. The `package.path` variable determines where Lua looks for libraries. You can view Lua's default package path in a terminal:

```
$ Lua
> print(package.path)
./?.lua;/usr/share/lua/5.3/?.lua;/usr/share/lua/5.3/?/init.lua;/usr/lib64/lua/
5.3/?.lua;/usr/lib64/lua/5.3/?/init.lua
```

In your Lua code, append your local library location to `package.path`:

```
package.path = package.path .. ';local/share/lua/5.3/?.lua
```

Parsing INI files with Lua

With the package location established, the next thing to do is to require the `inifile` library and then handle some OS logistics. Even though this is a simple example application, the code needs to get the user's home directory location from the OS and establish how to communicate filesystem paths back to the OS when necessary:

```
package.path = package.path .. ';local/share/lua/5.3/?.lua
inifile = require('inifile')

-- find home directory
home = os.getenv('HOME')

-- detect path separator
-- returns '/' for Linux and Mac
-- and '\' for Windows
d = package.config:sub(1,1)
```

Now you can use `inifile` to parse data from the config file into a Lua table. Once the data has been placed into a table, you can query the table as you would any other Lua table:

```
-- parse the INI file and
-- put values into a table called conf
conf = inifile.parse(home .. d .. 'myconfig.ini')

-- print the data for review
print(conf['example']['name'])
print(conf['example']['species'])
print(conf['example']['enabled'])
```

Run the code in a terminal to see the results:

```
$ lua ./main.lua
Tux
penguin
false
```

That looks correct. Try doing the same for the `demo` block.

Saving data in the INI format

Not all parser libraries read and write data (often called *encoding* and *decoding*), but the `inifile` library does. That means you can use it to make changes to a configuration file.

To change a value in a configuration file, you set the variable representing the value in the parsed table, and then you write the table back to the configuration file:

```
-- set enabled to true
conf['example']['enabled'] = true
conf['demo']['enabled'] = true

-- save the change
inifile.save(home .. d .. 'myconfig.ini', conf)
```

Take a look at the configuration file now:

```
$ cat ~/myconfig.ini
[example]
name=Tux
species=penguin
enabled=true

[demo]
```

```
name=Beastie
species=demon
enabled=true
```

Config files

The ability to save data about how a user wants to use an application is an important part of programming. Fortunately, it's a common task for programmers, so much of the work has probably already been done. Find a good library for encoding and decoding into an open format, and you can provide a persistent and consistent user experience.

Here's the entire demo code for reference:

```
package.path = package.path .. ';local/share/lua/5.3/?.lua'
inifile = require('inifile')

-- find home directory
home = os.getenv('HOME')

-- detect path separator
-- returns '/' for Linux and Mac
-- and '\' for Windows
d = package.config:sub(1,1)

-- parse the INI file and
-- put values into a table called conf
conf = inifile.parse(home .. d .. 'myconfig.ini')

-- print the data for review
print(conf['example']['name'])
print(conf['example']['species'])
print(conf['example']['enabled'])

-- enable Tux
conf['example']['enabled'] = true

-- save the change
inifile.save(home .. d .. 'myconfig.ini', conf)
```

Parsing command options in Lua

When you enter a command into your terminal, there are usually options, also called *switches* or *flags*, that you can use to modify how the command runs. This is a useful convention defined by the POSIX specification, so as a programmer, it's helpful to know how to detect and parse options.

As with most languages, there are several ways to solve the problem of parsing options in Lua. My favorite is alt-getopt.

Installing

The easiest way to obtain and use **alt-getopt** in your code is to install it with LuaRocks. For most use-cases, you probably want to install it into your local project directory:

```
$ mkdir local
$ luarocks --tree=local install alt-getopt
Installing https://luarocks.org/alt-getopt-0.X.Y-1.src.rock
[...]
alt-getopt 0.X.Y-1 is now installed in /tux/myproject/local (license: MIT/X11)
```

Alternately, you can download the code from GitHub.

Adding a library to your Lua code

Assuming you've installed the library locally, then you can define your Lua package path and then `require` the **alt-getopt** package:

```
package.path = package.path .. ';local/share/lua/5.1/?.lua'
local alt_getopt = require("alt_getopt")
```

If you've installed it to a known system location, you can omit the `package.path` line (because Lua already knows to look for system-wide libraries.)

Now you're set to parse options in Lua.

