Chapter 1: Getting Started

- Each piece of code that Lua executes is called a *chunk*
- Lua needs no seperator between consecutive statements (but you can use a semicolon if you wish)
- To exit the interpreter, use `os.exit()`
- To parse a file from Lua, use `dofile("filename.lua")`
- Identifiers can be any string of letters, digits and underscores not beginning with a digit
- Lua 5.2 accepts only English letters for identifiers (`a-z` and `A-Z`)
- Comments start with a double hypen (`--`) and go to the rest of the line
- Long comments start with a `--[[` and end with `]]`
- Global variables do not need declarations, they are `nil` by default
- In interactive mode, prepending and equals sign (`=`) to any expression prints the result of that expression
- Any arguments to a script are in the global variable `arg` by default

Chapter 2: Types and Values

- Lua has eight basic types: `nil`, `boolean`, `number`, `string`, `userdata`, `function`, `thread` and `table`
- The type of a variable can be checked using the `type()` function, which returns a string representing the type of the given variable
- Functions are *first-class values* in Lua, they can be used like any other type of variables
- Lua uses `nil` as a kind of non-value, representing the absence of a useful value
- **All** numbers in Lua are real (double) *floating-point numbers* (there is no integer type)
- In the Lua number type, any integer up to 2ˆ53 has an exact representation
- Due to using a double type, there can be **rounding errors**: `12.7-20+7.3` is not exactly zero because both 12.7 and 7.3 do not have an *exact representation*
- Number literals can be written with both an optional floating-point part and exponent, and can be either in base ten or hexadecimal (with the `0x` prefix)
- Strings can contain any characters (null, any UTF-8 characters, etc.)
- Strings are *immutable* in Lua (characters cannot be modified)
- The length of a string can be acquired with the *length operator* (`#`)
- Strings can be delimited by single and double quotation marks (`'str'` and `"str"`) as well as with double square brackets (`[[str]]`)
- Strings can be concatenated with double periods (`..`)
- Strings and numbers can be converted with `tostring()` and `tonumber()`

- The only real data type in Lua are tables, which can be used to construct *arrays* (sequences) as well as *records*
- Tables are handled **by reference**
- To access the member `abc` of table `t`, both `t["abc"]` and `t.abc` can be used
- Lua global variables are stored in a table
- Lua arrays are tables that use **numbers from 1 to n** as indexes
- The length of Lua arrays without *holes* (embedded nils) can be aquired using the *length operator* (`#`)
- Userdata variables allow C data to be stored in Lua variables

Chapter 3: Expressions

- Exponentiation is done in Lua with the caret (`^`)
- Modulus is obtained from a number with the percent sign (`%`)
- The fractional and integer part of a number can be obtained using the modulus operator (`n%1` for the former and `n-n%1` for the latter)
- The negation of the equality operator in Lua is a tilde and an equals sign combined (`~=`)
- Tables and userdata are compared **by reference**
- Strings are compared in **alphabetical order** (as determined by the locale)
- Logical operators (`not`, `and`, `or`) use *short-cut evaluation*, so `f() or error()` is only going to call `error()` if `f()` returns false
- The Lua idiom `x = x or v` sets `x` to `v` only if `x` is not `nil` or `false`
- The Lua idiom `c and t or f` returns `t` if `c` evaluates to `true`, and `f` otherwise (unless `t` evaluates to `false`)
- Concatenation in Lua is done with two dots (`..`). If one of the operands is a number, it is converted to a string automatically
- The *concatenation operator* does not modify it's operands
- The length operator (`#`) works on strings and tables, on the latter it gives the length of the sequence represented by it, a sequence being a table where numeric keys go from 1 to n without any holes (embedded nils)
- Tables can be constructed by a few different constructors:

1. **List constructor**, which constructs the table to be a sequence, looks like this: `{324, "value two", true, ...}`
2. **Record constructor**, which constructs the table to be a record, looks like this: `{fieldone=10, fieldtwo="value two", fieldthree=false, ...}`
3. **General constructor**, which can construct any kind of table, and it looks like this: `{["field one"]=324, ["field two"]="value two", ...}`

