io.write("Anyone Can Learn Programming")

LEARN CODING WITH LUA

2023

A Slow and Gentle Introduction to Basic Programming for Non-Programmers with Step by Step Instructions

-- Harry Yoon PhD

Learn Coding with Lua

A Slow and Gentle Introduction to Basic Programming for Non-Programmers (REVIEW COPY)

Harry Yoon

Version 1.0.0, 2023-04-05

REVIEW COPY

This is review copy, not to be shared or distributed to others. Please forward any feedback or comments to the author.

- feedback@codingbookspress.com

The book is tentatively scheduled to be published on March 22nd, 2023. We hope that when the release day finally arrives, you would also consider writing an honest review on Amazon.

- Learn Coding with Lua: A Slow and Gentle Introduction to Basic Programming for Non-Programmers [https://www.amazon.com/dp/B0BF19Q3DV/]

Copyright

Learn Coding with Lua

Published: March 2023

Harry Yoon
San Diego, California

ISBN: 1111

Preface

Starting from *zero* is really hard. For anything. Have you tried to learn to play tennis? Play the piano? How about chess? This is especially true for programming. Computer programming is becoming more accessible to more people these days. But, it is still rather difficult for real beginners to get started, that is, without some help from more experienced programmers.

Learn Coding with Lua: A Slow and Gentle Introduction to Basic Programming is specifically written for the people who have no or little experience with programming, or for the people who may consider themselves as "non-programmers".

The goal is to give the *ordinary people* like you a gentle introduction to the practice of programming. For example, *What does a programmer do?* A lot of books, and other resources, that are written for the "laypeople" tend to include a lot of stories like how the computers were invented, what kinds of programming languages exist, and who created which languages, etc. They may be interesting, but they do not help much in way of helping you learn coding. This book, however, focuses on real programming, both theory and practice. After a couple of brief introductory lessons, we go straight to hands-on programming, and we cover a lot of ground in a relatively small space.

Although it is a short book, and it uses rather simple examples, you will get a taste of real programming by going through this book. Note, however, that it is not a textbook. This book is intended for quick reading, and doing some hands-on exercises if desired, and it is not meant to be "studied". Just remember, you do not have to understand everything to get the benefits of reading this book. It's never *all or none*.

LUA is one of the most popular languages among the people who are just starting to learn coding, including children and young people of various ages.

Lua is easy to learn and use, and yet it provides enough complexity and flexibility to be useful even for professional software development. As a matter of fact, Lua is one of the most versatile modern programming languages. As a beginner, when you are just starting out, the choice of a particular language is not that significant. You will likely end up trying out a few different programming languages before you eventually settle with a language or two that you really like. Regardless, you will be glad that you started with Lua.

After learning the basics of programming using Lua, and this book, you will be ready to learn more difficult topics in programming, now on a firmer foundation. And, above all, you will most likely be eager to do more coding. If so, then this book has done its job.

Throughout your journey of learning coding, the most important thing is *to have fun.* If you don't feel like you are having fun while learning programming, then stop. Take a break. Do something else for a little while and come back to this later, *if you feel like it*. Learning should be fun. Learning programming should be fun, although it may be hard. Always keep that in mind.

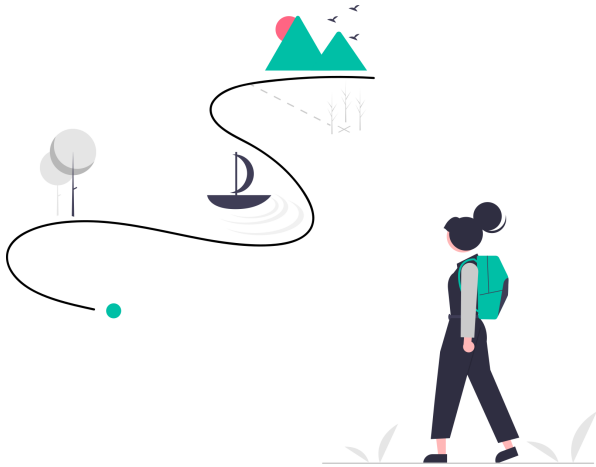
Have fun! Always!

