Lua Mini Reference 2023

A Quick Guide to the Lua Scripting Language for Busy Coders

Harry Yoon

Version 1.1.3, 2023-05-14

Copyright

Lua Mini Reference:
A Quick Guide to the Lua Scripting Language

Published: November 2022

Harry Yoon
San Diego, California

ISBN: 9798362177249

Preface

Few people who program actually go through any kind of formal learning process, for any programming languages. They just "pick up" pieces of information, here and there, and most of the time they learn by trials and errors, "on the job", so to speak.

This is especially true for languages like Lua. Many people pick up Lua while using beginner-friendly tools like Roblox, bit by bit. To many people who have experience with other programming languages, on the other hand, Lua appears to be an easy language (as in "not a serious language"), which does not require *learning*.

Clearly, these are misconceptions. Learning this way has, obviously, limitations. That is like building a house on the sand. Your progress will be slow in the long run when you are on the shaky ground. (And, Lua is no different from other "more professional languages".) This book will help you build a solid foundation on your future Lua programming journey. It goes through all the essential features of the language, not just the syntax, but all the crucial concepts.

This book is written for a broad audience with diverse background. We cover topics ranging from the absolute basics to rather advanced subjects. However, it is not for complete beginners. It is written as a reference style. Some basic programming knowledge is required to get the most out of this book. *It should be noted that this book is not a tutorial on how to program in Lua.*

On the flip side, although it is written as a reference, you can read, or browse, this book more or less from beginning to end to get the overall picture of the Lua programming language, as well as basic usages of all core library functions, if you have some experience in programming in Lua or some other similar languages.

Dear Readers:

Please read b4 you purchase, or start investing your time on, this book.

A programming language is like a set of standard lego blocks. There are small ones and there are big ones. Some blocks are straight and some are L-shaped. You use these lego blocks to build spaceships or submarines or amusement parks. Likewise, you build programs by assembling these building blocks of a given programming language.

This book is a *language reference*, written in an informal style. It goes through each of these lego blocks, if you will. This book, however, does not teach you how to build a space shuttle or a sail boat. If this distinction is not clear to you, it's unlikely that you will benefit much from this book. This kind of language reference books that go through the syntax and semantics of the programming language broadly, but not necessarily in gory details, can be rather useful to programmers with a wide range of background and across different skill levels.

This book is not for complete beginners, however. When you start learning a foreign language, for instance, you do not start from the grammar. Likewise, this book will not be very useful to people who have little experience in real programming. On the other hand, if you have some experience programming in other languages, and if you want to quickly learn the essential elements of this particular language, then this book can suit your needs rather well.

Ultimately, only you can decide whether this book will be useful for you. But, as stated, this book is written for a wide audience, from beginner to intermediate. Even experienced programmers can benefit, e.g., by quickly going through books like this once in a while. We all tend to forget things, and a quick regular refresher is always a good idea. You will learn, or re-learn, something "new" every time.

Good luck!

Table of Contents

Chapter 1. Introduction

Lua is a rather interesting language. It is one of the simplest general purpose programming languages, and it is easy to learn and easy to use. Lua is dynamically typed, and it supports automatic memory management (e.g., garbage collection).

Lua is primarily used as an embedded language, often to interface with the host programs written in C. Lua is also widely used as a standalone scripting language, for the purposes of configuring, scripting, and rapid prototyping, among other things.

Lua has many similarities with other dynamic languages like Python and JavaScript, and yet it is even simpler. It is an ideal language for beginning programmers to start learning programming with. If you want to enable scripting in your own programs, e.g., written in C or C++, or other similar languages, it is also the best language for your end users, who may not necessarily be experienced programmers.

For example, many gaming engines provide scripting via Lua. One of the most popular such gaming engines is Roblox. Many people learn Lua for Roblox programming. More professional software like Redis and Nmap also use Lua for scripting. In fact, the applications that use Lua for scripting are too numerous to list here. There are hundreds, if not thousands, of them.

If you like simplicity, efficiency, and convenience, then Lua is definitely a programming language for you. You will also see a lot of similarities between Lua and low-level, more rigorous, languages like Go (aka Golang). These languages pursue the "minimalism" in programming.

In this book, we go through all essential features of the modern Lua (as of 5.4) *as a standalone programming language.* Programming vs scripting is not a clear, mutually exclusive distinction, but there are some subtle differences. For example, when you write a Lua program

that runs on a terminal, the input/output of the program may be associated with the stdin/stdout of the terminal. On the other hand, when you write a Lua script in a host program environment, the input/output may be controlled by the host program.

Furthermore, although it's hard to make a generalization, the standalone programs generally tend to be longer than scripts, which can in turn require certain (subtly) different practices. For instance, an entire Lua script can be written in one line, whereas longer programs require better formatting, e.g., for readability, etc.

Note, however, that we will use the terms, programs and scripts, mostly interchangeably in this book.

For the purposes of this book, we will use the Lua Interpreter as the programming environment. It is the standard implementation of the official Lua language specification (produced and maintained by the creators of the language). You can skip the first chapter if you are familiar with the `lua` command line tool. Another popular Lua runtime is LuaJIT, which does a just-in-time compilation to machine code (like Java) during program execution.

Although Lua specifies various aspects of the embedded usage, including the C interface, there can be some variations from implementation to implementation of the host programs. Roblox, for example, uses a slight variation of the Lua language, called Luau. In this book, however, we will mainly focus on the standard Lua and the standard Lua interpreter implementation. Readers are encouraged to consult the documentations of the specific host environments.

In the next chapter, we discuss the top-level structure of the Lua programs and how they are executed, e.g., by the Lua interpreter. Lua also includes a few standard library functions for loading and executing external Lua code, which are included in this chapter for reference purposes. You can skip them in your "reading".

Lua programs can be organized into modules, which are just Lua code in separate files, following certain simple conventions.

As with any programs written in any programming languages, a Lua program is essentially a sequence of characters. The Lua interpreter first "tokenizes" it into various lexical elements such as identifiers and operators, and "literals". The builtin type literals such as numbers are discussed in the next chapter. Clearly, being able to recognize these lexical elements is a basic requirement for a programmer to be able to read and write Lua programs. All high-level programming languages have similar lexical components, and it is only the details that are different from language to language.

The Lua interpreter then parses this sequence of "tokens", according to the grammar, into an internal data structure (e.g., in the form of abstract syntax tree, or AST), and then it ultimately generates the "bytecode", so that it can be executed. The rest of the book focuses on the Lua language grammar.

Although Lua is a dynamically typed programming language, the type system is still at the foundation of the Lua language. Therefore, we begin our survey of the grammar from the Lua's basic types, namely, `nil`, `boolean`, `number`, `string`, `function`, `userdata`, `thread`, and `table`. All values in Lua belong to one of these eight basic types.

It is hard to clearly and succinctly define what a computer program is, in general. But, at the highest level, a program (in the imperative, or procedural, programming paradigm) is a sequence of instructions that processes an input and produces an output. In doing so, it maintains an internal state. A variable in Lua is a name that references a value, and they play a fundamental role in keeping the internal state during the program execution.

Lua has most of the usual constructs commonly found in other imperative programming languages such as expressions, functions, and statements. There are fewer, and they are only simpler, which makes

Lua the best beginner's programming language. This characteristic of Lua, the simplicity, also makes it an ideal language for embedding in other programs.

In the Expressions chapter, we go through all the essential types of Lua expressions, including arithmetic, logical and comparison operations, etc. One interesting thing about Lua is that it has a builtin length operator (`#`).

In Lua, all functions are anonymous, often known as the "lambda expressions" in other programming languages. A named function definition is essentially a variable declaration with the initial value of a function type. Lua also provides a more traditional syntax for defining a function which is comparable to those found in other programming languages. A function call in Lua is an expression, which can be used as a statement.

In the following Statements chapter, we go through all major Lua statements. Statements are primarily used for controlling program flow in imperative programming, e.g., for iterations and conditional executions, and so forth. Lua statements include

- The local variable declaration and assignment statements,
- The label and `goto` statements,
- The `do` block statement,
- The `break` statement,
- The `return` statement,
- The `if` statement,
- The `for` loop statement,
- The `while` and `repeat` loop statements,
- The (local and global) function definitions, and
- The function call statement.

The standard Lua comes with an absolutely minimal set of "standard libraries". It is clearly a tradeoff. If you need a rich set of standard library types and functions, then you will need to pick a different language for your task. On the other hand, this small footprint makes Lua ideal in many circumstances as stated earlier.

We first go through a few essential functions from the standard `math` library. Then, we review various operations on strings, in the following two chapters, string basics and advanced topics in strings.

If we can use terms like "silver bullets" in the context of Lua programming, then they apply to Lua Tables. In Lua, tables are used for basic data structures, as foundations of "objects", custom types, modules, and for object-oriented programming, to name just a few.

Some of the `table`'s power also come from the way that Lua allows the behavior of the tables to be customized using other associated tables, called the metatables. Lua uses a set of predefined methods in those metatables, known as metamethods, to provide the hook for customization, if you will. (They are rather similar to Python's "magic methods", for instance.) In the following chapter, we go through some examples of the predefined metamethods.

It should be noted that although metatables and metamethods, and tables in general, are covered (in detail) in the later part, they are used all throughout the book because of their pervasiveness and the overall importance in Lua programming. We introduce some essential concepts earlier in the book whenever relevant, and hence by necessity, there are some repetitions.

As is common in most modern programming languages, the iterations, e.g., using the `for` statements, can be customized for tables. The default implementations of the builtin `pairs` and `ipairs` functions can be used to iterate over tables. This behavior can be customized by overwriting the `__pairs` metamethod. This is described next in the Iterators chapter.

Lua is fundamentally an "object-based" programming language just like JavaScript and Python. The more traditional "class-based" object oriented programming style can be *emulated* using Lua's tables, if necessary, or if desired. Although it is not part of the language specification, we include a brief introduction to the object-oriented programming style in Lua for completeness since it is commonly used. Some platforms that provide the Lua API also use this OOP style (for various reasons). Hence, it is important to understand this basic concept even if you don't plan to create your own "classes".

Then, we go through some more of Lua's standard libraries, namely, the operating systems library and the input/output and file systems library. They are distributed over three tables, `os`, `io`, and `file`, and they provide essential functions when you write the (command line interface) programs in Lua. These "low-level" functions may not be too relevant in some hosting environments. For completeness, we briefly describe Lua's builtin error handling support next. This chapter is not, however, meant to provide a comprehensive coverage of error handling in Lua, which is beyond the scope of this book.

Finally, we describe the Lua's support for concurrency in the next chapter. The `thread` type, in conjunction with various helper functions from the standard library `coroutine` table, can be used for concurrent programming in Lua. A simple example in the context of the producer-consumer problem is provided as a final example, and as an exercise. The readers are encouraged to go through this section to make sure that they have a firm grasp of the Lua concurrency, one of the more difficult subjects in Lua programming.

One thing to note is that since we describe Lua as a standalone programming language, we do not include the C programming interface defined in the official Lua language specification in this book. If you plan to write a program (e.g., in a C-compatible language) that supports Lua scripting, or if you plan to use C libraries in your Lua program, then you may need to consult the relevant references on the C API.

Chapter 2. Lua Interpreter

2.1. Lua Scripts

Lua can be used as an embedded language, e.g., in a host program written in C or C++. Or, it can also be used as a standalone general purpose programming language. This book primarily focuses on the standalone usage of Lua.

The standard distribution comes with a command line Lua interpreter, called *lua*, which is built with Lua's all standard libraries.

- Lua: Download [https://www.lua.org/download.html]

It prints out the basic usage information if we invoke the command with an invalid syntax:

```
$ lua -h                                        ①

usage: lua [options] [script [args]]

Available options are:
  -e stat   execute string 'stat'
  -i        enter interactive mode after executing 'script'
  -l mod    require library 'mod' into global 'mod'
  -l g=mod  require library 'mod' into global 'g'
  -v        show version information
  -E        ignore environment variables
  -W        turn warnings on
  --        stop handling options
  -         stop handling options and execute stdin
```

① The *lua* command does not recognize the *-h* option. The output is slightly modified to reduce clutter. Here, and throughout this book, $ represents the shell prompt.

The options can be combined. If a Lua script name is provided at the end (followed by any optional program arguments), and/or if the Lua script, as a string, is provided as an argument to the *-e* flag, then the interpreter executes those scripts. For example,

```
$ lua -e "print('Hello,World!')"     ①
Hello, World!
$                                    ②
```

① The `print` function prints the function argument to the default output device, which is the standard out (e.g., the terminal) when using the Lua interpreter this way.

② The Lua interpreter automatically exits after executing the given script, as indicated by the shell prompt `$` in this line.

Invoking the *lua* command without an *-e* option or a script name argument, or with an explicit *-i*, starts the interpreter in the interactive mode (e.g., after executing the provided script, if any).

```
$ lua
Lua 5.4.4  Copyright (C) 1994-2022 Lua.org, PUC-Rio
>                                    ①
```

① The prompt of the lua interpreter (`>`). You can exit the interactive mode using the EOF signal (e.g., Ctrl+D on Unix, Ctrl+Z on Windows). Or, you can call Lua's standard library function `os.exit()`.

Command line arguments to a Lua script can be provided after the script name. Lua collects all command line components in a global table called `arg`. The script name goes to index 0, the first program argument after the script name goes to index 1, and so forth. Any components before the script name (e.g., the interpreter command plus its command line options) are collected in the negative indices.

Like all chunks in Lua, the given script is compiled as a vararg function, and the program arguments (after the script name), if any, are provided as arguments to the function.

```
$ lua print-arg.lua a b c           ①
1       a
2       b
3       c
-1      lua
0       print-arg.lua
```

① The *print-arg.lua* script includes the following code: `for k, v in pairs(arg) do print(k, v) end`. The `arg` is a special predefined global variable of the table type that holds the command line arguments. The `for` loop is described later in the Statements chapter. The builtin `pairs` function is used throughout this book, but it is formally defined in the Iterators chapter.

2.2. Lua REPL

In the interactive mode, Lua processes the input *one line at a time.*

- If the given input line is an expression (or, an expression list) then Lua evaluates it and prints its value.
- Otherwise, Lua interprets the line as a statement and, if it is a complete statement, it is executed.
    - If the input is an incomplete statement, Lua changes the prompt, e.g., to `>>`, and it waits for its completion.

(Note that an expression that is not a statement by itself cannot be split over multiple lines in Lua REPL.)

You can change the primary and secondary prompts by using the Lua-defined global variables, `_PROMPT` and `_PROMPT2`, respectively.

```
$ lua
Lua 5.4.4  Copyright (C) 1994-2022 Lua.org, PUC-Rio
>                                          ①
> _PROMPT = "lua> "
lua> _PROMPT2 = "...> "                    ②
lua> for k,v in pairs(arg) do
...> print(k, v)                           ③
...> end
0       lua                                ④
lua>                                       ⑤
```

① Lua's default prompt, `> `.

② `lua> ` is a new primary prompt in this session.

③ Since the `for` statement in the previous line has not ended, Lua changes its prompt to the secondary prompt, `...> ` in this example, to indicate that fact.

④ The output.

⑤ Back to the primary prompt.

2.3. LuaJIT

There is another widely used version of Lua language runtime, which is implemented as a JIT (just in time) compiler:

- LuaJIT [https://luajit.org/luajit.html]

JIT generally means that the compilation is done at run time, not as a separate build step, and it compiles only the portions that are necessary, rather than the entire program. (The standard Java distribution, for example, uses JIT compilation.)

For differences between the interpreters and compilers, refer to other references, for instance, High-level programming language [https://en.wikipedia.org/wiki/High-level_programming_language].

Chapter 3. Lua Program Execution

If you are completely new to Lua, then you can skip, or skim through, this chapter and come back to it later.

3.1. Chunks

A "program", which is executed as a unit, is called a *chunk* in Lua. A chunk is a sequence of statements. A chunk is also syntactically a *block*, and it can include local variables.

A chunk can be stored in a file or in a string inside a host program. Chunks are implemented as anonymous functions in Lua. Chunks can therefore receive arguments (as a vararg argument (`...`)) and they can return values.

The example script used in the previous chapter with the `-e` flag, `print('Hello, World!')` is a complete chunk since it was executed *as a unit*. This one statement chunk is *more or less* equivalent to the following, when Lua executes the chunk:

```
function(...)                    ①
  print('Hello, World!')         ②
end()                            ③
```

① The arguments `...` are ignored in this simple example.

② The "real" script.

③ Lua calls this implicitly defined function. (Note the trailing `()`.)

Here's another example of a chunk, which has been saved as a file (named *hello.lua* in this example) and executed with the `lua` command.

hello.lua

```
local name = ...                    ①

if name then                        ②
  print("hello" .. name)            ③
else
  print("hello, whoever you are")
end

return 0                            ④
```

① The Lua code in a file runs as a chunk in the context of an implicitly defined anonymous function. Hence it accepts arguments, a "vararg". The vararg symbol `...` denotes a list of function arguments. See the later chapter on functions. The keyword `local` is used to declare a local name.

② The `if` statement. All values in the `if` condition are effectively true except for `false` and `nil`.

③ See below for more information on the `print` function.

④ The value, `0` in this example, is returned to the host program (or, the Lua interpreter). The `return` statement can be used in any block, not just in a function, unlike in many other programming languages (since all Lua chunks are always wrapped in anonymous functions before being executed).

3.2. Blocks

A block is a sequence of zero, one, or more statements, and they are executed in the given order, one after another. Lua is a "lexically scoped" programming language. Blocks can be nested, and they are used for scoping purposes, among other things.

A block can end with an optional `return` statement.

3.3. Block Defining Statements

A block is defined, in addition to a chunk,

- By the following statements:
  - The `do` statement:
    - `do <block> end`,
  - The `if` statement:
    - `if <exp> then <block> elseif <exp> then <block> else <block> end`,
  - The numerical `for` statement:
    - `for <name> = <exp list> do <block> end`,
  - The generic `for` statement:
    - `for <name list> in <exp list> do <block> end`,
  - The `while` statement:
    - `while <exp> do <block> end`,
  - The `repeat` statement:
    - `repeat <block> until <exp>`, and
- By function definitions:
  - The global function definition:
    - `function <func name> <func body>`, and
  - The local function definition:
    - `local function <func name> <func body>`, where
  - `<func name>` is optional, and
  - `<func body>` is
    - `( <parameter list> ) <block> end`.

3.4. The do Block Statement

A block can be explicitly delimited to produce a single statement, using the do statement. For instance,

```
do ; end                                    ①
```

① This `do - end` block is a statement, which in turn contains a single empty statement (`;`) in this particular example.

The `do - end` block can include any number of statements, and it can be used to control the scope of variable declarations.

3.5. Scoping

Lua is a lexically scoped language, meaning that the blocks are defined through the static code text, and not at run time, for instance.

The scope of a local variable begins at the first statement after its declaration and lasts until the last non-void statement of the innermost block that includes the declaration. For example,

```
local x = 0                                 ①
print(x)                                    ②
do                                          ③
  local x = x + 1                           ④
  print(x)                                  ⑤
end                                         ⑥
print(x)                                    ⑦
```

① A local variable `x` is declared here, whose scope starts from the next statement.

② The variable `x` is "in scope", and it has the value `0`, when the `print` function is called.

③ A new inner block begins here.

④ A new local variable `x` is declared within this inner block. At this point, the variable `x` on the right hand side still refers to the outer scope `x`. Hence the new `x` is initialized with `1` (= `0 + 1`).