Option parsing in Lua

The first thing you must do to parse options is to define the valid options your application can accept. The **alt_getopt** library sees all options primarily as short options, meaning that you define options as single letters. You can add long versions later.

When you define valid options, you create a list delimited by colons (`:`), which is interpreted by the `get_opts` function provided by **alt-getopts**.

First, create some variables to represent the options. The variables `short_opt` and `optarg` represent the short option and the option argument. These are arbitrary variable names, so you can call them anything (as long as it's a valid name for a variable).

The table `long_opts` must exist, but I find it easiest to define the values of the long options after you've decided on the short options, so leave it empty for now.

```
local long_opts = {}
local short_opt
local optarg
```

With those variables declared, you can iterate over the arguments provided by the user, checking to see whether any argument matches your approved list of valid short options.

If a valid option is found, you use the `pairs` function in Lua to extract the value of the option.

To create an option that accepts no argument of its own but is either *on* or *off* (often called a *switch*), you place the short option you want to define to the right of a colon (`:`) character.

In this example, I've created a loop to check for the short option `-a`, which is a switch:

```
short_opt,optarg = alt_getopt.get_opts (arg, ":a", long_opts)
local optvalues = {}
for k,v in pairs (short_opt) do
   table.insert (optvalues, "value of " .. k .. " is " .. v .. "\n")
end

table.sort (optvalues)
io.write (table.concat (optvalues))

for i = optarg,#arg do
   io.write (string.format ("ARGV [%s] = %s\n", i, arg [i]))
end
```

At the end of this sample code, I included a for-loop to iterate over any remaining arguments in the command because anything not detected as a valid option must be an argument (probably a file name, URI, or whatever it is that your application operates upon).

Test the code:

```
$ lua test.lua -a
value of a is 1
```

The test script has successfully detected the option `-a`, and has assigned it a value of **1** to represent that it does exist.

Try it again with an extra argument:

```
$ lua test.lua -a hello
value of a is 1
ARGV [2] = hello
```

Options with arguments

Some options require an argument all their own. For instance, you might want to allow the user to point your application to a custom configuration file, set an attribute to a color, or set the resolution of a graphic. In **alt_getopt**, options that accept arguments are placed on the left of the colon (`:`) in the short options list.

```
short_opt,optarg = alt_getopt.get_opts (arg, "c:a", long_opts)
```

Test the code:

```
$ lua test.lua -a -c my.config
value of a is 1
value of c is my.config
```

Try it again, this time with a spare argument:

```
$ lua test.lua -a -c my.config hello
value of a is 1
value of c is my.config
ARGV [4] = hello
```

Long options

Short options are great for power users, but they don't tend to be very memorable. You can create a table of long options that point to short options so users can learn long options before abbreviating their commands with single-letter options.

```
local long_opts = {
   alpha = "a",
   config = "c"
}
```

Users now have the choice between long and short options:

```
$ lua test.lua --config my.config --alpha hello
value of a is 1
value of c is my.config
ARGV [4] = hello
```

Option parsing

Here's the full demonstration code for your reference:

```
#!/usr/bin/env lua
package.path = package.path .. ';local/share/lua/5.1/?.lua'

local alt_getopt = require("alt_getopt")

local long_opts = {
   alpha = "a",
   config = "c"
}

local short_opt
local optarg

short_opt,optarg = alt_getopt.get_opts (arg, "c:a", long_opts)
local optvalues = {}
for k,v in pairs (short_opt) do
   table.insert (optvalues, "value of " .. k .. " is " .. v .. "\n")
end

table.sort (optvalues)
io.write (table.concat (optvalues))

for i = optarg,#arg do
```

```
   io.write (string.format ("ARGV [%s] = %s\n", i, arg [i]))
end
```

There are further examples in the project's Git repository. Including options for your users is an important feature for any application, and Lua makes it easy to do. There are other libraries aside from **alt_getopt**, but I find this one easy and quick to use.

You're (almost) a Lua pro

You've reached the end of this book, but hopefully not the end of your Lua journey. There's actually not that much more to Lua, and that's a feature! Instead of spending your time learning more about a language, you're now free to start putting what you know into practice.

There are several great Lua resources out there, but as with much in open source, the best place to go when you have a question is straight to the source. The Lua website, https://www.lua.org, has great documentation, including the language specification itself and links to physical books you can keep stocked on your bookshelf for reference.

Lua has made programming a true pleasure for me, and I hope it does the same for you.

Creative Commons Attribution Share-alike 4.0