Author's Note

It should be noted that the word "beginner" is not a well-defined term, particularly in the context of programming. This book is written for people with no or little experience, say, from a few weeks to a few months of "dabbling" with coding, or possibly a bit longer, but, at the end of the day, you will have to decide whether this book is suitable for you and whether it can help you learn coding. We suggest you browse the beginning of the book, especially the table of content, before you purchase a copy. If you have already purchased, then go through the book quickly before you commit. If you feel like this book is not for you, or if you feel like you will end up getting very little out of this book, then return the book, and try to find a different resource.

Table of Contents

Lesson 1. A Journey of a Thousand Miles ...



Why are you interested in learning programming? Do you want to build your own mobile apps? Do you want to create websites? Do you plan to pursue a career in the software industry? Is it out of curiosity? Or, you just picked up this book by accident? ☺

Regardless of what your reasons are, this book will give you a real taste of *basic* programming. It may, or may not, provide you with what exactly you are looking for. But, it goes without saying that the foundation is the most important thing. Whether you have some coding experience or not, learning, and practicing, coding is a lifelong process. Having a solid foundation can help you learn coding faster, and enjoy more, in the long run.

This book is written for complete beginners, from 10 year olds to 50 year olds, and to 100 year old grandpas and grandmas, who are interested in learning.

Depending on your background, and the level of prior exposure to programming, you may still find this book difficult. No worries. Learning is a skill, which you can also *learn*. You may have to take some winding road, or take a break once in a while, but eventually, you will get there. Remember, this book is written for *you*.

On the other hand, you may find this book too easy (that is, if such a thing as "too easy" exists). That's all right as well. There must be something you can learn from this book. (As suggested, you can quickly browse the book and return it for a full refund. ☺) Otherwise, you can always try some exercises, and see if you can *graduate* from this level "zero" to the next level up, where programming will be *more fun*.

One final remark. Although this book is written for beginners, we emphasize *both theory and practice* unlike the majority of beginner's books. In fact, some readers may find this book rather too theory-heavy. If so, you can skip much of the theory parts, especially if long explanation bores you. Theories will help you build a solid foundation, primarily for your future learning, and they may not be essential for following the instructions and doing the exercises in this book.

1.1. What is Programming?

If you ask ten different programmers this same question, *what does programming mean to you?*, then you will get ten different answers. It's like the proverbial elephant with blind men.

First of all, what is a computer? A computer is a device that does "computation". Modern computers can do so much, and such complex tasks, that it may not always be obvious, but ultimately computers do computation, as in `1 + 2` is `3`. A program accepts an input data (like `1` and `2`) and produces an output data (like `3`). This example does a trivial operation, namely, addition of two numbers. Nevertheless, this is the computation, or the logic of the program. A program generally consists of two components, logic/computation and data.

Programming is a process of creating a program, *using a computer programming language*, which performs the desired computation, and which accepts a (broad) range of input. That is, we do not generally write a program for *one* specific task, but for a range of related tasks, if you will. This is an important concept to understand, which is, if you think about it, sort of obvious. A calculator that only adds `1` and `2` may not be that useful. A calculator that can add arbitrary two numbers, or even a calculator that can add an arbitrary number of input numbers, will be generally more useful.

Despite the recent advances in machine learning and AI technologies, which now can create pretty advanced chatbots and what not, computers are *not* that smart. We still use special languages to *really* talk to them. When you talk to a baby, you tend to use special "baby talk" languages. It's just like that.

There are many different programming languages. You can use any of the widely used languages like Python, JavaScript, C, etc. to learn programming.

When you are starting out, as stated earlier, a particular choice of programming languages is not that important. You will generally have to focus on learning the fundamental programming concepts, rather than other language-specific details.

Lua is one of the simplest and easiest-to-learn programming languages. As a matter of fact, Lua is as good a choice as any for beginning programmers, if not better. The fact that Lua has a rather simple grammar, and a smaller set of standard libraries, can be a big help for beginning programmers.