⑤ This `print` function will print out the value of the inner variable, `1`. The outer scope `x` is "shadowed" in the inner block.

⑥ The inner block ends here.

⑦ The variable `x` in this line refers to that of the outer scope, whose value remains to be `0`.

3.6. Environments

A Lua chunk is compiled in the scope of an external local variable named `_ENV`. Any reference to a name that is not explicitly bound to a declaration, a *free name*, is syntactically equivalent to referring to a field with the same name defined in `_ENV`.

For instance,

```
x = 10                          ①
print(_ENV[x])                  ②
```

① This statement is equivalent to `_ENV.x = 10`.

② This will print the value `10`.

Any table used as the value of `_ENV` is called an *environment*. Lua keeps a global variable `_G`, whose value is called the *global environment*.

When Lua loads a chunk, the default value for its `_ENV` variable is the global environment. Therefore, by default, free names in Lua code refer to entries in the global environment, and they are called the *global variables*.

3.7. Lua's Global Variables

`_G` A global variable that holds the global environment.

`_VERSION` A global variable that references the Lua's version string.

```
> print(_VERSION)
Lua 5.4
```

3.8. The `print` Function

Lua's "basic library" includes a number of core functions in Lua, which are comparable to the "builtin functions" in other programming languages. The `print` function is probably the most commonly used function while developing Lua scripts.

```
print(···)                          ①
```

① This function signature indicates that `print` accepts a vararg argument `...` and returns no value.

The `print` function receives an arbitrary number of arguments and it prints their values to *stdout*, separated by single spaces. It is primarily used for debugging and diagnostic purposes. That is to say, Lua's `print` is more comparable to JavaScript's `console.log` rather than Python's `print` or `println`, for instance.

3.9. The `dofile` Function

Lua's standard library includes a number of functions to load (and, execute) external Lua code. The (more commonly used) `require` function is discussed in the next chapter on modules.

The builtin `dofile` function is used to load and execute a Lua chunk from a file,

```
dofile(filename = nil) -> ...      ①
```

① An informal "function signature" notation.

We will use this somewhat non-standard notation to represent Lua function signatures in this book. In this example, the function `dofile` is defined with a single *optional* parameter, `filename`. The equal sign `=` followed by a value represents the default value, `nil`, and hence this parameter is optional. Lua does not allow optional parameters or overloading of the user-defined functions. But, this kind of notation can still be useful in many circumstances.

This particular "function signature" also indicates that `dofile` returns (as indicated by `->`) an arbitrary number of values (as indicated by `...`). This notation is intended to augment what is described in the text, and it should not be taken too literally. When there is an ambiguity, refer to the surrounding text.

The `dofile` function takes one string argument `filename`, and it attempts to open the named file. If the file is successfully loaded, it tries to execute its content as a Lua chunk. When `filename == nil`, it uses the standard input (stdin). If the Lua chuck was successfully loaded and executed, then `dofile()` (or, `dofile(nil)`) returns all values returned by the chunk. In case of errors, it propagates the error to its caller. For example,

```
$ cat hullo.lua                          ①
print("h e l l o !")
```

```
$ lua
Lua 5.4.4  Copyright (C) 1994-2022 Lua.org, PUC-Rio
> dofile("hullo.lua")                    ②
h e l l o !
> dofile()                               ③
print("h e l l")                         ④
print("o o")
h e l l                                  ⑤
o o
```

① A sample script file.

② Calling `dofile` with this script executes the script.

③ Calling `dofile` with `nil` accepts the input from the stdin.

④ A two-line lua script which is directly typed into the terminal.

⑤ At this point, we inputted the EOF signal (although not displayed). The following two lines are the output from the script.

3.10. The `load` Function

The builtin `load` function belongs to two categories:

```
load(chunk: string,                      ①
     chunkname = "chunk",
     mode = "bt",
     env = nil)
    -> function | nil, error             ②
load(chunk: function,                    ③
     chunkname = "=(load)",
     mode = "bt",
     env = nil)
    -> function | nil, error
```

① The `chunk` parameter should be a string in this function signature. In this signature, the default value of the `chunkname` is `"chunk"`.

② The vertical bar `|` represents a choice, among alternative values. This function can return a single value of type `function` *or* two values, `nil` and a value that represents an "error".

③ This function takes a value of the `function` type as its function argument. In this case, the optional parameter `chunkname`'s default value is `"=(load)"`.

The `load` function loads the given argument `chunk` as a chunk. If `chunk` is a string, this string is loaded as a Lua chunk. If `chunk` is a function, then `load` calls it repeatedly to get the chunk pieces until it returns an empty string or `nil`, or the EOF signal. The returned chunk pieces must be strings, and they are concatenated in the calling/return order. The `chunkname` argument is used as the name of the chunk for error messages and debug information.

The chunk can be text or binary, e.g., a precompiled chunk. The `mode` argument can be set to

- `"t"` for text chunks,
- `"b"` for binary chunks, or
- `"bt"` for both binary and text.

If there are no syntactic errors in the loaded (and combined) script, the `load` function returns the compiled chunk as a function. Otherwise, it returns `nil` and the error message. If the `env` argument is set, then the first *upvalue* of the returned function is set to this value.

3.11. The `loadfile` Function

```
loadfile(filename = nil,
         mode = "bt",
         env = nil)
    -> function | nil, error       ①
```

① This function signature indicates that `loadfile` can be called with no arguments, for instance, since all parameters are optional.

The `loadfile` function works like `load`, but it loads the Lua code from a file named `filename`, if specified, or from the standard input otherwise. The difference between `dofile` and `loadfile` is that the `loadfile` function returns the chunk as a function rather than executing it.

For instance, using the same *hullo.lua* file as before,

```
> f = loadfile("hullo.lua")              ①
> f()                                    ②
h e l l o !                              ③
```

① Refer to the comment earlier about the chunk. A loaded chunk is implicitly wrapped in a function in Lua. That is, in this example, `f` is a function.

② And hence, it is callable.

③ The output from the line, `print("h e l l o !")`, in the *hullo.lua* script, as shown earlier.

Error-related system functions, including `error`, `warn`, and `assert`, are described at the end of the book, Error Handling.

As a general recommendation, if you are going through this book more or less from beginning to end, then you can skip most of the details at your first reading. These builtin and other library functions are listed in this book mainly for reference purposes, and they are not meant to be read and "learned" at once.

Chapter 4. Modules

Modules are a basic unit of code sharing and reuse in Lua. A Lua module is, by convention, a Lua source code file that (implicitly) returns a `table` containing other functions and variables. Modules provide namespaces, and they are used to organize code.

A global function `require`, from the `package` library, is used to load modules in Lua code.

4.1. The `require` Function

```
require(modname) -> ...        ①
```

① As illustrates earlier, this function signature notation (informally) indicates that the function `require` takes a single argument `modname` and it returns (`->`) an arbitrary number of values (`...`).

The `require` function loads a module (a Lua chunk) named `modname`, e.g., from a file named *modname.lua*. `require()` loads the same module no more than once, within the context of the executing chunk, even if it is specified multiple times, either directly or transitively.

Modules are searched from the Lua's `package.path` global variable (string), which holds all paths used by a Lua loader. In the standalone Lua interpreter, an environment variable `LUA_PATH`, for example, can be used to overwrite the value of `package.path`.

Likewise, C shared objects, e.g., DLLs, are searched from Lua's global `package.cpath` string variable, with the corresponding environment variable, `LUA_CPATH`, for instance.

4.2. Table-Based Modules

There are a few different ways to create a module in Lua. Here's a recommended way, e.g., using a Lua table. In the following example, we create a module named `arithmetic` in a file named *arithmetic.lua*.

arithmetic.lua

```
local arith = {}                ①

arith.factor = 10               ②

function arith.add(a, b)        ③
  return a + b
end

return arith                    ④
```

① We create a single empty table and assign it to a local variable (local to this "module").

② We add one variable, `factor`, to the table in this example. Note that `factor` is a table field variable.

③ We add one function to the table, `add`, which is also a table field variable. Functions are described in more detail later in the book. It should be noted that this syntax is equivalent to `arith.add = function(a, b) return a + b end`.

④ Returns this table. As indicated, the `return` statement can be used in any Lua chunk, not just in (explicitly declared) functions.

As illustrated in this example, a module consists of roughly three "parts". That is, to create a module,

- We start by creating a local table in a file named <*modname*>*.lua*, for instance. (`{}` is a table initializer literal.) The local table name, e.g., `arith` in this example, is arbitrary.

- Next, we add all functions and variables to be "exported" to this table. Any local variables and function definitions that are not added to this table are, on the other hand, *local* to this module. The use of global variables in modules is not recommended.
- Finally, we `return` the table. A module, just like any other chunk in Lua, is automatically wrapped in an implicitly defined anonymous function. This return value is returned to the caller, the `require` function, which in turn returns this value to its own caller (e.g., another module).

The modules can be loaded using the `require` function, as shown above. In this module convention, the `require` function returns a single `table`, which is to be understood as a "module".

```
require(modname) -> table                ①
```

① In the table-based module convention, the module chunk should return one `table`.

For example, using the same `arithmetic` module example above, which returns a single table,

main.jua

```
local arith = require("arithmetic")      ①

local f = arith.factor                   ②
print("f =", f)                          ③

local sum = arith.add(2, 4)              ④
print("sum =", sum)                      ⑤
```

① We use the `require` function to "load a module" and give it a name `arith`. The `arith` local variable points to the table returned by the chunk from *arithmetic.lua*. In this example, *main.lua* and

arithmetic.lua are in the same folder. (In general, one can use a relative file path.) Otherwise, the *arithmetic.lua* file should be in a path specified in `package.path`, e.g., either by explicitly setting the path in the program or by using the environment variable, `LUA_PATH`.

② We can access the table field, `factor`. from the module `arithmetic`. But, since we assigned the returned table to a local variable `arith`, we now reference it as `arith.factor` in this example.

③ The output will be *f = 10*.

④ `add` is also syntactically a field of the table referenced as `arith`. Hence we call it as `arith.add()`.

⑤ This will print out, *sum = 6*.

The `package` library also contains a number of other functions defined in the `package` table, which can be used to load, or otherwise manipulate, modules. These functions are not included in this book.

Lua modules can be shared via services like *LuaRocks*. For more information, refer to their respective official websites. As for *luaRocks*, it is the de-facto standard package repository service for the Lua dev community:

LuaRocks

The Lua package manager [https://luarocks.org/]

Chapter 5. Lexical Elements

5.1. White Spaces

Lua is a free-form language. Lua recognizes the standard ASCII whitespace characters as spaces in source code:

- Space,
- Form feed,
- Newline,
- Carriage return,
- Horizontal tab, and
- Vertical tab.

Lua ignores spaces (and comments) between tokens, except as delimiters. In general, newlines and indentations, and other particular formatting of the source code, have no significance in Lua, unlike in some other programming languages (e.g., most notably, Python).

5.2. Comments

There are two kinds of comments in Lua. A comment that starts with a sequence of consecutive characters comprising two hyphens and an opening long bracket (e.g., `--[[[[` with two or more opening square brackets) is a *long comment*, and it continues until the *matching* closing long bracket (e.g., `]]]]`).

```
--[[                                ①
     I'm a long comment.
]]                                  ②
```

① The long comment starts here.

② And, it ends here.

On the other hand, if `--` is not immediately followed by an opening long bracket, then it starts a *short comment*. A short comment continues until the end of the line.

```
-- This is a short comment          ①
```

① The comment starts from `--` and it ends at the newline.

Comments cannot start within a string literal.

5.3. Names

Names, or identifiers, in Lua are used to denote

- Variables (including function names),
- Table fields (which are also table-scoped variables), and
- Labels.

Names can be any string of one or more Latin letters, digits, or underscores (_). They cannot start with a digit. The following are all valid names:

```
apple2manzana                       ①
_amazon_forest                      ②
micro_office_360                    ③
fakeFriendClub                      ④
```

① The name `apple2manzana` starts with an alphabet `a` and it contains all valid characters, and hence it is a valid name.

② A name that starts with an underscore _.

③ A name that follows the "snake name" convention.

④ A name that follows the "camel case" naming convention.

5.4. Keywords

A number of identifiers are reserved by Lua, and they cannot be used for other purposes. They are called the "keywords". There are 22 keywords in Lua, as of 5.4.

```
and       break     do        else      elseif    end
false     for       function  goto      if        in
local     nil       not       or        repeat    return
then      true      until     while
```

5.5. Operators

The following strings denote other tokens:

```
+     -     *     /     %     ^     #
&     ~     |     <<    >>    //
==    ~=    <=    >=    <     >     =
(     )     {     }     [     ]     ::
;     :     ,     .     ..    ...
```

These operators and other symbols are explained throughout this reference, e.g., in the Expressions chapter, in particular. Another important class of lexical elements, namely, the "literals", are described in the next chapter.

Chapter 6. Builtin Type Literals

6.1. Nil and Booleans

The tokens, `nil`, `true`, and `false` are constant literals. We go through the number and string literals in this chapter. Another important (non-constant) literal, the table constructor, is described later in the book.

6.2. Numerals

Numeric literals (or, numerals) are of the number type. There are two kinds of numeric literals in Lua, integers and floats. A numeral can be written with an optional fractional part with a radix point (`.`) and/or with an optional decimal exponent, marked by a letter `e` or `E`.

- A numeral with a radix point and/or an exponent denotes a float.
- Otherwise,
  - if its value fits in an integer of the Lua runtime (e.g., 32 or 64 bits), it denotes an integer.
  - Otherwise, it denotes a floating point number.

```
666                         ①
777.                        ②
888.888
999e-9
10000000000000000000        ③
```

① An integer numeral.

② These three tokens are all float numeral literals.

③ This will be treated as a float numeral by Lua (e.g., because it does not fit into a 64 bit signed integer).

Lua also accepts hexadecimal integer literals, which start with `0x` or `0X`. Hexadecimal literals also accept an optional fractional part plus an optional binary exponent, marked by a letter `p` or `P`. A hexadecimal numeral with neither a radix point nor an exponent always denotes an integer value.

```
0x10                    ①
0X10.                   ②
0x1p10                  ③
```

① An integer literal, `16`.

② A floating number literal, `16.0`.

③ A floating number literal, `1024.0`.

6.3. String Literals

The type of string literals is `string`. String literals in Lua can be defined using either a short format or long format.

6.3.1. Short string literals

A short literal string can be delimited by matching single or double quotes (e.g., `'abc'` or `"xyz"`). Lua does not support the character types. Short strings can contain the following C-like escape sequences:

`\"` (double quote), `\'` (single quote), `\\` (backslash), `\a` (bell), `\b` (backspace), `\f` (form feed), `\n` (newline), `\r` (carriage return), `\t` (horizontal tab), and `\v` (vertical tab).

In addition,

- A backslash followed by a line break results in a newline in the string.

- The escape sequence `\z` skips the following span of whitespace characters, including line breaks.

We can specify any byte in a short literal string by its numeric value. This can be done with

- The escape sequence `\xXX`, where `XX` is a sequence of two hexadecimal digits, or
- The escape sequence `\ddd`, where `ddd` is a sequence of one, two, or three decimal digits.

The null character `\0`, for instance, can be inserted this way. Furthermore, the UTF-8 encoding of a Unicode character can be inserted in a short literal string with

- The escape sequence `\u{XXX}`, where `XXX` is a sequence of one or more hexadecimal digits representing Unicode character code point.

6.3.2. Long string literals

Literal strings can also be defined using a long format enclosed in long brackets. We define an opening long bracket of level `n` as

- An opening square bracket (`[`), followed by
- `n` equal signs (`=`) with `n >= 1`, followed by
- Another opening square bracket (`[`).

A closing long bracket is defined similarly, with the opening brackets replaced by the closing brackets (`]`).

A long string literal starts with an opening long bracket of any level and ends at the first closing long bracket of the same level. Long string literals can contain any text except a closing long bracket of the same level. They can span several lines, and they are sometimes informally called the multiline string literals.

Long strings do not interpret any escape sequences in them. A long string literal can include other long string literals of *different levels*. Note that, despite what the term may imply, there is no hierarchical relationship among the long strings of different `levels`.

For example,

```
[[Hello]]                       ①
[=[World?]=]                    ②
[====[!Hola, mundo!]====]       ③
[==[
Roses are red, and
Violets are blue.]==]           ④
```

① A long literal string of level `0`.

② A long literal string of level `1`.

③ A long literal string of level `4`.

④ A multi-line string. The newline right after the opening long bracket is ignored by Lua, and it is not part of the string.

Note that multiline long string literals cannot be evaluated directly in the Lua REPL. For instance,

```
> [[I'm fine. Really]]
I'm fine. Really
> [[I'm
>> not :(
>> ]]
stdin:3: unexpected symbol near '[[I'm
not :(
]]'
```

An expression, which is not part of a statement, cannot span multiple lines in the REPL.

Chapter 7. Types

7.1. Values and Types

Lua is a dynamically typed language. *Values*, or "objects" as they are typically called in some other programming languages, are the fundamental entities in Lua. Values are associated with *types*, and these types are checked at run time.

The type of a value determines whether the value is valid and what kind of operations are allowed on that value, etc. All values in Lua belong to one of the following eight basic types:

- `nil`,
- `boolean`,
- `number`,
- `string`,
- `function`,
- `userdata`,
- `thread`, and
- `table`.

The values of the last 4 types, `functions`, `userdata`, `threads`, and `tables` are "mutable", that is, their values can change during the execution of the program. In contrast, the `nil`, `boolean`, and `number` values as well as `strings` do not change once they are created.

Variables in Lua do not directly contain, or hold, the values of these types. They are merely names of, or references to, these values. Assignment, parameter passing, and function returns, for example, always manipulate the references to such values. These operations do

not imply any kind of value copy or duplication. Only the variables, or the references, are copied or otherwise manipulated.

In practice, for the values of the immutable types like `nil`, `true`, `false`, strings, and integers and float numbers, however, there is little difference whether we deal with the values themselves or their references. Furthermore, functions generally do not keep states, and we do not deal with `userdata` in this reference, and hence it is only the tables and threads that we need to pay special attention to.

7.2. The `type` Function

A builtin function `type` can be used to query the type of a value. It has the following signature:

```
type(v) -> string                    ①
```

① This notation indicates, as throughout this book, that the function `type` takes one argument (arbitrarily named as `v` in this example), and it returns a value of the `string` type.

Note that we do not generally indicate the types of the function parameters in this notation. Many Lua functions can take values of multiple types (or, the union types), and when the arguments of improper types are given, they simply throw errors or return `nil`. This particular function `type`, for example, accepts a value of *any* type.

The `type` function returns the type of its argument `v`, *coded as a string*. The possible results of this function are

- `"nil"`, `"boolean"`, `"number"`, `"string"`, and
- `"function"`, `"userdata"`, `"thread"`, `"table"`,

corresponding to each of the 8 immutable/mutable types, respectively.

For example,

```
$ lua                                ①
Lua 5.4.4  Copyright (C) 1994-2022 Lua.org, PUC-Rio
> type("hello")                      ②
string                               ③
> type(3.5)                          ④
number
> type(function() end)               ⑤
function
> type({})                           ⑥
table
```

① We start a Lua interpreter by typing a command *lua*, or something comparable (depending on your system and the Lua setup).

