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 an 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 2^{53} 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 (eg. 0.1212) and exponent (eg. 3.4e-4), and can be either in base ten or hexadecimal (with the `0x` prefix, eg. 0xFFFF)
- Strings can contain any characters (null, any UTF-8 characters, etc.)
- Strings are *immutable* in Lua (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** (thus, `{} ~= {}`)
- 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, where `n` is the length of the array
- 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:

  List constructor constructs the table to be a sequence, has no notion of keys and looks like this: `{324, "value two", true, ...}`

Record constructor constructs the table to be a record, has keys which must be valid Lua identifiers and looks like this: `{fieldone=10, fieldtwo="value two", fieldthree=false, ...}`

General constructor can construct any kind of table, it's keys do not need to be valid identifiers and can in fact be of any type, 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 by 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`
- 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, they need 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` and 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 not be changed (use `break` to prematurely exit the loop)
- The *generic for* traverses all values returned by an iterator function, like so: `for key, value in pairs(table) 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 iterate over words in a string and more
- 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, out of a function or 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 additionally 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 allows us to have on key/value pair per argument, where the key is the name of that argument
- Can look like this: `copy{src="file1", dest="file2"}`

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 it can

access are neither local nor global variables, these are called *nonlocal variables* or *upvalues* (they *escape* their original scope)
- *Closures* make use of proper lexical scoping: they are functions with access to nonlocal variables
- *Nonlocal variables* persist between function calls, similar to `static` function variables in C
- 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 (the *state*)
- 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* (can be used to keep a state) and a *control variable*
- When the first variable returned by the iterator (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): these do not use nonlocal variables to keep their state
- 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
- They aren't used very much anymore since they have some drawbacks (like difficult parallel iteration)

Chapter 8: Compilation, Execution and Errors

- Lua always *precompiles* source code to intermediate form (bytecode) 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 (by executing the returned function multiple 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 `load()`ed strings 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 is raised, 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 variables 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:

  `normal` This is the state a coroutine gets into when it calls `coroutine.resume()` on some other coroutine: it is neither `running` nor `suspended`, since it can't be resumed when in this state.

  `running` This is the state a coroutine is in when it's running

  `suspended` Newly created coroutines are in this state, as well as coroutines that have suspended themselves (with `coroutine.yield()`)

  `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
- 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

Chapter 10: Complete Examples

- The *eight-queen puzzle* is solved with a configuration of eight queens on a chessboard in a way that no queen can attack another one
  - Any valid solution must have exactly one queen in each row
  - No two queens can be in the same column
  - No two queens can be directly diagonal to each other
- A *Markov-chain algorithm* can be used to generate pseudo-random text based on what words can follow a sequence of n previous words in a base text
- The resulting text is very, but not quite, random

Chapter 11: Data Structures

- Tables in Lua are *the* data structure
- All other data structures can be implemented quite efficiently on top of Lua tables
- *Arrays* in Lua are implemented by simply indexing tables with integers
- They can grow as needed, elements that aren't used (equal to `nil`) don't take up space
- Indexing commonly starts at 1, but it can start at any other value
- *Matrices and multi-dimensional arrays* can be represented as either true *multi-dimensional arrays* or as *flat arrays*
- *Two-dimensional arrays* can be implemented with arrays by using a large array for the rows, containing one array per row which holds the actual data
- Allow for the most freedom (can represent *triangular matrices*)
- *Flat arrays* can also be used to represent matrices
- These can have an integer index that is composed of the two matrix indexes
- Otherwise, they can have a string index which is the concatenation of the two matrix indexes
- Either way, they can both easily represent matrices and are also efficient when working with *sparse matrices*: only matrix elements that are not `nil` take up memory
- Keys in tables have not intrinsic order, so iterations with `pairs()` happen in no particular order
- Linked lists are particularly easy to implement in Lua: they are simply a list with a reference to the next list
- Linked lists aren't used very often since they are not really neccessary
- *Queues* and *double queues* can be implemented easily with tables that have a first and last index variables
- The first and last variables will continually increase when using the queues, but the available ~53 bits of integer precision (2^{53} representable integers) are unlikely to run out
- *Sets* and *bags* can be represented by storing the objects as the keys of a table (fast lookup)
- Objects in *sets* can either be in the set (then `set[obj]==true`) or not be in the set (then `set[name]==nil`)
- This behavior is achieved by using the object as indices of a table and setting it's value to `true`
- *Bags* (also called **multisets**) are like **sets**, but there can be duplicates
- This is again trivial to implement with tables, where `bag[obj]` is either `nil` (then `obj` isn't in `bag`) or a number describing how many times `obj` is in `bag`
- **String buffers** are sometimes needed in Lua since strings are immutable
- To read a file chunk by chunk, all the chunks can first be stored in a table

and then finally put together with `table.concat()`, which optionally takes a string to use as seperator