Chapter 4: Statements

- Lua allows *multiple assignment*, which assigns a list of values to a list of variables in one step, both lists have their elements seperated my commas
- Lua first evaluates all values and only then executes the assignments (allowing us to swap two variables with multiple assignment)
- When there are more variables than values, they are filled with `nil`s
- When there are more values than variables, they are silently discarded
- A frequent use of multiple assignment is to collect *multiple returns* from function calls
- Lua supports local variables with the keyword `local <variable name>`
- In interactive mode, **local variables don't work as expected** because every line is executed in it's own chunk
- To make local variables work in interactive mode, their use needs to be put into a *do-end block*
- Access to local variables is faster than to global ones
- A common idiom in Lua is `local foo = foo`, which creates a local variable `foo` assigns the global variable `foo` to it
- Lua supports *if-then-[[elseif-then]-else]-end*, *while-do*, *repeat-until*, *numeric for* and *generic for* control structures
- Numeric for starts at a given start value and ends at a given end value using the steps provided: `for <var> = <start>, <end>, [<step=1>] do...`
- **The value of the control variable should never be changed** (use `break` to prematurely exit the loop)
- The generic for traverses all values returned by an iterator function, like so: `for k, v in pairs(t) do...`
- There are several *iterators*: `pairs()` to traverse a table, `io.lines()` to iterate over the lines of a file, `ipairs()` to iterate over the entries of a sequence, `string.gmatch()` to iteratre over words in a string and so on
- A return statement can only appear as the last statement of a block
- Lua supports `goto` and labels, they are declared with `::labelname::` and can be jumped to with `goto labelname`
- You cannot jump into a block or out of a function and you cannot jump into the scope of a local variable
- The scope of a local variable ends on the last non-void statement of the block where the variable is defined, labels are considered void statements
- Gotos can be used to emulate functionality like `continue`

Chapter 5: Functions

- If a function has one single argument and that argument is either a string literal or a table constructor, the parentheses (in a function call) are optional

- The colon operator in Lua offers special syntax for object oriented programming: `o:method(a,b)` translates to `o.method(o,a,b)`
- You can call a function with a number of arguments different from it's number of parameters: extra arguments are thrown away, missing ones filled with nil
- Functions in Lua can return *multiple results*
- In some cases, like when the function is placed in parentheses (like so: `(f())`) or when it's used as an expression, only the first result is used
- The Lua function `table.unpack()` takes an array as input and returns the contents (using multiple return values)
- The opposite can be done with the function `table.pack()`, which turns all of it's parameter into an array and stores the size of that array in the field `n`
- Lua functions can take a variable amount of inputs with the *vararg expression* (`...`), which is used in place of the parameter list and expands to the given arguments in the function body
- *Named parameters* can be simulated in Lua by passing a table as the first and only argument, which can look like this: `func{arg1="this", arg2="that"}`

Chapter 6: More about Functions

- Functions in Lua are *first-class values* with *proper lexical scoping*, meaning that they can access variables of their enclosing functions
- Functions can be stored in tables, and passed to and returned from other functions
- Functions are *anonymous* (not bound to any name)
- A Function definition is actually an assignment
- Functions as first-class values can be used to write *callback functions* or provide a sorting strategy to `table.sort`
- Functions that get other functions as an argument are called *higher-order functions*
- The variables of the parent function that a function defined inside another function can access are neither local nor global variables, these are called *nonlocal variables* (these are said to *escape* their original scope)
- *Closures* make use of proper lexical scoping, they are functions with access to nonlocal variables
- Closures can be used to create sandboxes by redefining functions in a more limited manner and hiding the original functions
- Functions can also be stored in local variables, and Lua has syntactic sugar to do this (by prepending `local` before a function declaration)
- When using indirect local recursive functions, they need a kind of *forward*

declaration to indicate that they will be local (with `local name`) and they then need to be defined without the local function syntactic sugar
- Lua does proper *tail-call elimination* (tail calls do not cost stack space)
- Tail calls need to be in the form `return func(args)`

Chapter 7: Iterators and the Generic for

- An iterator is any construction that allows you to *iterate* over the elements of a collection
- They are typically represented by functions (closures) in Lua
- A closure iterator involves two functions: the closure itself and a *factory*, which creates the closure and it's nonlocal variables
- Iterators may not be easy to write, but they are easy to use
- The generic for does all the bookkeeping for an iteration loop and it also keeps an *invariant state variable* and a *control variable*
- When the first variable, which is called the control variable, is `nil`, the loop ends
- With the invariant state and the control variable, we can write *stateless iterators* (like `ipairs()`, which is also stateless)
- Complex states can be stored in the invariant state variable by using a table
- True iterators are functions that do the iteration themselfes, they take an anonymous function as argument and call that for every element
- True iterators were popular when the for loop wasn't in Lua yet, they have some drawbacks (like difficult parallel iteration)