② The prompt `>` indicates that we are in the Lua REPL. We call the `type` function with an argument `"hello"`, a value of the `string` type.

③ This function call returns the string `string`, as expected.

④ The type of a value `3.5` is `number`.

⑤ A function in Lua can be defined with the keyword `function` followed by the function body. The type of this (empty) anonymous function is `function`.

⑥ The literal `{}` is an expression that creates a new empty table, whose type is `table`.

7.3. Nil

The type `nil` has a single unique value, `nil`. It is often used to represent the absence of a (useful) value, or even an "invalid value". Or, to indicate a failure condition. The `nil` value is distinguishable from any other values in Lua, including `false`.

7.4. Boolean

The `boolean` type has two valid values, `true` and `false`, which represent the logical states, true and false, respectively. In a boolean condition, only `nil` and `false` evaluate to false in Lua. All other values in the given expression correspond to the logical true in Lua. This includes `0`, `0.0`, and an empty string `""`.

7.5. Number

The type `number` consists of two subtypes, integer and float, representing integer numbers and real floating-point numbers, respectively. The standard Lua generally uses 64-bit signed integers and double-precision (64-bit) floats. This is, however, an implementation detail.

Lua often treats the integer and float subtypes as same, and automatically converts between them as necessary in many situations. Hence, we can mostly ignore the differences and just handle any numbers as the `number` type (e.g., regardless of their internal representations).

In Lua, there are no "integer overflow errors". In cases where a value is not representable by an integer, it may be automatically converted to float. Or, in pure integer expressions such as bitwise operations, an overflow of an integer result may wrap around, using the two's complement arithmetic.

For example,

```
> 2000000000000000000              ①
2000000000000000000
> 20000000000000000000             ②
2e+19
```

① An integer.

② A number that cannot be represented with a 64-bit signed integer (in the 64 bit build of the standard Lua) is evaluated to a float.

7.5.1. The `tonumber` function

```
tonumber(e, base = 10) -> number   ①
```

① This notation indicates that `tonumber` takes one or two arguments, named `e` and `base`, and it returns a number value. If the function is called with one argument, the default value of `10` is used for the second argument, `base`.

The builtin `tonumber` function converts its number or string argument `e` to a number.

- If `e` is already a number, it returns the same number.
- If `e` is a string convertible to number, it returns this number. The conversion of strings can result in integers or floats.
- In all other cases, it returns `nil` indicating a failure.

When `e` is a string value, an optional second argument `base` of an integer type, between 2 and 36, can be provided. In that case, `e` is to be interpreted as an integer numeral in that base. If the conversion fails, it returns `nil`.

For example,

```
> tonumber(33)                    ①
33
> tonumber(true)                  ②
nil
> tonumber('33.33')               ③
33.33
```

```
> tonumber('33', 32)                ④
99
> tonumber("1001001001", 2)         ⑤
585
> tonumber('33', 2)                 ⑥
nil
```

① Calling `tonumber` with an integer or float number returns that number.

② Calling `tonumber` with an argument other than a number or string returns `nil`.

③ The string argument is converted to the corresponding integer or float number.

④ When the second argument, an integer between 2 and 36, is provided, the string argument is to be interpreted as an integer number in the specified `base`. The letters `a`/`A` through `z`/`Z` are used to represent the digits corresponding to 10 through 36.

⑤ The binary number `1001001001` is equivalent to `585` in the decimal representation.

⑥ `33` is an invalid number representation in base 2, and hence it returns `nil`.

7.6. String

The type `string` represents a sequence of bytes, each of which can contain any 8-bit value from `\0` to `\255` similar to the C strings. Unlike in the null-terminated C strings, however, the null byte `\0` is a valid element in Lua strings. Lua does not have a builtin concept of characters. Nor does it understand Unicode. In fact, Lua strings are encoding-agnostic.

Strings in Lua are immutable, as in many other programming languages.

The length of a string can be computed using the length operator (#), which is simply the number of bytes in the string, and it cannot be bigger than the maximum integer representable in a given implementation.

7.6.1. The `tostring` function

```
tostring(v) -> string
```

The `tostring` function converts an argument `v`, of any type, to a string.

- If an argument `v` is a string, it simply returns the same value.
- Otherwise,
  - If the metatable of `v` has a `__tostring` metamethod, then `tostring()` calls this metamethod with `v` as argument.
  - Otherwise,
    - If the metatable of `v` has a `__name` field with a string value, `tostring()` uses this string.
    - Otherwise, it returns an internal string representation of `v`.

```
> a = {}                              ①
> a = setmetatable(a, {__tostring = function() return "ha ha"
end})
> tostring(a)                         ②
ha ha
> b = {}
> b = setmetatable(b, {__name = "hoho"})
> tostring(b)                         ③
hoho: 0x5db5361980
> c = {}
> tostring(c)                         ④
table: 0x579bd78450
```

① The global variable `a` refers to a newly created empty table.

② `a` has `__tostring`, and hence `tostring(a)` calls this method.

③ `b` has a field `__name` in its metatable, and hence its value is used in the table's string representation.

④ `tostring(c)` simply returns `c`'s internal string representation.

Metatables and various metamethods are further explained later.

7.7. Functions

Lua can call (and manipulate) functions written in Lua as well as those written in C. Functions are represented by the type `function`. All lua functions are anonymous, and they are described in more detail in a later chapter, Anonymous Functions.

7.8. Userdata

Lua includes the type `userdata` to allow arbitrary C data to be stored in Lua variables. A `userdata` value represents a block of raw memory. Userdata values can only be created or modified through the C API. There are two kinds of `userdata`:

Full userdata	A value with block of memory managed by Lua.
Light userdata	A C pointer value.

We do not discuss the C program interface, and hence we will not use the `userdata` type in this book.

7.9. Thread

Objects of the type `thread` represent independent threads of execution and they are used for managing coroutines in Lua. Threads will be

further discussed in the concurrency chapter. Note that Lua threads are purely language constructs that are unrelated to the native threads that may or may not be supported in the hosting environments, e.g., the operating systems or the host programs running on those operating systems.

7.10. Table

A `table` in Lua lets you create a "custom object", or even a "custom type". An object is essentially a collection of zero, one, or more properties, or fields. A field is a pair of a name and its value. The value of a field can be of the `function` type, in which case the field is often called the method.

A Lua `table` is also used to create data structures like arrays (or, lists, sequences, etc.) and maps (or, dictionaries, hashtables, or associative arrays, etc.). An array is a (one-dimensional) sequence of values. A map is a collection of zero, one, or more fields, or key-value pairs.

In Lua, any values of any type, other than the two values `nil` and `NaN` (a floating point number representing "not a number"), can be used as keys, or field names, in a `table`. Any values other than `nil` can be used as the values of the table fields. The `nil` value indicates the absence of a value.

A `table` can also used to represent linear data structures like arrays or lists, using the (implicitly defined) integer keys. Any item in a table which does not have a key is given an integer key (or, index) starting from `1` and sequentially increasing to the left (skipping the fields with explicit names).

The values of field variables can be accessed using the so-called subscription or "index notation". For instance, given a table `t` which references `{a = true, b = "hi"}`, we can refer to the value of the first element as `t["a"]` and the second element as `t["b"]`.

When the type of a key/name is a string value or a variable, the property or "dot notation" can also be used for these field variables. That is, `t.a` and `t.b` are syntactically equivalent to `t["a"]` and `t["b"]`, respectively, using the same example.

Using another example,

```
> a = {name = "apple", taste = "sweet"}
> b = {"orange", "sour"}
> c = {name = "pear", "pineapple", taste = "salty", "bitter"}
```

The field `name` of `a`, for instance, can be referenced as either `a.name` or `a["name"]`. The two fields of `b` can be referenced with indexes, e.g., `b[1]` and `b[2]`. Likewise, the fields of `c` can be referenced with the names or indexes.

```
> a.name, a["name"], a.taste, a["taste"]
apple   apple   sweet   sweet
> b[1], b[2]
orange  sour
> c.name, c[1], c["taste"], c[2]
pear    pineapple       salty   bitter
```

Tables are the most versatile construct in Lua, and various uses of Lua tables are described in more detail throughout this reference. We can use tables as "classes" as well for object-oriented programming (OOP).

Chapter 8. Variables

In Lua, variables are simply names that refer to values. Before the first assignment to a variable, its value is `nil`. There are three kinds of variables in Lua:

- Local variables,
- Table fields, and
- Global variables.

8.1. Local Variables

Local variables are declared with the keyword `local`. Local variables can be declared anywhere, inside or outside an implicit of explicit block. The declaration can include an initialization. If present, an initial assignment has the same semantics of the multiple assignment statement. Otherwise, all local variables are initialized with `nil`.

For example,

```
> do                                    ①
>> local x                              ②
>> print(x)                             ③
>> end
nil
> do
>> local y = 100                        ④
>> print(y)                             ⑤
>> end
100
> print(y)                              ⑥
nil
```

① The `do` statement in Lua creates a block.

② Declaring a variable `x` with no explicit initial value within this block.

③ Its initial value is `nil` by default.

④ A local variable declaration with an initial value.

⑤ The value of `y` at this point is `100`.

⑥ The local variable `y` is no longer available outside the block.

Local variables are statically and lexically scoped. Local variables can be freely accessed by functions defined inside their scope. Note that the local variables are not generally useful in the global scope in the Lua REPL since each statement, including the `local` declaration, is executed in its own separate chunk.

Multiple variables can be declared in a single statement, with or without initial values.

```
> local x, y, z print(x, y, z)                ①
nil     nil     nil
> local x, y = 1, 2 x, y = y, x print(x, y)    ②
2       1
```

① It declares three local variables, `x`, `y`, and `z`. Their initial values are all `nil`. This line includes two statements, for illustration. They could have been separated by a semicolon statement.

② It first declares two local variables, `x` and `y`, with initial values `1` and `2`, respectively. Next, the values are swapped (`x,y = y,x`). We then print out the result. This is all done in one line in REPL, as an example.

8.2. Attributes

Each variable name in a local variable declaration may be postfixed by an "attribute", e.g., a name between angular brackets `<>`. This is a new

introduction to the language, as of 5.4.4. There are currently two predefined attributes, `<const>` and `<close>`.

8.2.1. The `<const>` attribute

A local variable declared with `<const>` is a constant variable, that is, a variable that cannot be re-assigned to after its initialization.

```
> do                                    ①
>> local x <const>, y = 1, 2            ②
>> y = "hello"                          ③
>> x = 10                               ④
stdin:4: attempt to assign to const variable 'x'
```

① Starting a new block in REPL, for illustration.

② `x` is a constant variable, whereas `y` is not.

③ `y` can be assigned a different value. `y` now refers to a string value, `"hello"`.

④ On the other hand, attempting to assign a new value to the constant variable `x` throws an error.

Note that the variables in Lua do not have types, unlike in the statically typed programming languages. (Only the values in Lua are associated with types.) Hence, the same (non-constant) variable can be used to reference the values of different types, as in the case of `y` in this example.

8.2.2. The `<close>` attribute

The `<close>` attribute can only be used with the tables which have the metamethod, `__close`. A local variable declared as `<close>` is a to-be-closed variable. When the variable goes out of scope, Lua automatically calls its `__close` metamethod. This metamethod can be used for local block cleanup. For instance,

```
> mt = { __close = function() print("Closing...") end }
> obj = setmetatable({}, mt)       ①
> do                               ②
>> local o <close> = obj           ③
>> end                             ④
Closing...                         ⑤
```

① The (global) variable `obj` references a table that has an associated `__close` metamethod.

② The start of a local block.

③ The variable `o` is a to-be-closed variable referencing the same table referred to by `obj`.

④ The end of the local block. All local variables, such as `o`, go out of scope at this point.

⑤ The `__close` metamethods of all to-be-closed variables are automatically called. The local variable `o`'s `__close` metamethod in this example simply prints out a string *Closing...*

A variable declaration can contain at most one to-be-closed variable.

8.3. Table Fields

Lua's `table` is explained in the `table` chapter later in the book. The names of the fields in the tables are also variables. The field variable access syntax is discussed in the previous chapter on types.

8.4. Global Variables

A new variable name in a program is global by default unless it is first declared as `local`. A global variable can be declared without using the `local` keyword. Global variables can be also used before they are explicitly declared. For example,

```
> f = function() return A end      ①
> f()                              ②
nil
> A = 3                            ③
> f()                              ④
3
> A = 5                            ⑤
> f()                              ⑥
5
```

① The variable `A` is global.

② The default value of a global variable is `nil`.

③ We explicitly assign a value, `3`, to this global variable `A` here.

④ This statement will print out `3`.

⑤ We update the value of `A` to `5`.

⑥ This will print out `5` now.

All global variables are also the table fields of Lua's builtin global variable `_G`, which is the default environment `_ENV` when a Lua chunk is loaded. Hence, a global variable `x` can be referred to as `_G.x` or `_ENV.x`. For example,

```
> a = "hello"                      ①
> a, _G.a, _ENV.a                  ②
hello   hello   hello
```

① A global variable declaration with an initial value, `"hello"`.

② At this point, `a`, `_G.a`, and `_ENV.a` all reference the same string, `"hello"`.

Chapter 9. Expressions

An expression specifies how to compute a value from operands, e.g., by applying functions or other operators. It returns the computed value.

9.1. Basic Expression Types

An expression in Lua comprises the following basic expressions. Some of them are further explained in this chapter, and some of them later in the book.

9.1.1. Parenthesis expressions

A parenthesis expression, e.g., with another expression enclosed in parentheses (`()`), results in at most one value regardless of the value of the enclosed expression. For example, `(f(x,y,z))` is always a single value, even if `f` returns more than one value. In this case, the value of `(f(x,y,z))` is the first value returned by `f` or `nil` if `f` does not return any values.

Parenthesis expressions are often used, to increase the readability, or when a given expression needs to be evaluated differently than using the default operator precedence rules.

9.1.2. Literal expressions

The following literals are constant expressions:

`nil`	The `nil` literal.
`true` and `false`	Boolean literals.
Numbers	Numeral literals are explained in Numerals.
Strings	String literals are explained in String Literals.

9.1.3. Variables

Variables are basic expressions. Variables are explained in the earlier chapter, Variables.

9.1.4. Vararg expressions

Vararg expressions `...` can only be used inside a vararg function.

9.1.5. Function calls

Function calls are explained in in the later chapter, Anonymous Functions. Function calls can be used as both expressions and statements. If a function call is used as a standalone statement, then it discards all returned values after evaluating the function call. Both function calls and vararg expressions can result in multiple values, in general.

9.1.6. Function definitions

An anonymous function definition is an expression in Lua. Function definitions are explained in the Functions chapter.

9.1.7. Table constructor literals

Table constructors `{}` are (non-constant) literal expressions. They are explained later in the Tables chapter.

9.1.8. Operators

Lua also includes most of the operators that are commonly found in other programming languages. An operator takes an expression or two and returns another expression. Operators can be viewed as a special kind of functions in Lua (e.g., with special syntax, etc.). All Lua operators are described in this chapter.

Unary operator expressions

Lua unary operators comprise the unary minus (`-`), the unary bitwise NOT (`NOT`), the unary logical not (`!`), and the unary length operator (`#`).

Binary operator expressions

Lua binary operators comprise arithmetic operators, bitwise operators, relational operators, logical operators, and the concatenation operator (`..`).

9.2. Arithmetic Operators

Lua supports the following arithmetic operators:

- `+` Addition.
- `-` Subtraction, Unary minus.
- `*` Multiplication.
- `/` Float division.
- `//` Floor division.
- `%` Modulo.
- `^` Exponentiation.

For most binary operators, if both operands are integers, then the operation is performed over integers and the result is an integer. Otherwise, if both operands are numbers, then they are converted to floats, and the result is a float following the standard floating point arithmetic.

Exponentiation `^` and float division `/` work differently. They always treat their operands as floating point numbers.

9.3. Bitwise Operators

Lua supports the following bitwise operators:

- `&` Bitwise AND.
- `|` Bitwise OR.
- `~` Bitwise exclusive OR.
- `>>` Right shift.
- `<<` Left shift.
- `~` Unary bitwise NOT.

All bitwise operations convert its operands to integers, and they operate on all bits of those integers. Their results are always integers.

The left shift `<<` operation removes the existing leftmost bit and fills it with the next left bit, and so forth. Right shift `>>` zero-fills the leftmost bit after the bit shift operation.

9.4. Relational Operators

Lua supports the usual relational operators, which return a boolean value, `true` or `false`:

- `==` Equality.
- `~=` Inequality.
- `<` Less than.
- `>` Greater than.

`<=` Less or equal.

`>=` Greater or equal.

9.4.1. Equality operator

Equality `==` first compares the type of its operands. If the types are different, then the result is `false`. Otherwise, the values of the operands are compared. Strings are equal if they have the same byte content. Numbers are equal if they denote the same mathematical value.

Tables, userdata, and threads are compared by reference. Two objects are considered equal if and only if they are the same object.

A function is always equal to itself. Functions with any significant difference (e.g., different behavior, different definition) are always different. Two functions with the same behavior, created separately, may or may not be equal to each other.

One can customize the way that Lua compares `tables` (and `userdata`) by implementing a custom `__eq` metamethod.

9.4.2. Inequality operator

The operator `~=` is exactly the negation of equality `==`. Note that this operator is different from the C-style `!=`.

9.4.3. Comparison operators

The order operators, `<` and `<=`, work as follows.

- If both arguments are numbers, then they are compared according to their mathematical values, regardless of whether either or both are integer or floating point number types.

- Otherwise,
  - If both arguments are strings, then their values are compared according to the current locale.
  - Otherwise, Lua attempts to call the `__lt` or `__le` metamethods, if either metamethod defined in either of the operands.

Greater-than or greater-equal comparisons, `a > b` and `a >= b`, are translated to `b < a` and `b <= a`, respectively.

9.5. Logical Operators

The logical operators in Lua are `not` (unary) and `and` and `or` (binary). Logical operators consider

- Both `false` and `nil` as `false`, and
- Anything else as `true`.

The second operand in a binary `and` or binary `or` expression is conditionally evaluated:

```
<exp1> and <exp2>                ①
<exp1> or <exp2>                 ②
```

① `<exp2>` is evaluated only if `<exp1>` is `true`. Otherwise, it return the value of `<exp1>` without evaluating `<exp2>`.

② `<exp2>` is evaluated only if `<exp1>` is `false`. Otherwise, it return `<exp1>` without evaluating `<exp2>`.

This is known as the "short circuiting".

Note that the conjunction (`and`) and disjunction (`or`) operators do not necessarily return boolean values, depending on the actual types of `<exp1>` or `<exp2>`. On the other hand, the negation operator (`not`)

always returns boolean `false` or `true` regardless of the type of its operand.

9.6. The Length Operator

Lua supports a unary prefix operator `#`, which returns the length of the given operand. Note that the length is defined only for the values of the `string` and `table` types.

9.6.1. String length

When the operand is a `string`, it returns the number of bytes.

9.6.2. Table borders

A "border" in a table `t` is an integer satisfying the following condition:

- If a field at index `1` is absent in `t`, the border is `0`.
- Otherwise,
  - If there is a positive integer index followed by an absent next index, then the border is this positive integer index.
  - If an index with the maximum value for an integer is present, then the border is the maximum integer.

Note that the borders are determined by positive integer indexes only. Non-positive integer indexes and string keys do not play a role when computing the borders.