- File should, however, rather be read with `io.read("*a")`

Chapter 12: Data Files and Persistence

- Writing data to file is easier than reading it back, since then it needs to be *parsed*
- Coding robust input routines is always difficult
- Lua started out as a data description language
- BibTeX was one of the inspirations for the constructor syntax in Lua
- We can use plain Lua to store data, which will look like this:

```
Person{
    name = "John Doe",
    age = 35,
    email = "john@example.com",
}
```

- `Person` both describes the data and represents a Lua function call, so we only have to define a sensible callback function and run the data file
- This is a *self-describing data format*, which means that it's easy to read and edit by hand
- Lua runs fast enough to store data like this, since data description has been one of the main applications of it
- To be able to write data which needs to be read back, it needs to be put in a known state, this process is called *serialization*
- We can do this recursively in Lua
- Floating-point numbers may loose precision when written and read back in decimal form, but we can use a hexadecimal format when writing: `string.format("%a", 0.4342)`
- Strings can also be properly escaped: `string.format("%q", str)`
- Tables can be serialized recursively **only** if they do not have *cycles* (where some parts of the table refer to other parts of the same table)
- To represent cycled tables, named tables are needed

Chapter 13: Metatables and Metamethods

- *Metatables* allow us to change the behavior of a value when confronted with an undefined operation
- Whenever Lua tries to perform arithmetic operations on tables, it checks if any of them have a *metamethod* which defines this operation and runs it, otherwise it raises an error
- Tables and userdata have individual metatables, all other data types share one single metatable for all values of that type
- Lua always creates new tables without metatables
- We can use `setmetatable(t, mt)` to set or change the metatable of any type
- Any table can be the metatable of any value
- A group of related tables can share a metatable which describes their common behavior
- A table can be it's own metatable so that it describes it's own behavior
- Arithmetic metamethods are as follows:

  `__add` addition (`+`)

  `__mul` multiplication (`*`)

  `__sub` subtraction (`-`)

  `__div` division (`/`)

  `__unm` negation (`not`)

  `__mod` modulus (`%`)

  `__pow` exponentiation (`^`)

  `__concat` concatenation (`..`)

- When two tables are in an undefined expression, the left table's metamethod will be used if it exists, else the right table's metamethod will be used, and if that doesn't exist either, an error will be raised
- Metatables can also be used for *relational operators*:

  `__eq` equality (`==`)

  `__lt` less than comparison (`<`)

  `__le` less than or equal (`<=`)

- Equality comparison doesn't work if objects have different types (always returns `false`)

- Some library functions also use metamethods to change their behaviour:

  `__tostring` Used by `tostring()` to convert the table to a string

  `__metatable` Returned by `getmetatable()` if it exists, and subsequent to setting this, `setmetatable()` will raise an error (as this marks the table as protected)

  `__pairs` If defined, this is called by `pairs()` to get an iterator for the table (especially useful for proxy tables, where the actual data is not stored in the table)

  `__ipairs` Just like `__pairs`, but for the `ipairs()` function

- The behaviour of tables can also be changed with metatables:

  `__index` Called when a nonexisting index of the table it accessed. Can be either a metamethod (function) or a table, the latter is useful for using this to implement inheritance. Otherwise, it can also be used to change the default value of an empty table. This metamethod can be bypassed by using the `rawget()` function.

  `__newindex` Called when a new table index is defined. This can be bypassed by using the `rawset()` function.