Chapter 8: Compilation, Execution and Errors

- Lua always *precompiles* source code to intermediate form before running it
- Lua is still considered an *interpreted language* since it is possible to execute code generated on the fly (with functions such as `load`)
- The function `loadfile` loads a Lua chunk from a file and returns a function that will call the chunk if called, or an error code
- We can use `loadfile` to run a file several times
- The `load` function is similar, but it reads its chunk from a string
- The `load` function is powerful and rather expensive, so it should be used **with care** and only when needed
- `load` compiles code in the global environment, **without lexical scoping**
- You can use vararg expressions in `file` since the code is treated as an anonymous function
- The `string.rep` function repeats a string a given number of times

- `load` can take a reader function as argument, which returns the chunk in parts
- `io.lines(filename, "*L")` returns a function that iterates over the lines in the given file
- `io.lines(filename, 1024)` is more efficient since it uses a fixed-size buffer
- The `load` and `loadfile` functions **never** have any side effects
- External chunks should be run in a *protected environment*
- Lua allows code to be distributed in precompiled form, such code is allowed anywhere normal code would be allowed as well
- Code can be precompiled with the `luac` program
- `string.dump` returns the precompiled code (as a string) of any Lua function
- **Maliciously corrupted binary code can crash the Lua interpreter or even execute user-provided machine code!**
- As a second parameter, `load` can accept a name of the chunk to be loaded for debugging purposes
- The third parameter to `load` controls what kind of chunks can be loaded ('t' for textual, 'b' for binary and 'bt' for both)
- Lua supports *dynamic linking* even though that is not standard ANSI C
- To dynamically link to a library, use `package.loadlib(libpath, funcname)`, which returns the requested function
- Often libraries are loaded with `require`, which auto-imports all functions and puts them into a package
- Whenever an error happens, Lua ends the current chunk and returns to the application
- The `assert` functions checks if it's first argument is not false, if so it returns it, else it raises an error
- Functions can return false and an error code to show errors or call the `error` function directly
- Most functions return `false` and an error code so the error can be handled
- Errors raised with `error` can be caught using the `pcall` function, which stands for *protected call*
- `pcall` takes a function to be called in protected mode as well as a level argument to tell which of the functions in the call stack is the culprit
- If we want a traceback of the error, we can use the `xpcall` function, which takes a *message handler function* (which is called before the stack unwinds)
- Two common message handlers are `debug.debug` (provides interactive console) and `debug.traceback` (builds an extended error message with the traceback)

Chapter 9: Coroutines

- Coroutines in Lua are like threads: they are a line of execution with their own stack, local variable and instruction pointer but sharing the global variables
- Coroutines run *concurrently*, not *parallel*: there's always **just one** coroutine currently running
- Coroutines are a means of *cooperative multitasking* (as opposed to *preemptive multitasking*): their execution is only suspended if they explicitly ask for it
- All coroutine functions are in the `coroutine` table
- They can be created with `coroutine.create()`, which takes a function as argument
- Coroutines are of type `thread`
- Coroutines can be in one of four states:

1. `normal`: This is the state a coroutine gets into when it calls `coroutine.resume()`: it is neither running nor suspended, since it **can't be resumed** when in this state.
2. `running`: This is the state a coroutine is in when it's running
3. `suspended` Newly created coroutines are in this state, as well as coroutines that have suspended themselves (with `coroutine.yield()`)
4. `dead`: The coroutine enters this state if the coroutine function returns, it is **not possible** to resume a dead coroutine

- Their status can be checked with `coroutine.status()`
- The real power comes from the `coroutine.yield()` function, which suspends the currently running coroutine and passes control back to the coroutine that caused it to run in the first place (with `coroutine.resume()`)
- `coroutine.resume()` runs in *protected mode*, so any error raised from within the coroutine will be returned by it, just like with `pcall()`
- The resume and yield functions can **exchange data**: an argument to any of them becomes a return value of the other
- *Symmetric coroutines* of other languages can be easily emulated in Lua
- Coroutines offer a great way to tackle the *producer-consumer problem* (the *who-has-the-main-loop* problem)
- They can also turn the caller/callee relationship inside out: now the callee can request from the caller (by resuming the caller)
- Coroutines offer a kind of *non-preemptive* (*cooperative*) multitasking, but since there is no parallelism involved, the code is easy to debug and there is no need for synchronization methods
- The cost of switching between coroutines is **really small** compared to switching between processes (as in UNIX pipes)
- They can be used to easily write iterators without having to worry about keeping a state

- The non-preemptive multitasking that they offer can be used to concurrently download files from the internet if *non-blocking sockets* are available