9.6.3. Table length

When the operand is a `table`, the length operator `#` returns a border in that table. Note that a table can have more than one border, and the Lua language does not specify which one to return from the length operator. It is implementation-dependent.

For example,

```
> #{}                              ①
0
> a = { x = 3, y = 5 }             ②
> #a                               ③
0
> b = {}
> b[3], b[4] = 'x', 'y'
> #b                               ④
0
> c = {}
> c[1], c[2], c[5] = 'p', 'q', 'r'
> #c                               ⑤
2
```

① The length of an empty table is `0`.

② The table constructor literal syntax is described in the `table` chapter.

③ The `table a` does not contain a field with index `1`, and hence its length is `0`.

④ Ditto with `b`.

⑤ The `table c` has indices `1` and `2`, but not `3`. Likewise, there is `5` but not `6`. Hence there are two borders, `2` and `5`. In this example, the length expression `#c` happens to return `2`.

9.6.4. Sequence length

A table with exactly one border is called a sequence. For example,

```
{}                                 ①
{10, 20, 30, 40}                   ②
{'a', 'b', nil, 'd'}               ③
```

① An empty table is a sequence of border `0`.

② This table is a sequence since it has one border, `4`.

③ This is not a sequence since it has two borders, 2 and 4. The `nil` value at index 3 is called a hole.

When a table `t` is a sequence, `#t` returns its only border, which corresponds to the intuitive notion of the length of a sequence. For instance, `#{10, 20, 30, 40}`, is `4` since this sequence has "4 elements".

9.6.5. Customizing the length operator

The behavior of the length operator `#` can be modified by implementing a `__len` metamethod. Note, however, that the values of the `string` type do not use this metamethod.

9.7. Operator Precedence

Operator precedence in Lua follows the table below, from lower to higher priority:

```
or
and
<    >    <=    >=    ~=    ==
|
~
&
<<    >>
..
+    -
*    /    //    %
unary operators (not   #    -    ~)
^
```

As usual, you can use parentheses to change the precedences of expressions.

The concatenation `..` and exponentiation `^` operators are right associative. All other binary operators are left associative. For example,

```
> 10 / 2 * 5                    ①
25.0
> 2 ^ 3 ^ 2                     ②
512.0
```

① This expression is equivalent to `(10 / 2) * 5`.

② This expression is equivalent to `2 ^ (3 ^ 2)`.

9.8. Conversions

Lua provides some implicit conversions between some builtin types at run time.

- Bitwise operators always convert float operands to integers.
- Exponentiation and float division always convert integer operands to floats.
- All other arithmetic operations applied to mixed numbers (integers and floats) convert the integer operand to a float.
- String concatenation automatically converts numbers to strings.

Chapter 10. Anonymous Functions

Functions are an essential component of any procedural programming languages, including Lua. A function in Lua is just a *value* that is "callable". Lua functions can return multiple values.

10.1. Function Definitions

A function can be defined as follows:

```
function (<parameter list>) <block> end
```

It starts with the keyword `function` and ends with `end`. A comma-separated list of zero, one, or more function parameters is included in parentheses following `function`. The function body `<block>` can comprise zero, one, or more statements. A function definition is a value, or more generally, an expression. For example,

```
> function(a, b) return a + b end
function: 0x55b1dbd83010
```

A function defined this way has no name. That is, it is anonymous. Functions are often assigned to variables, and we use the variables throughout the program. For instance,

```
product = function(a, b)          ①
  return a * b
end
p = product(2, 5)                 ②
```

① The anonymous function (a value) is assigned to a (global) variable `product` in this example.

② Then you can call this function using this name. The value of `p` is `10`.

Lua has a syntactic shortcut for this kind of use cases. A function name can be included after the `function` keyword and before the parenthesized parameter list. For example,

```
function sum(a, b) return a + b end
```

This function definition is equivalent to

```
sum = function(a, b) return a + b end
```

Likewise, the following two statements are more or less equivalent as far as the rest of the program is concerned:

```
local function product(a, b) return a * b end
local product = function(a, b) return a * b end
```

More precisely, the named `local function` definition, e.g., that on the first line, is equivalent to the following two (or, three) statements:

```
local product; product = function(a, b) return a * b end
```

This distinction is not generally significant, however, unless the function definition includes references to itself (e.g., as in recursive functions). One thing to note is that, although a function is just a value in Lua, the more traditional named function definition syntax is more commonly used in Lua programs, likely because that is what many programmers are used to.

10.2. Method Syntax

When a function is assigned to a variable with at least one dot (`.`), the last dot can be replaced with a colon (`:`) when using the named function definition form, while adding an *implicit* extra parameter `self` to the function, to emulate the method syntax. For example, the following two function definitions are equivalent to each other.

```
function x:exlen(factor) return #self * factor end
x.exlen = function(self, factor) return #self * factor end
```

10.3. Function Parameters

The parameter list can be a vararg parameter (`...`), or a named parameter list followed by an optional vararg parameter. For example,

```
function(...) end                   ①
function(a, b, c) end               ②
function(x, y, ...) end             ③
```

① A vararg parameter only.

② A list of named parameters only.

③ Named parameters followed by a trailing vararg parameter.

Parameters are local variables within the function body. Their scopes are limited to this function body block. The matching arguments, if any, are the initial values for the corresponding parameters.

10.3.1. Vararg parameter

A function that includes a vararg parameter is called the vararg function. When a function is called, it matches the list of arguments to

the list of parameters, from left to right.

- If there are more parameters than the provided arguments, then
  - All remaining parameters are initialized with `nil`.
- Otherwise, if more arguments are provided than the parameters, then
  - If there is a vararg parameter, then All excess arguments are collected into a list and it is assigned to the vararg parameter,
  - Otherwise, The excess arguments are ignored.

10.4. Function Calls

A function call is an expression, which can also be used as a standalone statement. When a function call is used as a statement, its value (e.g., the return values from the function) is discarded.

The function call expression in Lua consists of two parts, the prefix expression, which can be simply an anonymous function value or a function name, and the argument list.

- First, both the prefix expression and all arguments are evaluated.
- Next,
  - If the prefix expression is of the `function` type, it is called with the evaluated argument list.
  - Otherwise, if the prefix expression has a `__call` metamethod, then it is called with an augmented argument list with the value of the prefix expression as the first argument.

10.4.1. Arguments

The function call argument, a list of zero, one, or more comma-separated expressions, is enclosed in a pair of parentheses. In case the

argument is a single (short or long) string literal or a table constructor literal, the parentheses can be omitted. For example,

```
prefexp1 (10, 20, "hello")          ①
prefexp2 "universe"                 ②
prefexp3 [[ wide universe ]]        ③
prefexp4 { key = 333 }              ④
```

① The space between the function prefix expression and the argument list is not significant, in general.

② This function call expression is equivalent to `prefexp2('universe')`.

③ This is equivalent to `prefexp3('wide universe')`.

④ This is equivalent to `prefexp4({ key = 333 })`.

10.5. Method Call Syntax

When a colon `:` is used in place of the last dot `.` in a function call, the function call prefix expression is implicitly provided as the first argument. For example, the following two function call expressions are more or less equivalent to each other.

```
obj1.obj2:exlen(factor)
obj1.obj2.exlen(obj1.obj2, factor)
```

Note that `obj1.obj2` is evaluated only once in either syntax.

Chapter 11. Statements

Lua supports most of the conventional set of statements, similar to those found in other imperative languages.

- Empty statements (`;`),
- Multiple assignments,
- Label statements,
- `goto` statements,
- `break` statements,
- `return` statements,
- `if` statements,
- `for` statements,
- `while` statements, and
- `repeat` statements.

The `do` block statement was described earlier, in the context of the Lua program structure. Local variable declarations are also statements. They are syntactically similar to the assignment statements, but they primarily introduce the (new) variables to the local scope, with optional initial values. Function definitions (local or otherwise) and function call expressions (including method calls) can also be used syntactically as statements.

11.1. Empty Statements

The empty statement does nothing.

```
;                                                    ①
```

① In Lua, the semicolon `;` is a statement. In other C-style languages, semicolons are typically used to terminate a statement.

Empty statements are primarily used as placeholders (e.g., similar to `pass` in Python) or to separate other statements.

```
> print("hello"); print("world")   ①
hello
world
```

① This line has three statements. In this particular example, the middle empty statement is superfluous.

11.2. Multiple Assignments

The assignment statement specifies a list of one or more variables, on the left hand side of the assignment operator `=`, and it assigns a value to each variable by specifying a list of expressions, on the right hand side. The elements in both lists are separated by commas.

```
a, b = 1, 10
```

The assignment proceeds as follows:

1. All expressions on the right hand side are evaluated first.
2. If any expression contains a function call, then all values returned by that call is inserted into the expression list.
3. The list of values is adjusted to the length of the list of variables.
    - If there are more variables than the values on the right hand side, then the value list is extended with `nil`'s.
    - If there are fewer variables, then the excess values are discarded.

4. Each of the values on the right right hand side are assigned to the corresponding variable, from left to right.

11.3. Label Statements

Labels in Lua are syntactically statements. Similar to the empty statements, label statements perform no actions. These two types of statements are called void statements. For example,

```
::name1::                          ①
```

① This label statement introduces a new name, `name1`, to the scope.

A label is visible in the entire block where it is defined, except inside nested functions.

11.4. Goto Statements

The `goto <name>` statement transfers the program control to the label statement with the given `<name>`.

```
goto name1                         ①
```

① This `goto` statement transfers control to the label with `name1`.

A `goto` may jump to any visible label as long as it does not enter into the scope of a local variable.

11.5. Break Statements

The `break` statement terminates execution of the innermost enclosing `for`, `while`, or `repeat` statement. It effectively transfers the program control to the next statement after the loop.

11.6. Return Statements

The `return` statement is generally used to return values from a function or a chunk (which is handled as an anonymous function). In fact, any block in Lua (explicit or implicit) can end with a `return` statement.

Lua functions can return more than one value, so the syntax for the `return` statement is

```
return <expression list>
```

11.7. If Statements

The `if` statement consists of an `if-then` block, zero, one, or more `elseif-then` block, and an optional `else` block. For instance,

```
if <exp> then                    ①
    <block>
elseif <exp> then                ②
    <block>
elseif <exp> then
    <block>
else                             ③
    <block>
end                              ④
```

① The `if` statement starts with the `if` block. (Note that Lua is a free-form language and this particular formatting style is just conventional, as with any other example in this book.)

② Any number of `elseif` blocks can be included.

③ Zero or one `else` block.

④ The `if` statement ends with the keyword `end`.

The condition expressions (e.g., `<exp>` in the above pseudo-code) can return any values of any types. In the `if` and `elseif` contexts, all values are evaluated to true other than `false` and `nil`, which are evaluated to false.

Note that the number `0` and the empty string `""`, for instance, test true. This behavior is different from other programming languages that support implicit boolean conversion, like (©) and Python.

11.8. For Statements

Lua's `for` statement has two forms:

- Numerical `for`, and
- Generic `for`.

11.8.1. The numerical `for` loop

The numerical `for` loop repeats a block of code while a control variable goes through an arithmetic progression in the specified numeric range. It has the following syntax:

```
for <name> = <init>, <limit>, <step> do
    <block>
end
```

The three control expressions, `<init>`, `<limit>`, `<step>`, are evaluated *once* at the beginning of the loop. The `<step>` is optional, and its default value is `1` (integer).

The control variable, `<name>`, is a local variable to the `for` statement block. This variable is initially assigned the value of `<init>`. While the

value is not over that of `<limit>`, it executes the `<block>`, after incrementing the value by `<step>` at each iteration.

11.8.2. The generic `for` loop

The generic `for` statement works over a list of values returned from special type of functions, called iterators. On each iteration, the iterator function is called, which returns the new "next value" on each call. When the iterator returns `nil`, the loop terminates.

The generic `for` loop has the following syntax:

```
for <name list> in <exp list> do
    <block>
end
```

Note that the name list can contain one, two, or more variables. In other programming languages, the control variables in (their corresponding) `for` loops are often limited to only one or two. In Lua, an arbitrary number of variables can be used. For instance, refer to an example for the `string.gmatch` function, which can return an arbitrary number of items at each iteration.

11.9. While Statements

The `while` statement has the following syntax:

```
while <exp> do
    <block>
end
```

As with `if` and `repeat` statements, any value of the `<exp>` expression evaluates to true except for `false` and `nil`.

The `while` statement repeatedly executes the `<block>` as long as `<exp>` evaluates to true. Note that `<exp>` can be initially false, and in such a case, `<block>` will never be executed, in contrast with the `repeat` statement

11.10. Repeat Statements

The `repeat` statement has the following syntax:

```
repeat
    <block>
until <exp>
```

As with `if` and `while` statements, any value of the `<exp>` expression evaluates to true except for `false` and `nil`.

The `repeat` statement block ends after the `<exp>` condition, not at the `until` keyword, and hence the condition can refer to the local variables declared inside `<block>`.

For example,

```
local a, b = 1, 1
repeat
  a, b = b, a + b
until b > 20
print(b)
```

This script will print out `21`.

Chapter 12. The Math Library

The standard `math` library provides a number of mathematical functions within a `math` table.

12.1. Math Constants

`math.maxinteger`	The maximum value for an integer.
`math.mininteger`	The minimum value for an integer.
`math.huge`	The float value `HUGE_VAL`, which is greater than any other numeric value in terms of the comparison `>`.
`math.pi`	The float value of π.

```
print("math.maxinteger: " .. math.maxinteger)
print("math.mininteger: " .. math.mininteger)
print("math.huge: " .. math.huge)
print("math.pi: " .. math.pi)
```

On the author's computer, using the standard 64 bit Lua interpreter, `math.maxinteger`, `math.mininteger`, `math.huge`, and `math.pi` are `9223372036854775807` (`2^64 - 1`), `-9223372036854775808` (`-2^64`), `inf`, and `3.1415926535898`, respectively.

12.2. General Functions

```
math.type(x) -> string              ①
```

① The `math.type` function returns string `"integer"` or `"float"` depending on whether the argument `x` is an integer or float,

respectively. Otherwise, it returns `nil`.

```
math.tointeger(x) -> integer       ①
```

① The `math.tointeger` function returns `nil` if `x` is not convertible to an integer number. Otherwise, it returns `x` as an integer.

```
math.ult(m, n) -> boolean          ①
```

① The `math.ult` function returns `true` if and only if integer `m` is below integer `n` when they are compared *as unsigned integers*.

For example,

```
> math.ult(1, 10)
true
> math.ult(-10, -1)
true
> math.ult(1000000000, -1)
true
> math.ult(math.maxinteger, -1)    ①
true
```

① A negative number, e.g., `-1`, is bigger than any other positive number in the unsigned comparison.

12.3. Basic Math Functions

```
math.abs(x) -> number              ①
```

① The `math.abs` function returns the absolute value of `x`. If `x` is an `integer`, it returns an integer. Otherwise, it returns `float`.

```
math.sqrt(x) -> float                ①
```

① The `math.sqrt` function returns the square root of `x`. This is equivalent to the expression `x ^ 0.5`.

```
math.exp(x) -> float                 ①
```

① The `math.exp` function returns the value `e ^ x` where `e` is the base of natural logarithms.

```
math.log(x, base = e) -> float       ①
```

① The `math.log` function returns the logarithm of `x` in the given `base`.

12.4. Rounding Functions

```
math.ceil(x) -> integer              ①
```

① The `math.ceil` function returns the smallest integral value greater than or equal to `x`.

```
math.floor(x) -> integer             ①
```

① The `math.floor` function returns the largest integral value less than or equal to `x`.

```
math.fmod(x, y) -> number            ①
```

① The `math.fmod` function returns the remainder of the division of `x` by `y` that rounds the quotient towards zero. If both `x` and `y` are integers, it returns an integer. Otherwise, it returns float.

```
math.modf(x) -> integer, float      ①
```

① The `math.modf` function returns two values, the integral part and the fractional part of the given argument `x`.

12.5. Trigonometric Functions

```
math.deg(x) -> float                ①
```

① The `math.deg` function converts the angle `x` given in radians to degrees.

```
math.rad(x) -> float                ①
```

① The `math.rad` function converts the angle `x` given in degrees to radians.

```
math.cos(x) -> float                ①
```

① The `math.cos` function returns the cosine of `x` given in radians.

```
math.sin(x) -> float                ①
```

① The `math.sin` function returns the sine of `x` given in radians.

```
math.tan(x) -> float                ①
```

① The `math.tan` function returns the tangent of `x` given in radians.

```
math.acos(x) -> float               ①
```

① The `math.acos` function returns the arc cosine of `x` given in radians.

```
math.asin(x) -> float                ①
```

① The `math.asin` function returns the arc sine of `x` given in radians.

```
math.atan(y, x = 1) -> float         ①
```

① The `math.atan` function returns the arc tangent of `y/x`. The `math.atan(y)` call returns the arc tangent of `y`.

12.6. Min and Max Functions

```
math.max(x, ...) -> number           ①
```

① The `math.max` function returns the maximum value from the given arguments, according to the Lua less-than operator `<`.

```
math.min(x, ...) -> number           ①
```

① The `math.min` function returns the minimum value from the given arguments, according to the Lua less-than operator `<`.

12.7. Random Functions

12.7.1. The `math.random` function

The `math.random` function has the following 4 different forms:

```
math.random() -> float               ①
math.random(m, n) -> integer         ②
```

```
math.random(n) -> integer          ③
math.random(0) -> integer          ④
```

① The `math.random()` call returns a pseudo-random number with uniform distribution in the range `[0.0, 1.0)`, that is, from `0.0` (inclusive) to `1.0` (exclusive).

② The `math.random(m, n)` call returns a pseudo-random integer with uniform distribution in the range `[m, n]`.

③ The `math.random(n)` call is equivalent to `math.random(1, n)`.

④ The special call `math.random(0)` generates an integer with all bits pseudo-random.

12.7.2. The `math.randomseed` function

```
math.randomseed(x, y = 0) -> integer, integer  ①
math.randomseed() -> integer, integer          ②
```

① The integer parameters `x` and `y` are joined into a 128-bit seed that is used to reinitialize the pseudo-random generator.

② If `math.randomseed()` is called with no arguments, Lua generates a seed with a weak attempt for randomness.

The `math.randomseed` function returns the two 64 bit seed components that were effectively used so that the sequence can be reproducible using the same pair of seeds.

As an illustration, here's a simple module with a function that generates random dice rolls:

diceroll.lua

```
local diceroll = {}                    ①

function diceroll.setSeed()            ②
```

```
  local x, y = math.randomseed(os.time(), os.time() * 12345)
  return x, y                                    ③
end

local DICE = 6                                   ④

function diceroll.rollDice(n)                    ⑤
  if n == nil or type(n) ~= "number" or n <= 1 then
    return math.random(DICE)                     ⑥
  else
    n = math.floor(n)                            ⑦
  end

  local index, tab = 1, {}                       ⑧
  while index <= n do                            ⑨
    tab[index] = math.random(DICE)
    index = index + 1
  end

  return tab                                     ⑩
end

return diceroll
```

① The module table.

② This `setSeed` function uses the `os.time` function listed later in the OS Functions chapter.

③ A Lua function can return multiple values.

④ This is a module local variable and hence it is not "exported".

⑤ The argument `n` of the `rollDice` function is effectively optional. If it is not provided in the call, its value will be `nil`, and we handle this case in the function implementation. This is a common idiom in Lua, and in many dynamically typed programming languages.