- The table access metamethods allow us to write a table that has no data of it's own but simply proxies all access to another table, for example to track access or to block certain kinds of access (eg. *read-only*)

Chapter 14: The Environment

- Lua keeps all global variables in a regular table, called the *global environment*
- This table can be manipulated like any other table
- It is globally accessible by the name `_G`
- Can be used to dynamically get the contents of variables, like so: `_G["varname"]`
- However, something such as `_G["io.read"]` won't work, since `io.read` is not a variable (only `io` is, and `read` is a member of it)
- Global variables do not need declarations
- We can change this behaviour if we like
- Since the global environment is a regular table, we can use metatables to change it's behavior
- In a metamethod, `debug.getinfo(2, "S")` returns a table whose field `what` tells whether the function that called it is a main chunk, a regular Lua function or a C function
- In Lua, global variables do not need to be truly global (eg. the could be *nonlocal*)

- A *free name* is a name that is not bound to an explicit declaration, that is, it does not occur inside the scope of a local variable with that name
- The Lua compiler translates any free name `var` to `_ENV.var`
- This new `_ENV` variable is **not** a global variable
- Lua compiles any chunk in the presence of a predefined upvalue called `_ENV`, which is initialized by default with the global environment
- That is, Lua only has local variables, and any global variables are translated to local ones (`var` to `_ENV.var`, where `_ENV` is a local variable or upvalue)
- `_ENV` is, like `_G`, a plain regular (table) variable
- `_ENV` will refer to whatever `_ENV` variable is visible at that point in the code
- The assignment `_ENV=nil` will invalidate any direct access to global variables in the rest of the chunk
- The main use of `_ENV` is to change the environment used by a piece of code
- Again, we can change the behavior of `_ENV` with metatables
- The `load()` function has an optional fourth parameter that gives the value for `_ENV`
- Using that, we can limit or change the environment for external code
- We can also use `debug.setupvalue()` to change the upvalue for a compiled function
- When the function is the result of `load()` or `loadfile()`, Lua ensures that it has only one upvalue and that this upvalue is `_ENV`

Chapter 15: Modules and Packages

- A *module* is some code (Lua or C) that can be loaded through `require()` and that creates and returns a table
- Everything that a module exports is defined inside this table, which works as a *namespace*
- Thus, we can manipulate modules like any other table
- Once a module is loaded, it will be reused by whatever part of the program requires it again
- The first step of the call `require "modname"` is to check in `package.loaded` whether the module is already loaded
- If not, it searches for a Lua file with the module name, which would be loaded with `loadfile()`
- If that doesn't work either, it looks for a C library with the module name
- If it finds a C library, it looks for the function `luaopen_`*modname*`()`
- Once it got a loader function, this is called with two arguments: the module name and the name of the file where it got the loader
- To find Lua files, Lua uses a couple of *templates* defined in `package.paths`
- The same applies the C libraries, but their paths are in `packages.cpaths`
- Writing a module is simple: create a table with all the functions that

should be exported in it and return it
- We can use some `_ENV` tricks to automatically build the table
- Lua also supports *submodules*, their names are formed with dots: `module.submodule`
- To find submodules, Lua translates the name of the submodule into a path, like this: `/usr/local/lua/`*module*`/`*submodule*`.lua`
- For C libraries, `require()` looks for a function called `luaopen_`*module_submodule*`()` to load the submodule

Chapter 16: Object-Oriented Programming

- A table in Lua is like an object in more than one sense
  - tables have a state
  - tables have an identity (a self) that is independent of their values
  - two objects (tables) with the same value are different objects
  - an object can have different values at different times
  - tables can, like objects, have their own operations
- To implement *methods* (functions that work on objects), a reference to the state of the object is needed
- Lua has special syntax for this:

```
Account = {balance=0} -- array object table

function Account:withdraw(amount)
    -- 'self' is the state of the object
    self.balance = self.balance - amount
end
```

- The new syntax uses the *colon operator* (`:`), which hides having to pass the state to methods away
- It adds an extra hidden parameter in a method definition and in a method call (eg. `Account:withdraw(20)` implicitly passes an argument `self` to the function)
- *Classes* in Lua can be implemented with tables and some metatables magic
- Because of the way metatables work in Lua, a `new()` method can create instances of a class and provide an interface to make subclasses:

```
function Account:new(o)
    o = o or {}
```

```
    setmetatable(o, self)   -- setting metatable
    self.__index = self     -- allows subclassing
    return o
end
```

- With this kind of setup, any methods and default values will come from Account (due to the metatable), anything else will be stored in the table
- We can now override class methods in an object
- Also, like this any object acts as a class itself (you can call the `new()` method on any object of this class), this is how subclassing works:

```
-- standard account object
acc = Account:new{balance = 5}

-- subclass Account
SpecialAccount = Account:new()

-- special account object
sacc = SpecialAccount:new{balance = 10}
```

- Multiple inheritance needs some extra work to set up the metatable correctly
- There are ways of restricting access to object data in Lua, but they are not used often