⑥ In this case, we just return a single random integer value in the range of `[1, 6]`.

⑦ If the input argument is a floating number, we take its "floor" using the `math.floor` function, which returns an integer value.

⑧ The local variable declaration.

⑨ This `while` loop could have been written as a numerical `for` or as a `repeat` statement.

⑩ We return the table `tab` in this case. Note first that although `tab` is a *local* variable, the caller will have access to the underlying table that `tab` references. Note second that this function `rollDice` sometimes returns a single value (an integer) and returns a table (more specifically, a sequence). This is a common practice in Lua, and in many other dynamically typed programming languages.

You can call `diceroll.setSeed()` first, (optionally) store the seed values for later reproduction, if necessary, and then call the `diceroll.rollDice()` function.

For example, here's an example driver program:

main.lua

```
local x, y = diceroll.setSeed()          ①
print("seed =", x, y)

local r = diceroll.rollDice()            ②
print("dice =", r)

r = diceroll.rollDice("hey")             ③
print("dice =", r)

local rolls = diceroll.rollDice(3.14)    ④
print("rolls =", table.unpack(rolls))    ⑤

rolls = diceroll.rollDice(4)             ⑥
print("rolls =", table.unpack(rolls))
```

① We set the random number generator seed before we start using any

random number functions.

② The `rollDice()` function can be called without an argument.

③ If it is called with an invalid argument like a function, it just returns a single integer value.

④ The float number argument, e.g., `3.14`, is truncated to an integer, e.g., `3`.

⑤ The `table.unpack` function is described later in the tables chapter. It returns a list of elements from the given sequence.

⑥ It rolls the dice `4` times.

If you run this Lua script, e.g., via *lua main.lua,* it will output something like this:

```
seed =  1666892182      20577783986790
dice =  5
dice =  6
rolls = 4       6       2
rolls = 1       1       2       3
```

Chapter 13. Strings - Basics

Lua strings are sequences of bytes, and they are immutable. String indices, and table indices, in Lua are 1-based, and negative indices are interpreted as indexing backwards, from the end of the string. String literals were described earlier, including (multiline) long string literals.

Lua standard library provides a number of functions for string manipulation in the table `string`. The string library assumes one-byte character encodings (e.g., ASCII). The string functions can be used in the (object-oriented) method style syntax when the argument is a variable. For example, `string.len(s)` can be written as `s:len()`, using colon `:`.

13.1. String Concatenation

The builtin string concatenation operator is denoted by two dots (`..`), as we see throughout this book. String concatenation is an expression.

- If both operands are strings or numbers, then the default string concatenation operation is performed. The number type operands, if any, are converted to strings first.
- Otherwise,
    - If a `__concat` metamethod is defined in the first operand, this method is called.
        - If not, and if this metamethod is defined in the second operand, then it is called.
        - Otherwise, this expression throws an error.

The default string concatenation behavior:

```
> "hello" .. "world"
helloworld
> greeting, name = "hola", "juan"
```

```
> greeting .. 2 .. name
hola2juan
```

Here's a simple example of customizing string concatenation. Metatables and metamethods are explained more thoroughly in the later part of the book.

```
> mt = {                                ①
>>   __concat = function(a, b) return a.val .. ":" .. b.val
end
>> }
> obj1 = setmetatable({}, mt)           ②
> obj1.val = "[object 1]"
> obj2 = setmetatable({}, mt)
> obj2.val = "[object 2]"
> obj1 .. obj2                          ③
[object 1]:[object 2]
```

① This table `mt` has a function/method with a key `__concat`.

② We create two tables, `obj1` and `obj2`, using `mt` as their metatables.

13.2. The `string.len` Function

We will go through some of the more commonly used string functions in this chapter.

```
string.len(s) -> integer
```

The `s:len()` function returns the length of the given string `s`. This is the same as the length expression `#s`.

```
> string.len("abracadabra")
11
```

```
> s = "hocus pocus"
> s:len()
11
```

13.3. The `string.lower` Function

```
string.lower(s) -> string
```

The `s:lower()` function returns a (new) string as a copy of `s`, but with all uppercase letters changed to lowercase. Which are uppercase letters vs lowercase letters is dependent on the current locale.

```
> string.lower("UNder Ground !!!")
under ground !!!
```

13.4. The `string.upper` Function

```
string.upper(s) -> string
```

The `s:upper()` function returns a copy of `s` with each of the lowercase letters changed to uppercase.

```
> string.upper("UPPER Floor ??")
UPPER FLOOR ??
```

13.5. The `string.rep` Function

```
string.rep(s, n, sep = "") -> string
```

The `s:rep(n, sep)` function concatenates `s` for `n` times, separated by `sep`, and returns its result. When `n` is a non-positive number, the function returns an empty string `""`.

```
> string.rep("hocus", 4, " ")
hocus hocus hocus hocus
```

13.6. The `string.reverse` Function

```
string.reverse(s) -> string
```

The `s:reverse()` function returns a string that is the string `s` reversed.

```
> string.reverse("Rainbow? Or Snowbow?")
?wobwonS rO ?wobniaR
```

13.7. The `string.format` Function

```
string.format(s, ···) -> string
```

The `s:format(···)` function returns a "formatted" string version of the format string `s`, which can include zero or more conversion specifiers. The format string follows the same rules as the ISO C function `sprintf` with some minor differences.

```
> month, day, holiday = "December", 25, "Christmas"
> string.format("%s %d is a %s day!", month, day, holiday)
December 25 is a Christmas day!
```

Chapter 14. String Manipulation

14.1. The `string.sub` Function

```
string.sub(s, i, j = -1) -> string
```

The `s:sub(i, j)` function returns the substring of `s` from the start index `i` to the end index `j` (both inclusive). Negative indices are allowed. For instance,

```
> string.sub("Expecto Patronum", 1)      ①
Expecto Patronum
> string.sub("Expecto Patronum", 1, 7)   ②
Expecto
> string.sub("Expecto Patronum", -8)     ③
Patronum
> string.sub("Expecto Patronum", 6, 13)  ④
to Patro
> string.sub("Expecto Patronum", 1, -20) ⑤

>                                        ⑥
```

① The `s:sub(1)` call returns the entire string.

② The `s:sub(1, n)` call returns a "prefix", e.g., the first `n` characters.

③ The `s:sub(-n)` call returns a "suffix", e.g., the last `n` characters.

④ A more general application of `s:sub`.

⑤ This particular call returns an empty string since the end index goes to the left of the start index.

⑥ The following prompt is included to show the previous output, e.g., the empty string.

14.2. The `string.find` Function

```
string.find(s, pattern, init = 1, plain = false) -> integer,
integer, ...
```

The `s:find(pattern, init, plain)` function searches for the first match of `pattern` in the string `s`, starting from index `init`, and if one is found, then it returns its start and end indices. If no `pattern` is found in `s`, it returns `nil`.

By default, `string.find` uses the "pattern match". When `plain` is set to `true`, this function does a plain substring search.

If the pattern has captures, then in a successful match the captured values are also returned, after the two indices.

```
> string.find("hello there, hello all!", "hell")
1       4
> string.find("hello there, hello all!", "hello", 3)
14      18
> string.find("hello there, hello all!", "(%w+)%W+(%w+)")
1       11      hello   there
```

14.3. The `string.match` Function

```
string.match(s, pattern, init = 1) -> ...
```

The `s:match(pattern, init)` function works in a similar manner to `s:find`. It searches for the first `pattern` in the string `s`, starting from the index `init`. If a match is found, then it returns the captures from the `pattern`. If `pattern` includes no captures, then the entire matched string is returned. If it fails to find `pattern`, it returns `nil`.

```
> string.match("My SSN is 123-45-6789.", "%d+-%d+-%d+")
123-45-6789                             ①
> print(string.match([[                 ②
>> Today is the 21st day of the 3rd month in 2023
>> ]],
>> "(%d+)%D*(%d+)%D*(%d+)"))
21      3       2023                    ③
```

① The pattern includes no capture. Hence, it returns the whole match.

② We need this `print()` call here to display the output of the multiline expression in REPL.

③ The output has three captures.

14.4. The `string.gmatch` Function

```
string.gmatch(s, pattern, init = 1) -> iterator
```

The `s:gmatch(pattern, init)` function finds *all* matches of `pattern` in the string `s`, starting from `init`. It returns an iterator function that yields, in each iteration, the next list of captures from `pattern`, or the whole match if there is no capture specified. The `pattern` should not generally include a caret '^' at its start since it will prevent the iteration more than once. For example,

```
> s = "sun=sol:sol, moon=lua:luna, star=estrela:estrella"
> for w1, w2, w3 in s:gmatch("(%w+)=(%w+):(%w+)") do
>> print(w1, w2, w3)
>> end
sun     sol     sol
moon    lua     luna
star    estrela estrella
```

14.5. The `string.gsub` Function

```
string.gsub(s, pattern, repl, n = nil) -> string, integer
```

The `s:gsub(pattern, repl, n)` function performs a "global search and replace" operation with `repl` on the string `s`, to the maximum of `n` (`>= 0`), if provided. It returns two values:

1. A copy of `s` in which all (or the first `n`) occurrences of `pattern` have been replaced by `repl`, and
2. The total number of matches that occurred.

The `repl` can be a string, a table, or a function:

- If `repl` is a string, then its value is used for replacement.
- If `repl` is a table, then the table is queried for every match, using the first capture as the key.
- If `repl` is a function, then this function is called every time a match occurs, with all captured substrings passed as arguments, in order.

For example,

```
> string.gsub("hello hell hell", "hel", "Jel")
Jello Jell Jell    3
> string.gsub("hello hell hell", "hel", "Jel", 1)
Jello hell hell    1
```

Chapter 15. Regular Expressions

Lua's pattern-matching functions take the regular expression patterns as their first argument (after `self`), as described in the previous chapter, e.g., `string.match`, `string.gmatch`, `string.gsub`, and `string.find` (when `plain` is `false`).

Here's a brief explanation of the regular expression syntax as used in these string functions.

15.1. The Patterns

In Lua, a regular expression pattern is just a regular string. A pattern comprises zero, one, or more pattern items, as explained below.

- A caret `^` at the beginning of a pattern anchors the match at the beginning of the subject string.
- A dollar `$` at the end of a pattern anchors the match at the end of the subject string.

For example, `"^$"` matches an empty string (or, an empty line).

15.1.1. Character class

A pattern item is made of character classes. A character class represents a set of characters.

- Any single character which is not one of the "magic characters" `^$()%.[]*+-?` represents the character itself.
- A dot `.` represents any character.
- A character class `%C`, where `C` is a non-alphanumeric character, represents the character `C` itself. This class can be used to escape the magic characters, for instance.

- `[set]` represents the class which is the union of all characters in the specified set. A range of characters can be specified by putting a hyphen `-` between the two end characters of the range. (The `-` character can be represented as `%-`.)
- `[^set]` represents the complement of the specified set. That is, all characters that are not in the set.

The following single character classes depend on the current locale. In particular, the definitions of letter, space, and other character groups depend on the locale.

- `%a`: Any letter.
- `%c`: Any control character.
- `%d`: Any digit.
- `%g`: Any printable character except space.
- `%l`: Any lowercase letter. In an English locale, this is equivalent to `[a-z]`.
- `%p`: Any punctuation character.
- `%s`: Any space character.
- `%u`: Any uppercase letter. In an English locale, this is equivalent to `[A-Z]`.
- `%w`: Any alphanumeric character. In an English locale, this is equivalent to `[0-9A-Za-z]`.
- `%x`: Any hexadecimal digit.

For these classes, the corresponding uppercase letter represents the complement of the class. For example, `%W` represents any non-alphanumeric character since `%w` represents an alphanumeric character.

15.1.2. Pattern item

A pattern is a sequence of the following pattern items:

A single character class

Matches any single character in the class.

A single character class followed by `+`

Matches the longest possible sequences of one or more characters in the class.

A single character class followed by `*`

Matches the longest possible sequences of zero or more characters in the class.

A single character class followed by `-`

Matches sequences of zero or more characters in the class. Unlike `*`, it will match the shortest possible sequence.

A single character class followed by `?`

Matches at most one occurrence of a character in the class.

`%n`, for `n` between `1` and `9`

Matches a substring equal to the `n`-th captured string. Captures are explained below.

`%bxy`, where `x` and `y` are two distinct characters

Matches strings that start with `x`, end with `y`, and `x` and `y` are *balanced*. This means that, if one reads the string from left to right, counting +1 for an `x` and -1 for a `y`, the ending `y` is the first `y` where the count reaches 0.

The pattern item `%b{}`, for example, matches `"{hello}"`, but not `"{{world}"` in its entirety (although it still matches `{world}` or `{{{Hello World}}}"`, etc.).

`%f[set]`, a frontier pattern

Matches a sequence of spaces whose subsequent character belongs to the set and whose preceding character does not belong to the set.

15.2. The Captures

A pattern can contain sub-patterns enclosed in parentheses. They are called captures.

When a match succeeds, the parts of the subject string that match any captures are stored. Captures are numbered from left to right according to their left parentheses.

For instance, in the pattern `"(.(%d+)%s*(%w+))"`, the part of a subject string that matches `.(%d+)%s*(%w+)` is stored as the first capture with the number `1`. The substring that matches `%d+` is stored as the capture number `2`. Finally, the captured part that matches `%w+` has the capture number `3`.

As a special case, the empty parentheses `()` captures the current string position (an index). For example, when the pattern `"()ll()"` matches a string `"hello"`, it will end up with two captures, `3` and `5`.

Chapter 16. Tables

Lua's `table` is comparable to JavaScript's `Object`.

Tables can be used to create values and data structures like lists and maps, to provide namespaces (e.g., modules), and even to create custom types. In fact, in Lua, `table` is the primary, and essentially the only, mechanism to support much of the language extension.

As a data structure, `table` has a dual nature. Some tables can be viewed as arrays, and some can be viewed as maps. As we have seen earlier in the context of the length operator, there is a special category of tables, namely, sequences.

Although it is an implementation detail, the standard Lua interpreter optimizes the memory allocation of (small) linear tables (e.g., with only integer indices), including sequences, by utilizing consecutive memory space. Hence, in such cases, Lua tables are more like the native array types in many other programming languages, for example, with similar performance characteristics.

16.1. Table Constructors

Table constructors are expressions that create tables. They are syntactically literals, but they do not yield constant expressions. Every time a constructor is evaluated, a *new* `table` is returned.

A constructor literal can be used to create an empty table or to create a table with some of its fields initialized.

```
local t = {}              ①
```

① The `table` constructor expression on the right hand side creates a new empty table.

The list of initial values can be provided in the constructor, as a series of fields, separated by commas `,` or semicolons `;`.

A table field is a pair of a key/name and its value. There are three different ways to specify a field:

- `[<exp1>] = <exp2>`,
- `<name> = <value>`, and
- `<value>`.

Each field of the form `[<exp1>] = <exp2>` in a table constructor adds an entry with key `<exp1>` and value `<exp2>`. A field of the second form `<name> = <value>` is equivalent to `["<name>"] = <value>`. Fields of the third form `<value>` are equivalent to `[i] = <value>`, where `i` are sequential integer indices starting from `1`, skipping other forms.

For example,

```
local t1 = { a = 1, b = 2 }                          ①

local t2 = {                                         ②
     ["apples"] = 10,
     ["oranges" .. "bananas"] = 20,
}

local t3 = { "hello"; k = "beautiful"; "world" }     ③

local f = function(a, b) return a + b end
local t4 = { [f(1, 2)] = f }                         ④
```

① The `table` constructor creates a new table with two fields, `["a"] = 1` and `["b"] = 2`.

② The constructor creates a new table with two fields, `["apples"] = 10` and `["orangesbananas"] = 20`. The trailing comma or semicolon is optional.

③ The constructor creates a new table with three fields, `[1] = "hello"`, `["k"] = "beautiful"`, and `[2] = "world"`.

④ The constructor creates a new table with one field, `[3] = f`, where `f` is a function, as defined in the example.

The `table` Table

The `table` library provides generic functions for table manipulation. It provides all its functions inside the namespace of the `table` table.

Computing the length of a table is described earlier in the book, in the length operator chapter. When dealing with the table functions requiring the length of a table, therefore, one need to be aware of how the lengths are computed. In particular, it should be noted that all non-numeric keys, and negative integer keys, are ignored when computing the length of a table.

16.2. The `table.pack` Function

```
table.pack(···) -> table
```

The `table.pack` function returns a new table with all arguments stored into positive integer keys `1`, `2`, `3`, etc.

For example,

```
> function packDemo()
>> local list = table.pack(1, 3, 5)
>> for i, v in ipairs(list) do
>> print(i, v)
>> end
>> end
> packDemo()
```

```
1       1
2       3
3       5
```

16.3. The table.unpack Function

```
table.unpack(list, i = 1, j = #list) -> ...
```

The table.unpack function returns zero, one, or more elements from the given list. This function is equivalent to

```
function(list, i, j)
    return list[i], list[i+1], ···, list[j]
end
```

For example,

```
> function unpackDemo()
>> local list = { "red", "green", "blue" }
>> local unpacked = table.unpack(list)
>> print(table.unpack(list))
>> end
> unpackDemo()
red     green   blue
```

16.4. The table.concat Function

```
table.concat(list, sep = "", i = 1, j = #list) -> string
```

The table.concat function concatenates elements of list, between i

and `j`, and it returns the concatenated string. The elements of `list` should be either strings or numbers. Effectively, `table.concat` returns the string `list[i]..sep..list[i+1] ··· sep..list[j]`.

Here's an example:

```
> function concatDemo()
>> local list = { 1, 2, "apple" }
>> local s = table.concat(list, ",")
>> print("s =" .. s)
>> end
> concatDemo()
s = 1,2,apple
```

16.5. The `table.insert` Function

```
table.insert(list, pos, value)
table.insert(list, value)
```

The `table.insert` function inserts element `value` at position `pos` in `list`. When `pos` is not specified, `table.insert(t, v)` inserts `v` at the end of the list `t`.

For instance,

```
> function insertDemo()
>> local list = { 1, 3 }
>> print(table.unpack(list))
>> table.insert(list, 5)
>> print(table.unpack(list))
>> table.insert(list, 2, 20)
>> print(table.unpack(list))
>> end
> insertDemo()
```

```
1       3
1       3       5
1       20      3       5
```

16.6. The table.remove Function

```
table.remove(list, pos = #list) -> value
```

The table.remove function removes from list the element at position pos and it returns the removed element. Calling table.remove(l) without pos removes the last element in l.

Here's a simple example:

```
> function removeDemo()
>> local list = { 1, 3, 5, 7, 9 }
>> print(table.unpack(list))
>> table.remove(list)
>> print(table.unpack(list))
>> table.remove(list, 2)
>> print(table.unpack(list))
>> end
> removeDemo()
1       3       5       7       9
1       3       5       7
1       5       7
>
```

16.7. The table.move Function

```
table.move(a1, f, e, t, a2 = a1) -> table
```

The `table.move` function moves elements from the table `a1` to the table `a2`, and it returns the destination table `a2`. When the `a2` argument is omitted, the same source table `a1` is also used as the destination.

In effect, `table.move(a1, f, e, t, a2)` is equivalent to the following multiple assignment:

```
a2[t], ··· = a1[f], ···, a1[e]
```

For example,

```
> function moveDemo()
>> local list = { 1, 3, 5, 7 }
>> print(table.unpack(list))
>> local moved = { 'a', 'b', 'c', 'd', 'e', 'f' }
>> moved = table.move(list, 1, 3, 2, moved)
>> print(table.unpack(moved))
>> moved = table.move(moved, 1, 3, 5)
>> print(table.unpack(moved))
>> end
> moveDemo()
1       3       5       7
a       1       3       5       e       f
a       1       3       5       a       1       3
```

16.8. The `table.sort` Function

```
table.sort(list, comp = nil)
```

The `table.sort` function sorts the `list` elements in a given order, in-place. The sort algorithm is not *stable*, that is, the relative positions among the elements that are considered equal by the specified order may change by the sort.

By default, the standard Lua operator `<` is used for sorting. If `comp` is given, then it must be a function that receives two list elements and returns true when the first element must come before the second in the final order, so that, after the sort, `i ⇐ j` implies `not comp(list[j], list[i])`.

Here's a simple example using `table.sort`. The demo function is included in a module `sortdemo`:

sortdemo.lua

```
local sortdemo = {}

function sortdemo.sort()
  local list = { 3, 2, 7, 1 }
  print("input list:", table.unpack(list))
  table.sort(list, function(i, j)
    -- return i < j                                    ①
    return j < i                                       ②
  end)
  print("sorted list:", table.unpack(list))
end

return sortdemo
```

① This will sort in the normal ascending order.

② This will sort in the reverse or descending order.

If we try calling this `sortdemo.sort()` function in the Lua REPL,

```
> s = require("sortdemo")
> s.sort()
input list:     3       2       7       1
sorted list:    7       3       2       1
```

Chapter 17. Metatables

Every value in Lua may be associated with a companion table called the *metatable*. The value's metatable defines its behavior, e.g., for certain (well-defined) events. You can change various aspects of the value's behavior by setting specific fields in its metatable.

The key for each event in a metatable is a string with the event name prefixed by two underscores `__`. The corresponding value is called a metavalue. For most events, the metavalue must be a function, which is then called a metamethod. We will discuss some predefined metamethods in the Lua language in the next chapter.

It should be noted that multiple values can share the same metatable, and hence it is a "many to one" relationship. As we will see in a later chapter, shared metatables can be used as "protoptypes" for creating other tables, e.g., with the similar behavior.

The values of all builtin types, other than `tables` and `userdata`, share one single metatable per type. That is, there is one single metatable for all numbers, one for all strings, etc.

17.1. The `__index` Metavalue

```
__index(obj, key) -> value
```

The indexing access operation, e.g., `obj[key]`, can be delegated to the object's metatable's `__index` field if one is found.

When `obj` is not a table or it does not have a field with the given `key`, it looks up its `key` in its metatable's `__index`. This metavalue can be a function, or a table. Or any other table with its own `__index` field. If the `key` is not found even in this metavalue, then it looks it up again in the

`__index` field of this metavalue if it has one. And so forth, until the field with the given `key` is found, or until there are no more `__index` fields.

If a field is found with a given `key`, and if this field is a table, it returns the value of the table field for the given `key`. If the found field is a function, then it calls the function with the argument `key`. Otherwise, it returns `nil`.

17.2. The `__newindex` Metavalue

```
__newindex(tbl, key, value)
```

Similar to `__index`, the indexing assignment operation, e.g., `obj[key] = value`, can be delegated to the object's metatable's `__newindex` field if one is found.

When `obj` is not a table or the given `key` is not present in `obj`, it looks up its key in its metatable's `__newindex` metavalue. Like the `__index` field, if this key is still not found, then it is looked up in this metavalue's `__newindex` field, and so on.

17.3. The `__metatable` Metavalue

The metatable relationship can be "chained", or more aptly, "redirected". If a given table's (direct) metatable happens to have a field `__metatable`, then the value of this field is used as the given table's (real) metatable. This indirect metatable may have a field `__metatable` as well. If so, then again this becomes the original table's (real) metatable. And so forth, until it reaches the (ultimate) metatable which does not have the `__metatable` field set.

(Note that, by default, this metavalue is not set, and hence this kind of " metatable chaining" is rather uncommon in practice.)

17.4. The `getmetatable` Function

You can query the metatable of any value using the builtin `getmetatable` function.

```
getmetatable(obj) -> table          ①
```

① For a given value `obj`, it returns the value's metatable.

Some objects, e.g., of the `table` or `userdata` types, may not be associated with metatables. In such a case, the `getmetatable` function returns `nil`.

As indicated, this function returns the "ultimate" metatable, through the `__metatable` chaining.

17.5. The `setmetatable` Function

You cannot change the metatable of values of any builtin types other than `tables`. For this, we can use the builtin `setmetatable` function to set or replace exiting metatables of a table.

```
setmetatable(tbl, metatable) --> table   ①
setmetatable(tbl) -> table               ②
```

① It sets the second argument `metatable` as the metatable of `tbl`.

② It removes the metatable of the given table `tbl`.

If successful, the `setmetatable` function returns the original `tbl` after updating/removing its metatable.

If the existing metatable of a given table has a `__metatable` metavalue, then this metatable cannot be updated or removed, using this

`setmetatable` function, e.g., until this metavalue is removed. Attempting to do so will raise an error.

```
> m, t = {}, {}
> m, t
table: 0x55d4edf75270   table: 0x55d4edf7a8f0
> setmetatable(t, m)
table: 0x55d4edf7a8f0
> getmetatable(t)
table: 0x55d4edf75270
```

```
> t, m, mm = {}, {}, {}
> m.__metatable = mm
> t, m, mm
table: 0x55d4edf76250   table: 0x55d4edf75c50   table:
0x55d4edf75be0
> setmetatable(t, m)
table: 0x55d4edf76250
> getmetatable(t)
table: 0x55d4edf75be0
```

Chapter 18. Metamethods

A number of metamethods are predefined in Lua, and they have special meanings. These methods are used to customize the behavior of a table (e.g., when used as an "object"). They are comparable to Python's "dunder methods".

By convention, all keys of the predefined metamethods start with two underscores `__` followed by lowercase alphabets, somewhat similar to that of Python.

Metatables can have other fields besides these special metamethods listed in this, and other, chapters. Some functions in the standard library use other fields in metatables for their own purposes, such as `__tostring` or `__gc`, etc.

18.1. The `__call` Method

```
__call(f, ...) -> ...
```

A non-function value can be made *callable* by implementing this `__call` metamethod.

When a function call expression is used on `f`, and `f` is not a function, then `__call` is looked up in the metatable of `f`. If this metamethod is found, it is called with `f` as its first argument, followed by the arguments of the original call, if any.

The results of this metamethod call are returned as the results of the original function call on `f`. The `__call` is the only one that allows multiple results.

18.2. The `__len` Method

```
__len(tbl) -> integer
```

The `__len` metamethod is looked up in the length operation `#` when the operand `tbl` is not a string. If one exists, Lua calls this metamethod with `tbl`, and it returns the result as the value of the original length operation.

If this metamethod is not found, but if `tbl` is of the `table` type, then it falls back to calling the table length operation. Otherwise, it raises an error.

18.3. The `__concat` Method

```
__concat(tbl, obj)
```

The `__concat` metamethod is described in the string basics chapter. Lua will try this metamethod when the concatenation `..` operation is performed on non-string or non-number type values.

18.4. Arithmetic Operations

If any operand for an arithmetic operation is not a number, Lua will try to call a metamethod of its type. It starts by checking the first operand. If that operand does not define a metamethod corresponding to the operation, then Lua will check the second operand. If Lua can find the corresponding metamethod, it calls the metamethod with the two operands as arguments, and the result of the call is the result of the operation. Otherwise, if no metamethod is found, Lua raises an error.

18.4.1. The `__add` method

The addition `+` operation:

```
__add(tbl, rhs) -> value
```

18.4.2. The `__sub` method

The subtraction `-` operation:

```
__sub(tbl, rhs) -> value
```

18.4.3. The `__mul` method

The multiplication `*` operation:

```
__mul(tbl, rhs) -> value
```

18.4.4. The `__div` method

The division `/` operation:

```
__div(tbl, rhs) -> value
```

18.4.5. The `__mod` method

The modulo `%` operation:

```
__mod(tbl, rhs) -> value
```

18.4.6. The `__pow` method

The exponentiation `^` operation:

```
__pow(tbl, exponent) -> value
```

18.4.7. The `__unm` method

The unary negation `-` operation:

```
__unm(tbl) -> value
```

18.4.8. The `__idiv` method

The floor division `//` operation:

```
__idiv(tbl, rhs) -> value
```

18.5. Comparison Operations

The behavior is similar to that of arithmetic metamethods. But, the comparison metamethods are checked only when both operands are `tables` or both operands are full `userdata` (which we do not discuss in this book).

When the operands are the same objects, the operations result in trivial result. (They are identical.) Otherwise, the corresponding metamethod is checked, again from the left operand first and then the right one.

If one is found, this (first found) metamethod is called, and its result is returned as a `boolean` as its original comparison expression. Otherwise, it raises an error.

18.5.1. The `__eq` method

The equal `==` operation:

```
__eq(tbl, rhs) -> boolean
```

18.5.2. The `__lt` method

The less than `<` operation:

```
__lt(tbl, rhs) -> boolean
```

18.5.3. The `__le` method

The less equal `<=` operation:

```
__le(tbl, rhs) -> boolean
```

18.6. Bitwise Operations

The behavior is similar to that of arithmetic metamethods except that the relevant metamethod is checked only if either of the operand is not an integer (or, not a float coercible to an integer).

18.6.1. The `__band` method

The bitwise AND `&` operation:

```
__band(tbl, rhs) -> integer
```

18.6.2. The `__bor` method

The bitwise OR `|` operation:

```
__bor(tbl, rhs) -> integer
```

18.6.3. The `__bxor` method

The bitwise exclusive OR `~` operation:

```
__bxor(tbl, rhs) -> integer
```

18.6.4. The `__bnot` method

The bitwise unary NOT `~` operation:

```
__bnot(tbl, rhs) -> integer
```

18.6.5. The `__shl` method

The bitwise left shift `<<` operation:

```
__shl(tbl, shift) -> integer
```

18.6.6. The `__shr` method

The bitwise right shift `>>` operation:

```
__shr(tbl, shift) -> integer
```

Chapter 19. Iterators

A `table` can be iterated over, e.g., in the `for` loop, using the builtin functions like `pairs` and `ipairs`, etc. These are often known as iterators or iterator functions. The iteration behavior of a value can be customized by implementing a `__pairs` metamethod.

19.1. The `pairs` Function

```
pairs(tbl) -> val1, val2, val3
pairs(tbl) -> function, table, nil
```

If the given value `tbl` has a metamethod `__pairs` defined, Lua calls it with `tbl` as an argument. Then, it returns the first three results from this metamethod call.

If `tbl` does not have a `__pairs` metamethod, then Lua returns three values: the `next` function, the table `tbl`, and `nil`. This set of return values allows an iteration over all key-value pairs of table `tbl` using the generic `for` loop construction:

```
for k, v in pairs(tbl) do
    <body>
end
```

19.2. The `ipairs` Function

```
ipairs(tbl) -> function, table, 0
```

It works in a similar way as `pairs`. The `ipairs` function returns three

values: an iterator function, the table `tbl`, and `0`. This allows an iteration over all key-value pairs of table `tbl` in the following way:

```
for i, v in ipairs(tbl) do
    <body>
end
```

This will iterate over the index-value pairs `(1, tbl[1])`, `(2, tbl[2])`, ..., up to the first absent index.

19.3. The `__pairs` Metamethod

A `table`'s `__pairs` metamethod can be implemented to customize the table's iteration behavior. Here's a simple example:

```
local mt = {}                          ①
function mt.__pairs(t)                 ②
  local function _p(t, k)              ③
    if k == nil then
      return 1, "a"
    elseif k == 1 then
      return 2, "b"
    else
      return nil
    end
  end

  return _p, t, nil                    ④
end

local function new()                   ⑤
  local o = {}                         ⑥
  setmetatable(o, mt)                  ⑦
  return o
end
```

```
local obj = new()                    ⑧
for k, v in pairs(obj) do            ⑨
  print(k, v)
end
```

① We will use the table `mt` as an metatable for the tables we will create below.

② The `__pairs` metamethod.

③ The `_p(t, k)` local function is to be used as a next function. It takes two arguments, a table `t` and an index `k`, and it returns two values, the next index and its value. The return value of `nil` indicates the end of iterable items.

④ The `__pairs` metamethod returns three values, the next function, the table, and `nil`.

⑤ The factory method is described in the next chapter, OOP in Lua. This function returns a new table every time it is called.

⑥ We create a new table, referenced by a local variable `o`.

⑦ And, we then set `mt` as its metatable, using the setmetatable function. This table is then return to the caller of the `new()` function.

⑧ We create a new table, `obj`, with the earlier defined `__pairs` metamethod.

⑨ Now when this table `obj` is used, for instance, in the `for` iteration with the `pairs` function, Lua will use its `__pairs` metamethod implementation.

For example, if we run this script on the terminal, we will see the following output:

```
1       a
2       b
```

19.4. The next Function

```
next(tbl, idx = nil) -> integer, value
```

One can iterate over all fields of a table using the builtin `next` function.

Its first argument is a table `tbl` and its second argument is an index `idx` in this table. A call to `next` returns the next index of the table and its associated value, for instance, something like `idx + 1, tbl[idx + 1]`.

When the `next()` function is called only with the first argument, it returns the first index and its associated value. The return value of `nil` in this case means that the given table `tbl` is empty. When the argument `idx` happens to be the last valid index in the table, the `next()` call also returns `nil`.

Here's an example code that uses the `next` function:

```
local t0 = {}                          ①
local t1 = { 1, 2, 3 }
local t2 = { a = 2, b = 4, c = 6 }

local function isEmpty(t)              ②
  return next(t) == nil
end

for _, t in ipairs { t0, t1, t2 } do ③
  print(isEmpty(t))
end

local function iterate(t)              ④
  local i, v = next(t)                 ⑤
  while i ~= nil do
    print(v)
```

```
    i, v = next(t, i)                ⑥
  end
end

for _, t in ipairs { t0, t1, t2 } do ⑦
  iterate(t)
  print("------------")
end
```

① We define three tables for illustration.

② The implementation of the `isEmpty` function relies on the fact that `next(t)` returns `nil` if and only if `t` is an empty table.

③ A test. The output will be *true false false* (in three separate lines).

④ The `iterate` function iterates over a given table `t`. In this example, we merely print out the value of each item.

⑤ The first `next(t)` call.

⑥ Given the current index for table `t`, `next(t, i)` returns the next index and its value. If there is no more item, it returns `nil`, and hence the `while` loop will terminate when `next` runs out of items.

⑦ A test. Output will be something like the following:

```
------------
1
2
3
------------
2
6
4
------------
```

19.5. The `select` Function

```
select (idx, ···) -> ...
```

The `select` function can be used for two different purposes.

- If `idx` is a string `"#"`, then it returns the total number of extra arguments after `"#"`.
- If `idx` is an integer, then `select()` returns all arguments after `idx` from the trailing list of arguments. A negative number indexes from the end (`-1` is the last argument)

For example,

```
local y = select("#", "a", "b", "c")
print(y, type(y))

for i, v in ipairs {select(2, "a", "b", "c", "d")} do
  print(i, v)
end
```

This script will output something like this:

```
3       number
2       b
3       c
4       d
```

Chapter 20. Object Oriented Programming in Lua

A `table` is an "object" as the term is commonly used in programming languages like JavaScript and Python. Tables in Lua have invariant *identities*, separate from their values. Structurally, a table contains other variables (table fields) which reference other values including other tables.

In class-based OOP programming languages like Java and C#, you create a class and then you create (one or more structurally identical) objects of the given class. In contrast, in object-based languages like Lua, you just create objects, which are all one of a kind, so to speak. Languages like JavaScript and Python, and Lua, use the prototype pattern for "mass production" of the structurally similar objects, metaphorically speaking. Unlike classes-objects, prototypes are just objects, or just tables in case of Lua. An object can be a prototype of another object. In Lua, a metatable (just another table) and its `__index` field act as a prototype of the given table. In fact, it is a common pattern that one table is used for both metatable and its `__index`.

Tables sharing the same metatable are all behaviorally and structurally similar, and they can even share the same variables (or, "state"). The structural similarity largely comes from their shared metatable's `__index` field. On the other hand, the behavioral similarity comes from their shared metamethods as well as other methods defined in the metatable's `__index` field (and, the `__newindex` field).

Using the prototypes, object-based languages can "emulate" the more traditional OOP syntax of the class-based languages. Modern JavaScript (ES2015 and later) and Python have builtin support for "classes" (which are essentially based on prototypes). Lua does not. But, there are some commonly used patterns in Lua that are rather similar to the OOP syntax of JavaScript and Python. We will discuss some of them in this

chapter. (Note that it is not a primer to the OOP, and rather the OOP *specifically applied to Lua*. You will need some prior knowledge of the OOP style. Otherwise you can skip this chapter.)

20.1. Factory Methods

In class-based OOP programming languages, objects are created using special functions of a given class, called the constructors. In non-object oriented programming languages like, for example, Go and Rust, the common pattern is using the "factory methods".

Here's an example, as applied to Lua.

```
local mt = {}                          ①
mt.__index = mt                        ②

mt.sharedSecret = "Huh?"               ③

local function new(name)               ④
  local o = {}                         ⑤
  o.name = name                        ⑥
  o.greeting = function()
    print("hello " .. o.name)
  end
  setmetatable(o, mt)                  ⑦
  return o
end

local obj1 = new("joe")                ⑧
local obj2 = new("jill")

print(obj1.sharedSecret)               ⑨
print(obj2.sharedSecret)

obj1.greeting()                        ⑩
obj2.greeting()
```

① This is the table that we are going to use as a common metatable for our *objects*.

② It is conventional to use the same table for both the metatable and its `__index` table unless there is a specific reason why two separate tables are required.

③ The `sharedSecret` belongs to the metatable, and hence it is shared across all objects with this same metatable.

④ This is a factory method that creates and returns an *object*.

⑤ A new object is created every time we call this `new` function.

⑥ Any field that belongs to this object is an "instance variable".

⑦ Every object that is created by `new` share the same metatable `mt`.

⑧ These two calls will create two (similar) objects with the common metatable.

⑨ Each of these two lines will print out *Huh?* since the `sharedSecret` field belongs to `mt`.

⑩ The first and second lines will print out *hello joe* and *hello jill*, respectively. Each object has a different value for the field `name`.

Although it is not explicitly included in this sample code, if we want to share this `new` function, we can make the function either global (not recommended) or we can put the whole code in a table-based module.

20.2. Classes and Constructors

As just illustrated in this example, the factory method consists of two components, if you will: The shared part, e.g., the metatable `mt`, and the individual part, the object `o`.

The object `o`, in this example, contains the metatable, and hence `o` is the overall "template" for the objects we are creating.

We can *call* this object a "class". Using *classes* may be a bit easier on eyes to some people, especially those who are used to the OOP-style *syntax*. (Note that a *class* object does not really create a *new type* in Lua.) Here's an updated code using the above example:

monkey.lua

```
local mod = {}                                          ①

mod.Monkey = {}                                         ②
mod.Monkey.__index = mod.Monkey                         ③
mod.Monkey.__newindex = mod.Monkey                      ④

mod.Monkey.tribe = "Dance Monkey"                       ⑤
mod.Monkey.dance = function(cls)                        ⑥
  print("Dance, dance, dance. We are " .. cls.tribe)
end

local metaMonkey = {}                                   ⑦
metaMonkey.__call = function(cls, name)                 ⑧
  local o = {}                                          ⑨
  o.name = name or ""                                   ⑩
  o.__tostring = function(self)                         ⑪
    return "We are " .. self.tribe
  end

  o.greeting = function(self)                           ⑫
    print("hello, I'm monkey " .. self.name)
  end

  o = setmetatable(o, mod.Monkey)                       ⑬
  return o                                              ⑭
end

mod.Monkey = setmetatable(mod.Monkey, metaMonkey)       ⑮

return mod
```

① We define our *class* in a *module*.

② The module variable `Monkey` is the "*class*" that we are going to create.

③ This `__index` metamethod handles the getter parts of the class variables and methods.

④ Likewise, this `__newindex` metamethod handles their setter parts.

⑤ `Monkey.tribe` corresponds to a "class variable".

⑥ `Monkey.dance` corresponds to a "class method". Note the first parameter to the function, `cls`, which is used to access the *class variables*.

⑦ The `metaMonkey` table is used to define a "constructor". It will be used as the metatable of `Monkey`.

⑧ This `__call` metamethod makes the associated table, `Monkey` in this example, *callable*. We use this function as a constructor for the `Monkey` instances.

⑨ We return a new table (object) every time the constructor is called. The object `o` is used to store "instance variables" and "instance methods".

⑩ The table field, `o.name`, corresponds to an instance variable.

⑪ The `o.__tostring`, metamethod corresponds to an instance method. Note the first parameter to the function, `self`, which is used to access, in this particular example, the *class variable*, `tribe`. The parameter names, `cls` and `self`, are arbitrary in this example. What matter is that they are the first parameter, and they represent an instance of `Monkey`.

⑫ The table field, `o.greeting`, corresponds to another instance method. The first parameter to the function, `self`, is used to access the `self.name` instance variable in this example.

⑬ This is what makes `Monkey` and `o` behave like a class and an instance, respectively, through the metatable relationship.

⑭ As stated, we return a *new* table/instance every time.

⑮ Now, `Monkey` is tied to the `__call` metamethod.

Here's a simple unit test program using the `assert` function.

monkey_test.lua

```
local monkey = require("monkey")                    ①

local monk1 = monkey.Monkey("joe")                  ②
local monk2 = monkey.Monkey("jill")

assert(monk1.tribe == "Dance Monkey",               ③
  "The monkey's tribe is Dance Monkey.")
assert(monk2.tribe == "Dance Monkey",
  "The monkey's tribe is Dance Monkey.")

assert(monk1.name == "joe",                         ④
  "The monkey 1's name is joe.")
assert(monk2.name == "jill",
  "The monkey 2's name is jill.")

monk1.name = "bond"                                 ⑤
monk2.name = "moneypenny"

assert(monk1.name == "bond",                        ⑥
  "The monkey 1's name is now bond.")
assert(monk2.name == "moneypenny",
  "The monkey 2's name is now moneypenny.")

monkey.Monkey.tribe = "Party Monkey"                ⑦

assert(monk1.tribe == "Party Monkey",               ⑧
  "The monkey's tribe is Party Monkey.")
assert(monk2.tribe == "Party Monkey",
  "The monkey's tribe is Party Monkey.")
```

① We import the `monkey` module first. The variable `monkey` is a table,

and it is primarily used as a namespace.

② We call `Monkey`'s constructor with two different names for testing. They return instances (tables) of the `Monkey` class.

③ We can access the class variable `tribe` as the field of a class table, e.g., `Monkey.tribe`, or as the field of an instance table, e.g., `monk1.tribe` or `monk2.tribe`.

④ We verify that the instance variable `name` has been correctly set for each instance/table.

⑤ We can update the instance variables, independently of other instance/tables.

⑥ We then verify that their `name` fields are correctly updated.

⑦ We can also update the class variable, e.g., `tribe`, thanks to the trick, `Monkey.__newindex = Monkey`. We could have also used an instance-based syntax, e.g., `monk1.tribe = "Party Monkey"`.

⑧ We verify that the class variable `tribe` is correctly updated in both instances (since it is just one table field shared across of all instances of `Monkey`).

If you run this unit test script, and if there is no failure (e.g., no assert output), then that means that the unit test succeeded.

Here's another Lua program, which essentially tests the same things as the above test script.

main.lua

```
local monkey = require("monkey")

local monk1 = monkey.Monkey("joe")
local monk2 = monkey.Monkey("jill")

monk1:dance()
monk2:dance()
```

```
monk1:greeting()
monk2:greeting()

monk1.name = "bond"
monk2.name = "moneypenny"

monk1:greeting()
monk2:greeting()

monkey.Monkey.tribe = "Party Monkey"

monk1:dance()
monk2:dance()
```

If we run this program, then it will generate an output similar to the following:

```
Dance, dance, dance. We are Dance Monkey
Dance, dance, dance. We are Dance Monkey
hello, I'm monkey joe
hello, I'm monkey jill
hello, I'm monkey bond
hello, I'm monkey moneypenny
Dance, dance, dance. We are Party Monkey
Dance, dance, dance. We are Party Monkey
```

One can also easily *emulate* the "class inheritance" as well, e.g., using essentially the same techniques. We will leave it as an exercise to the readers. One thing to note, however, is that the simplicity always has a premium, especially when you program in the minimalistic languages like Lua.

Chapter 21. The OS Functions

A number of `os` library functions from the table `os` are included in this chapter. The `os.rename` and `os.remove` functions are described in the next chapter.

21.1. The `os.date` Function

```
os.date("*t", time = nil) -> table
os.date(format = "%c", time = nil) -> string
```

The `os.date` function returns a formatted string, or a table containing date and time, corresponding to the given `time`. When `time` is not provided, the current time is used. If `format` starts with `!`, then the date is formatted in Coordinated Universal Time.

If `format` is the string `"*t"`, then date returns a table with the following fields: *year, month (1-12), day (1-31), hour (0-23), min (0-59), sec (0-61), wday* (weekday, 1-7, Sunday through Saturday), *yday* (day of the year, 1-366), and an optional boolean *isdst* (daylight saving time flag).

If `format` is not `"*t"`, then date returns the date as a string, formatted according to the same rules as the ISO C function `strftime`. (Refer to the `strftime` documentation, e.g., on the Web, for the list of the supported format strings.) The default format `"%c"` gives a human-readable date and time representation using the current locale.

```
> os.date()
Mon Oct 24 20:24:03 2022
```

```
> os.date("%c", 1665891677)
```

```
Sat Oct 15 22:41:17 2022
> os.date("%D", 1665891677)
10/15/22
```

```
> t = os.date("*t")
> for k, v in pairs(t) do
>> print(k, v)
>> end
year    2022
isdst   false
min     24
sec     54
hour    20
day     24
yday    297
month   10
wday    2
```

21.2. The os.time Function

```
os.time(tbl = nil) -> number
```

The os.time function returns a Unix epoch time, e.g., on POSIX-compliant systems or on Windows, representing the local date and time specified by the given table tbl. When tbl is not specified, the current time is used.

The meaning of the return value of os.time, a number, is generally implementation- and system- dependent. An epoch time is the number of seconds since the given start of the time. (The Unix epoch time is based on the epoch, *00:00 AM on January 1st, 1970, UTC*.)

```
> os.time()
```

```
1665891677
> os.time{year=2023, month=1, day=1}
1672596000
```

21.3. The `os.clock` Function

```
os.clock() -> number
```

The `os.clock` function returns an approximation of the amount in seconds of CPU time used by the program, as returned by the underlying ISO C function `clock`.

```
$ lua
Lua 5.4.4  Copyright (C) 1994-2022 Lua.org, PUC-Rio
> os.clock()
0.033748
> os.clock()
0.035476
> os.clock()
0.042967
>
```

21.4. The `os.getenv` Function

```
os.getenv(varname) -> string
```

The `os.getenv` function returns the value of the process environment variable `varname`. If such a variable is not found, then it returns `nil`.

```
> os.getenv("HOME")            ①
/home/harry
```

```
> os.getenv("WORK")            ②
nil
```

① An environment variable *HOME* is set on the current shell (as is common on a Unix/Linux shell). Hence it returns its value.

② An environment variable *WORK* is not set, and its value is `nil`.

21.5. The `os.execute` Function

```
os.execute() -> boolean
os.execute(command) -> boolean, string, integer
```

The `os.execute` function passes the `command` to an operating system shell, if provided, so that it can be executed by the shell. It returns three values.

- The first value indicates whether the `command` terminated successfully:
  - `true`, if the command terminated successfully,
  - `false`, otherwise.
- The second and third values indicate whether the command ended due to a signal or not:
  - A string `"exit"` and a number representing the exit status of the command if the command exited normally,
  - A string `"signal"` and the signal that terminated the command.

When the `os.execute` function is called without an argument, it returns a boolean that is true if a shell is available.

```
> os.execute()
```

```
true
```

```
> os.execute('pwd')
/home/harry/projects/codeandtips/lua/sundry
true    exit    0
> os.execute('ls')
concatdemo.lua  sortdemo.lua  ...
true    exit    0
> os.execute('ls -l')
total 36
-rw-r--r-- 1 harry harry 136 Oct 15 17:32 concatdemo.lua
-rw-r--r-- 1 harry harry 284 Oct 15 16:19 sortdemo.lua
...
true    exit    0
```

21.6. The os.exit Function

```
os.exit(code = true, close = false)
```

The os.exit function calls the corresponding ISO C function exit to terminate the host program.

When the argument code is true, the host program returns the status EXIT_SUCCESS upon termination. If is called with code = false, then the return code is EXIT_FAILURE. Otherwise, if the os.exit is called with a numeric code, then the exit status is this same number.

If the optional second argument close is true, it closes the Lua state before exiting.

21.7. The os.setlocale Function

```
os.setlocale(locale = nil, category = "all") -> string
```

The os.setlocale function sets the current locale of the program with a given locale, if provided.

- If locale is "", the current locale is set to an implementation-defined native locale.
- If locale is "C", the current locale is set to the standard C locale.
- Otherwise, locale is a system-dependent string.

If the locale cannot be set according to request, it return nil. If successful, then it returns the name of the new locale.

If os.setlocale is called with locale = nil, then it returns the name of the current locale for the given category.

The category is an optional string describing which category to change. The valid values are "all", "collate", "ctype", "monetary", "numeric", and "time".

```
> os.setlocale()
LC_CTYPE=en_US.UTF-
8;LC_NUMERIC=C;LC_TIME=C;LC_COLLATE=C;LC_MONETARY=C;LC_MESSAGE
S=C;LC_PAPER=C;LC_NAME=C;LC_ADDRESS=C;LC_TELEPHONE=C;LC_MEASUR
EMENT=C;LC_IDENTIFICATION=C
> os.setlocale("fake")
nil
> os.setlocale("")
C.UTF-8
> os.setlocale("en_US.UTF-8")
en_US.UTF-8
```

Chapter 22. The I/O and File System Functions

Lua's input/output and file system related functions are distributed over three tables, `os`, `io`, and `file`.

The functions in the `io` table use implicit or default file handles, such as `io.stdin`, `io.stdout`, and `io.stderr`, whereas the functions in the `file` table use explicit file handles, e.g., those returned by `io.open`.

22.1. The `os.rename` Function

```
os.rename(oldname, newname) -> true | nil, string, integer
```

The `os.rename` function renames the file or directory named `oldname` to `newname`.

- The `os.rename()` call returns `true` if the rename operation was successful.
- Otherwise, it returns `nil`, and a string describing the error and the error code.

22.2. The `os.remove` Function

```
os.remove(filename) -> true | nil, string, integer
```

The `os.remove` function deletes the file with the given `filename`. On a POSIX system, it can be used with empty directories as well.

- The `os.remove()` call returns `true` if the system successfully

deleted the given file.

- Otherwise, it returns `nil`, and a string describing the error and the error code.

22.3. The `io.read` Function

```
io.read(···) -> string | integer
```

The `io.read` function is equivalent to `io.input():read(···)` using the current default input file.

22.4. The `io.write` Function

```
io.write(···) -> file
```

The `io.write` function is equivalent to `io.output():write(···)` using the current default output file.

Basic I/O Example

As indicated, Lua's builtin `print` function is primarily used for debugging during the development, just like JavaScript's `console.log`. The `io.read` and `io.write` functions, and other `io` functions, should be normally used for input and output.

Here's an example:

basicio.lua

```
local basicio = {}

function basicio.ioDemo()
```

```
  local name

  repeat
    io.write("Hi, what is your name? ")
    name = io.read()
    if name ~= "" then
      io.write("Welcome, " .. name .. "!\n")
    else
      io.write("Bye, whoever you are~~\n")
    end
  until name == ""
end

return basicio
```

If we run this in the Lua REPL,

```
> b = require("basicio")
> b.ioDemo()
Hi, what is your name? harry
Welcome, harry!
Hi, what is your name? sally
Welcome, sally!
Hi, what is your name?
Bye, whoever you are~~
```

22.5. The io.input Function

The io.input function has three forms.

```
io.input() -> file
io.input(filename)
io.input(file)
```

- When `io.input()` is called without arguments, it returns the current default input file.
- When it is called with a file name `filename`, it opens the named file (in text mode). If successful, it sets the file handle as the default input file.
- When it is called with a file handle `file`, it simply sets this file handle as the default input file.

In case of an error, `io.input` raises the error.

22.6. The `io.output` Function

The `io.output` function works similar to `io.input`, and it likewise has three forms.

```
io.output() -> file
io.output(filename)
io.output(file)
```

- When `io.output()` is called without arguments, it returns the current default output file.
- When it is called with a file name `filename`, it opens the named file (in text mode). If successful, it sets the file handle as the default output file.
- When it is called with a file handle `file`, it simply sets this file handle as the default output file.

In case of an error, `io.output()` raises the error instead of returning an error code.

File I/O Example I

Here's a simple example that does a "file copy". That is, the `fileio.fileDemo` function reads the content of the given input file (hardcoded to *hello.txt,* in this demo) and writes it to the given output file (hardcoded to *byebye.txt*).

It is written as a table-based module, in a file *fileio.lua.*

fileio.lua

```
local fileio = {}

function fileio.fileDemo()
  local ifile = "hello.txt"
  local ofile = "byebye.txt"
  io.input(ifile)
  io.output(ofile)

  local text = io.read()
  repeat
    io.write(text, "\n")
    text = io.read()
  until text == nil
end

return fileio
```

Note that the `io.read` function returns `nil` when it reaches the end of file (EOF). The `repeat` loop can also be written as the following `while` loop. They are equivalent in this particular context.

```
while text ~= nil do
  io.write(text, "\n")
  text = io.read()
end
```

If we call this `fileio.fileDemo` function in REPL,

```
> f = require("fileio")
> os.execute("cat hello.txt")
Hello, everyone!
You too~~~~~~~~~~
true    exit    0
> f.fileDemo()
> io.flush()
true
> os.execute("cat byebye.txt")
Hello, everyone!
You too~~~~~~~~~~
true    exit    0
```

22.7. The `io.open` Function

```
io.open(filename, mode = "r") -> file
```

The `io.open` function attempts to open a file with the given `filename`, in the file mode specified in the `mode` argument (a string). If the file is successfully opened, then it return a new file handle.

The `mode` argument can be any of the following:

`"r"`	Read mode.
`"w"`	Write mode.
`"a"`	Append mode.
`"r+"`	Read update mode. All existing data is preserved.
`"w+"`	Write update mode. All existing data is erased.

`"a+"` Append update mode. All data is preserved, and writing is only allowed at the end of the file.

The mode string can also have a suffix `b` at the end, which is needed in some systems to open the file in binary mode.

22.8. The `io.lines` Function

```
io.lines() -> iterator
io.lines(filename, ···) -> iterator
```

The `io.lines` function attempts to open a file with the given `filename` in the read mode, if provided. If the given file is successfully opened, it returns an iterator function. The `io.lines(filename, …)` call works like `file:lines(···)` over the opened file, `file`. When the iterator function fails to read any value, it automatically closes the file.

The call `io.lines()` with no argument is equivalent to `io.input():lines("l")`. That is, it iterates over the lines of the default input file. In this case, the iterator does not close the file when the loop ends.

In case of an error opening the file, `io.lines` raises the error.

22.9. The `io.flush` Function

```
io.flush()
```

The `io.flush` function is equivalent to `io.output():flush()`.

22.10. The `io.close` Function

```
io.close()
io.close(file)
```

The `io.close(file)` call is equivalent to `file:close()`. When it is called with no `file` argument, it closes the current default output file. That is, `io.close()` is equivalent to `io.output().close()`.

22.11. The `io.type` Function

```
io.type(obj) -> string
```

The `io.type` function checks whether `obj` is a valid file handle. It returns

- A string `"file"` if `obj` is an open file handle,
- A string `"closed file"` if `obj` is a closed file handle, or
- `nil` if `obj` is not a file handle.

22.12. The `file.read` Function

```
file:read(... = "l") -> string | integer
```

The `file:read` function, or method, reads the file, according to the given formats. For each format, the function returns a string or a number with the characters read.

If the `file:read()` call fails to read data with the specified format, it returns `nil`.

The available formats are

`"l"` Reads the next line skipping the end of line, returning fail on end of file.

`"L"` Reads the next line keeping the end-of-line character (if present), returning `nil` on end of file. The formats `"l"` and `"L"` can be used only with text files.

`"n"` Reads a number and returns it as a float or integer, following the lexical conventions of Lua. (The numeral may have leading whitespaces and a sign.) This format always reads the longest input sequence that is a valid prefix for a numeral literal.

`"a"` Reads the whole file, starting at the current position. On end of file, it returns the empty string (`""`). This format never fails.

Number Reads a string with up to this number of bytes, returning `nil` on end of file. If number is zero, it reads nothing and returns an empty string, or `nil` on end of file.

22.13. The `file.write` Function

```
file:write(···) -> file
```

The `file:write` function writes the given arguments to the file. The arguments must be strings or numbers. In case of success, this function returns the same `file`.

File I/O Example II

The earlier "file copy" example module can be rewritten as follows, using the file:read and file:write functions.

filedemo.lua

```
local filedemo = {}

function filedemo.fileDemo()
  local filei, fileo = "hello.txt", "byebye.txt"

  local fi = io.open(filei, "r")
  if fi == nil or io.type(fi) ~= "file" then
    print("Failed to open input file", filei)
    return
  end
  local fo = io.open(fileo, "w+")
  if fo == nil or io.type(fo) ~= "file" then
    print("Failed to open output file", fileo)
    return
  end
  fo:setvbuf("full")

  local text = fi:read()
  while text ~= nil do
    fo:write(text, "\n")
    text = fi:read()
  end
  fo:flush()

  local si = fi:close()
  print("Input file closed", si)
  local so = fo:close()
  print("Output file closed", so)
end

return filedemo
```

22.14. The `file.lines` Function

```
file:lines(··· = "l") -> iterator
```

The `file:lines` function returns an iterator function that, each time it is called, reads the file according to the given formats. Here's a simple example:

```
> f = io.open("hello.txt")
> for l in f:lines() do
>> print(l)
>> end
Hello, everyone!
You too~~~~~~~~~
> f:close()
true
```

22.15. The `file.flush` Function

```
file:flush()
```

The `file:flush` saves any written, and buffered, data to file.

22.16. The `file.close` Function

```
file:close()
```

The `file:close` function closes the file.

Chapter 23. Error Handling

Lua supports both errors and warnings. Errors interrupt the normal flow of the program unless they are caught. Warnings, on the other hand, do not interfere with the program execution, and they are generally used to generate messages to the user.

23.1. The `error` Function

Lua code can explicitly raise an error by calling the `error` function.

```
error(message, level = 1)
```

The `error` function raises an error with the given `message` as the error object. This function does not return. Calling `error` passes control to the host program, or the Lua interpreter.

The `level` argument specifies the "error position". For example, the "level 1" indicates that the error occurred in a function where this error function was called. The "level 2" indicates that the error occurred in a caller of this function where the `error()` function was called, etc. Specifying `0` for `level` removes this error position information from the error message.

23.2. The `assert` Function

```
assert(v, message = "assertion failed!", ...) -> ... | error
```

The `assert` function raises an `error` if the given expression `v` evaluates to false at run time (i.e., `nil` or false). Otherwise, it returns all its arguments. (Note the type of the first argument need not be

boolean.) In case of error, the second `message` argument is the error object, and the rest of the arguments are ignored.

The `assert` function is often used when the program execution cannot continue unless a certain condition is met. In Lua, the `assert` statement is also used in the unit testing. For example,

```
$ lua
Lua 5.4.4  Copyright (C) 1994-2022 Lua.org, PUC-Rio
> assert(1 == 1)
true
> assert("hi" == "hola")
stdin:1: assertion failed!
stack traceback:
    [C]: in function 'assert'
    stdin:1: in main chunk
    [C]: in ?
```

23.3. The `warn` Function

Lua also offers a system of warnings, which can be used to generate a message to the user. Warnings are not errors and they do not interfere with program execution.

```
warn(msg1, ···)
```

The `warn` function works in a similar way to `print`. It concatenates all string arguments to create a warning message, but using an empty string separator instead of a single white space.

One can enable/disable the propagation of the warning message by calling `warn` with `@on` and `@off` control messages, respectively. For example,

```
> warn("@off")                    ①
> warn("warning warning")         ②
> warn("@on")                     ③
> warn("real warning")
Lua warning: real warning         ④
```

① This disables the warning system.

② It generates no warning message since the warning system is currently disabled.

③ Re-enabling the warning system by sending the `@on` control message.

④ An example warning message from the Lua interpreter now that warning has been enabled.

23.4. Protected Calls (`pcall` and `xpcall`)

Lua does not have an exception handling mechanism, such as the `try - catch` statement, commonly found in other programming languages. Instead, to catch errors in Lua, we make protected calls using `pcall` and `xpcall` functions.

The function `pcall`, or `xpcall`, calls a supplied function in the *protected mode*. In the protected mode, an error raised by the given function stops its execution, control returns immediately to `pcall` (or, `xpcall`). The `pcall`/`xpcall` function then returns a status code (a boolean value).

23.4.1. The `pcall` function

```
pcall(f, arg1, ···) -> true, ...
pcall(f, arg1, ···) -> false, error
```

The `pcall` function calls the given argument function `f` with the given arguments (`arg1`, `...`) in protected mode. It catches any error from `f` and returns a status code. If the `f` call succeeds without errors, then `pcall` returns `true` as its first result. It also returns all results from the `f` call after the status code. In case of any error, `pcall` returns `false` followed by the `error` object.

For example,

```
local f = function(msg)                  ①
  error(msg)
end

-- f("Unusual error")                    ②

local s = pcall(f, "Usual error")        ③
print("Status:", s)                      ④
```

① This test function just raises an error with the given `msg`.

② This unprotected call will crash the program.

③ On the other hand, `pcall` will catch any errors.

④ The return value of `false` indicates that the call `f` raised an error.

23.4.2. The `xpcall` Function

```
xpcall(f, msgh, arg1, ...) -> true, ...
xpcall(f, msgh, arg1, ...) -> false, error
```

The `xpcall` function works the same way as `pcall`, except that it sets a new message handler `msgh`.

Chapter 24. Concurrency

Lua supports concurrent programming via `thread` objects and what is generally known as the `coroutine` in programming.

Lua's threads are slightly different from the threads commonly used in other programming languages. Lua's concurrency model is often called "collaborative multithreading".

The standard library provides a few functions in the `coroutine` table to support the coroutine management, e.g., to create new coroutines, suspend running coroutines, and resume suspended coroutines, etc.

24.1. Coroutines

A coroutine is similar to a subroutine or procedure, which is commonly called the "function" in imperative programming languages including Lua (although they are not real functions, in the mathematical sense).

Unlike a function, which is called and returned at most once after performing a given task (e.g., until it is called again), a coroutine can be entered, at different points of execution, and it can be suspended and resumed, e.g., from the last suspended points. It maintains its internal state throughout its lifetime, that is, until it is terminated, explicitly or otherwise. This is sometimes called the "multiple entry".

24.2. Creating Coroutines

A coroutine is created using the `coroutine.create` function. The main function of the coroutine is provided as an argument to `coroutine.create`. It returns a `thread` object, in the *suspended* state.

```
> t = coroutine.create(function() end)          ①
> type(t)                                       ②
```

```
thread
> coroutine.status(t)                    ③
suspended
```

① The `coroutine.create` function returns a thread object. In Lua, the terms, coroutines and threads, are pretty much synonymously used (although one may argue that `thread` is a type and `coroutine` is a threaded flow of control).

② The type of the returned object is `thread`.

③ We can check the status of the thread using the `coroutine.status` function. A newly created thread's initial status is `"suspended"`.

24.2.1. The `coroutine.create` function

```
coroutine.create(f) -> thread
```

The `coroutine.create` function creates a new coroutine with the given function, `f`. It returns a coroutine object of the `thread` type, with the status *suspended*. It does *not* start the coroutine.

24.2.2. The `coroutine.status` function

```
coroutine.status(co) -> string
```

The `coroutine.status` function returns the current status of the given coroutine `co`, as a string:

`running`	If the coroutine is running.
`suspended`	If the coroutine is waiting for the next *resume*.
`normal`	If the coroutine is active but not running, and

`dead`	If the body function is terminated.

24.2.3. The `coroutine.isyieldable` function

```
coroutine.isyieldable(co = nil) -> boolean
```

The `coroutine.isyieldable` function returns `true` when the coroutine `co` can yield, regardless of its current status. All Lua coroutines are yieldable, except the main thread that started the current chunk. When it is called without an argument, or `co == nil`, it uses the current running coroutine.

24.3. Starting and Resuming Coroutines

You can start, or resume, a coroutine/thread by calling the `coroutine.resume` function with the given thread object as its first argument. The rest of the arguments, if any, are passed to the main or body function of the coroutine. After the coroutine starts running, it runs until it terminates or yields.

```
> t = coroutine.create(function() end)          ①
> coroutine.resume(t)                           ②
true
> coroutine.status(t)                           ③
dead
```

① The `coroutine.create` function call creates a new `thread` object in the *suspended* state.

② The first call of `coroutine.resume(t)` starts the given coroutine. It returns `true` if successful. The coroutine runs until the first `coroutine.yield` function call statement or until the function terminates (naturally or due to an error).

③ This particular body function of `t` did not do much and just returned. Hence the coroutine status of `t` is `"dead"`. A dead coroutine cannot be resumed again.

24.3.1. The `coroutine.resume` function

```
coroutine.resume(co, val1, ···) -> boolean, ...
```

The `coroutine.resume` function starts or resumes the given coroutine `co` in the *suspended* state.

When it is called the first time on a newly created coroutine object, it calls the body function, with the arguments, `val1`, etc., if any. When `coroutine.resume` restarts a previously yielded coroutine, they are returned as the return value of that `yield` function call.

The `coroutine.resume` call returns a boolean value to indicate the success or failure status. If it has succeeded, it returns `true` and, in addition, any return values from the body function. If it has failed, then it returns `false` and any error message.

24.3.2. The `coroutine.running` function

```
coroutine.running() -> co, boolean
```

The `coroutine.running` function returns the *currently running* coroutine object (in which this function is called). In addition, it returns `true` if it is the main coroutine (which is not-yieldable). Otherwise, it returns `false`.

24.4. Suspending and Resuming

A running coroutine "yields" by calling `coroutine.yield`, which makes the previous `coroutine.resume` to immediately return. The coroutine remains in the suspended state until `coroutine.resume` is called again with the corresponding thread object or until the coroutine is explicitly terminated.

The next time the suspended coroutine is resumed, it continues its execution from right after where it previously yielded. Here's a sample program that calls `coroutine.resume` multiple times.

```
local function f(a)
  print("Inside f:\tStarted with " .. tostring(a))
  coroutine.yield("huh?", "huhhuh?")
  print("Inside f:\tResumed with " .. tostring(a))
end

local t = coroutine.create(f)

print("Status: ", coroutine.status(t))
print("Resumed:", coroutine.resume(t, 10))
print("Status: ", coroutine.status(t))
print("Resumed:", coroutine.resume(t, 20))
print("Status: ", coroutine.status(t))

print("Resumed:", coroutine.resume(t, 30))
print("Status: ", coroutine.status(t))
```

If you run this as a script, you will get an output like this:

```
Status:         suspended
Inside f:       Started with 10
Resumed:        true    huh?    huhhuh?
Status:         suspended
Inside f:       Resumed with 10
```

```
Resumed:        true
Status:         dead
```

24.4.1. The `coroutine.yield` function

```
coroutine.yield(···) -> ...
```

The `coroutine.yield` function suspends the current coroutine's execution (in which this `yield()` is called). Any arguments provided to this call are passed as additional return values to the following `resume()` function call.

Likewise, any extra arguments provided to `resume()` are returned as the result of this `yield()` call.

24.4.2. The `coroutine.wrap` function

```
coroutine.wrap(f) -> function
```

The `coroutine.wrap` function creates a new coroutine, with the given function `f` as its body. It returns an iterator-like function that resumes the coroutine each time it is called.

```
returned_function(...) -> ...
```

Any arguments passed to this function are used as the extra arguments to the (implicit) `coroutine.resume()` call. In case of error, it closes the coroutine and throws an error. If the call is successful, it returns the same values returned by `coroutine.resume`, excluding the first result `true`.

24.5. Coroutine Termination

A coroutine can terminate unexpectedly if there is an unprotected error. In such as case, the last `coroutine.resume()` call returns `false` with an error object. Otherwise, it terminates its execution when its body function returns. In the normal termination, `coroutine.resume()` returns `true`, plus any values returned by the coroutine's body function. Here's a sample script and its output:

```
local function f(a)
  print("Inside f:\tStarted with " .. tostring(a))
  coroutine.yield("uh?")
  print("Inside f:\tResumed with " .. tostring(a))
end

local t1 = coroutine.create(f)

print("Status: ", coroutine.status(t1))
print("Resumed:", coroutine.resume(t1, 10))
print("Status: ", coroutine.status(t1))

print("Closing:", coroutine.close(t1))
print("Status: ", coroutine.status(t1))

print("Resumed:", coroutine.resume(t1, 20))
print("Status: ", coroutine.status(t1))
```

```
Status:         suspended
Inside f:       Started with 10
Resumed:        true    uh?
Status:         suspended
Closing:        true
Status:         dead
Resumed:        false   cannot resume dead coroutine
Status:         dead
```

24.5.1. The `coroutine.close` function

```
coroutine.close(co) -> boolean, error
```

The `coroutine.close` function closes the coroutine `co` in the `suspended` state. It first closes all pending to-be-closed variables, and then updates the coroutine's status to `dead`. Calling it on a `dead` coroutine is a no-op. If successful, it returns `true`. Otherwise, it returns `false` and any error object.

24.6. Coroutine Example

As an example, here's a simple one-producer - one-consumer version of the classic producer-consumer problem [https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem]. (Note that we do not add much comments or annotations in this final example in the book.)

Here's the *producer* module.

producer.lua

```
local producer = {}

local produce_count = 5

function producer.produce()
  return coroutine.create(function()
    for i = 1, produce_count do
      print("Produced:", i)
      coroutine.yield(i)
    end
  end)
end

return producer
```

The `producer.produce()` function creates and returns a coroutine, which is supposed to print 5 lines of output. Next, the *consumer* module:

consumer.lua

```
local consumer = {}

local consume_count = 3

function consumer.consume(f)
  local count = 0;
  for i = 1, consume_count do
    local status, value = coroutine.resume(f)
    print("status:", status, "received:", value)
    if status then
      count = count + 1
    end
  end
  return count
end

return consumer
```

The `consumer.consume()` function does 3 iterations of waking up the producer thread and printing out its status and the value produced by the producer.

For convenience, the interaction between the producer and consumer is encapsulated in the *driver* module:

driver.lua

```
local driver = {}

local producer = require("producer")
local consumer = require("consumer")

function driver.drive()
```

```
  local res = consumer.consume(producer.produce())
  return res
end

return driver
```

Now, the "main" program:

main.lua

```
local driver = require("driver")

local res = driver.drive()
print("Final result =", res)
```

Here's a sample output:

```
Produced:       1
status: true    received:       1
Produced:       2
status: true    received:       2
Produced:       3
status: true    received:       3
Final result =  3
```

This is a very basic use of Lua's coroutine APIs, and a lot more can be done using these APIs. In this particular example, we merely demonstrate that the two separate routines, e.g., the coroutines produced by `producer.produce()` and `consumer.consume()`, can run "concurrently", as illustrated by the above output.

A. How to Use This Book

> Tell me and I forget. Teach me and I remember. Involve me and I learn.
>
> — Benjamin Franklin

The books in this "Mini Reference" series are written for a wide audience. It means that some readers will find this particular book "too easy" and some readers will find this book "too difficult", depending on their prior experience related to programming. That's quite all right. Different readers will get different things out of this book. At the end of the day, learning is a skill, which we all can learn to get better at. Here are some quick pointers in case you need some advice.

First of all, books like this are bound to have some errors, and some typos. We go through multiple revisions, and every time we do that there is a finite chance to introduce new errors. We know that some people have strong opinions on this, but you should get over it. Even after spending millions of dollars, a rocket launch can go wrong. All non-trivial software have some amount of bugs.

Although it's a cliche, there are two kinds of people in this world. Some see a "glass half full". Some see a "glass half empty". *This book has a lot to offer*. As a general note, we encourage the readers to view the world as "half full" rather than to focus too much on negative things. *Despite* some (small) possible errors, and formatting issues, you will get *a lot* out of this book if you have the right attitude.

There is this book called *Algorithms to Live By*, which came out several years ago, and it became an instant best seller. There are now many similar books, copycats, published since then. The book is written for "laypeople", and illustrate how computer science concepts like specific algorithms can be useful in everyday life.

Inspired by this, we have some concrete suggestions on how to best read this book. This is *one* suggestion which you can take into account while using this book. As stated, ultimately, whatever works for you is the best way for you.

Most of the readers reading this book should be familiar with some basic algorithm concepts. When you do a graph search, there are two major ways to traverse all the nodes in a graph. One is called the "depth first search", and the other is called the "breadth first search". At the risk of oversimplifying, when you read a tutorial style book, you go through the book from beginning to end. Note that the book content is generally organized in a tree structure. There are chapters, and each chapter includes sections, and so forth. Reading a book sequentially often corresponds to the *depth first traversal*.

On the other hand, for reference-style books like this one, which are written to cover broad and wide range of topics, and which have many interdependencies among the topics, it is often best to adopt the *breadth first traversal*.

This advice should be especially useful to new-comers to the language. The core concepts of any (non-trivial) programming language are all interconnected. That's the way it is. When you read an earlier part of the book, which may depend on the concepts explained later in the book, you can either ignore the things you don't understand and move on, or you can flip through the book to go back and forth. It's up to you. One thing you don't want to do is to get stuck in one place, and be frustrated and feel resentful (toward the book).

The best way to read books like this one is through "multiple passes", again using a programming jargon. The first time, you only try to get the high-level concepts. At each iteration, you try to get more and more details. It is really up to you, and only you can tell, as to how many passes would be required to get much of what this book has to offer.

Again, *good luck!*

Index

A

C

D

E

F

G

H

I

M

N

O

P

R

S

T

U

V

W

X

Y

About the Author

Harry Yoon has been programming for over three decades. He has used over 20 different programming languages in his academic and professional career. His experience spans broad areas from scientific programming and machine learning to enterprise software and Web and mobile app development.

He *occasionally* hangs out on social media:

- Instagram: @codeandtips [https://www.instagram.com/codeandtips/]
- TikTok: @codeandtips [https://tiktok.com/@codeandtips]
- Twitter: @codeandtips [https://twitter.com/codeandtips]
- YouTube: @codeandtips [https://www.youtube.com/@codeandtips]
- Reddit: r/codeandtips [https://www.reddit.com/r/codeandtips/]

Other Lua Books by the Author

- Learn Coding with Lua: A Slow and Gentle Introduction to Basic Programming for Total Beginners with Step-by-Step Instructions [https://www.amazon.com/dp/B0BF19Q3DV/]

About the Series

We are creating a number of books under the series title, *A Hitchhiker's Guide to the Modern Programming Languages*. We cover essential syntax of the 12 select languages in 100 pages or so, Go, C#, Python, Typescript, Rust, C++, Java, Julia, JavaScript, Haskell, Scala, and Lua.

These are all very interesting and widely used languages that can teach you different ways of programming, and more importantly, different ways of thinking.

All Books in the Series

- Go Mini Reference [https://www.amazon.com/dp/B09V5QXTCC/]
- Modern C# Mini Reference [https://www.amazon.com/dp/B0B57PXLFC/]
- Python Mini Reference [https://www.amazon.com/dp/B0B2QJD6P8/]
- Typescript Mini Reference [https://www.amazon.com/dp/B0B54537JK/]
- Rust Mini Reference [https://www.amazon.com/dp/B09Y74PH2B/]
- C++20 Mini Reference [https://www.amazon.com/dp/B0B5YLXLB3/]
- Modern Java Mini Reference [https://www.amazon.com/dp/B0B75PCHW2/]
- Julia Mini Reference [https://www.amazon.com/dp/B0B6PZ2BCJ/]
- JavaScript Mini Reference [https://www.amazon.com/dp/B0B75RZLRB/]
- Haskell Mini Reference [https://www.amazon.com/dp/B09X8PLG9P/]
- Scala 3 Mini Reference [https://www.amazon.com/dp/B0B95Y6584/]
- Lua Mini Reference [https://www.amazon.com/dp/B09V95T452/]

Community Support

We are building a website for programmers, from beginners to more experienced. It covers various coding-related topics from algorithms to machine learning, and from design patterns to cybersecurity, and more. You can also find some sample code in the GitLab repositories.

- www.codeandtips.com
- gitlab.com/codeandtips

Mailing List

Please join our mailing list, join@codingbookspress.com, to receive coding tips and other news from **Coding Books Press**, including free, or discounted, book promotions. If we find any significant errors in the book, then we will send you an updated version of the book (in PDF). Advance review copies will be made available to select members on the list before new books are published.

Request for Feedback

If you find any errors or typos, or if any part of the book is not very clear to you, or if you have any general suggestions or comments regarding the book, then please let us know. Although we cannot answer all the questions and emails, we will try our best to address the issues that are brought to our attention.

- feedback@codingbookspress.com

Please note that creating and publishing quality books takes a great deal of time and effort, and we really appreciate the readers' feedback.

Revision 1.1.3, 2023-05-14