Regardless, a lot of concepts you learn from this book will be also applicable when you learn, or program in, different languages, hopefully, in the near future. (All programming languages are different, in some aspects, and they are all fun.) Note that the title of this book is *Learn Coding*, and not *Learn Lua*.

1.2. Two Big Words - Syntax and Semantics

Although this book is for total beginners, we will start with two big words, *syntax* and *semantics*. Many of the readers may have heard of these terms, possibly from different contexts. Their exact definitions are not very important for our purpose. But, roughly, syntax means "form" and semantics means "meaning".

As a side note, programming languages are built based on how the natural languages like English work. A linguist, Noam Chomsky, created a formal theory of languages in the mid 20th century, and all programming languages are constructed on this foundation. An English sentence, for example, "I fly", has a subject *I*, which is a pronoun, and a verb *fly*. That is syntax. In addition, this sentence has a meaning, as in "I fly to New York", or something like that. That is semantics. Saying "Arts fly" may not have much meanings although this sentence is syntactically correct. Coding languages have essentially the same corresponding structure, with a similar grammar comprising syntax and semantics.

When you start learning programming, learning a programming language is an important part (and, these two are not the same). You cannot play Mozart without learning how to play musical instruments like the piano or the violin. Although, as indicated, the choice of Lua

as the programming language in this book is somewhat incidental, you will have to learn the essentials of the Lua language grammar, its syntax and semantics, to learn how to program.

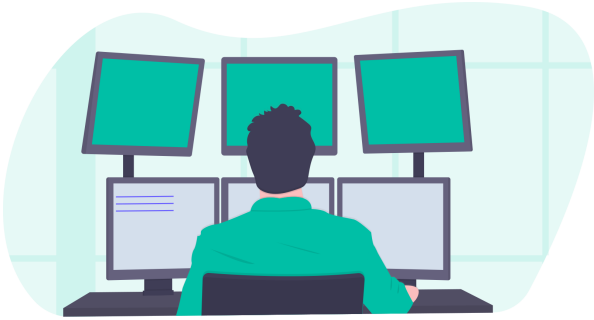
All code examples in this book are written in Lua. All coding concepts are explained in the context of Lua programming, using the Lua syntax. But, more importantly, just like the distinction between the form and the meaning, the readers are encouraged to focus more on the fundamental concepts of programming rather than (possibly language-specific) details.

One thing to note is that although anyone of any age who has a reasonably good command of English can learn basic programming using this book, it is not specifically written for children, and they may find some languages used in this book rather difficult to understand. (This book is more of `PG-13` than `G`. ☺)

All exercises are optional, and no solutions are included in this book. This is primarily to encourage the readers to try them out for themselves, without relying on provided answers. Just remember that there is no single "correct" answer for any given problem. Then, how do you know that you have done right? *Does your program do what you intend it to do?* That's the only question that matters. At least, for the time being.

Good luck!

Lesson 2. … Begins with a Single Step



As we suggested, a programming language is a tool. In fact, it is one of the (many) tools for programming. Other important tools are a code editor, in which you write programs, and the *code interpreter*. Lua belongs to a category of languages, along with Python and JavaScript, which do not generate executable programs (like EXE files on Windows). Lua code, or "script", is directly run by a special program called the interpreter.

Setting up a development environment is an important first step to be able to do programming. But, in this book, we will skip most of it, and instead we will use an "online IDE", which is essentially a combination of code editor and interpreter. In particular, we will use one of the most popular free services called *Replit*. If you would like to follow along while reading this book, we suggest you use the same, or similar, online service.

2.1. Online Code Editor

Let's go ahead and create a new account on Replit, if you have never used this service before. If you happen to already have a command-line Lua interpreter on your computer, or if you have other IDEs, like VSCode, that support Lua software development, then you can use that as well.

We are using Replit in this lesson, for concreteness, and for simple illustration, e.g., mainly for people who have no prior (Lua) programming experience.

2.1.1. Creating a Replit account

Visit their website,

- replit.com

And, press the big blue **Sign Up** button, and provide the necessary information to create a new account. You may have to verify your email, depending on how you have signed up.

- replit.com/signup

2.2. Hello World!

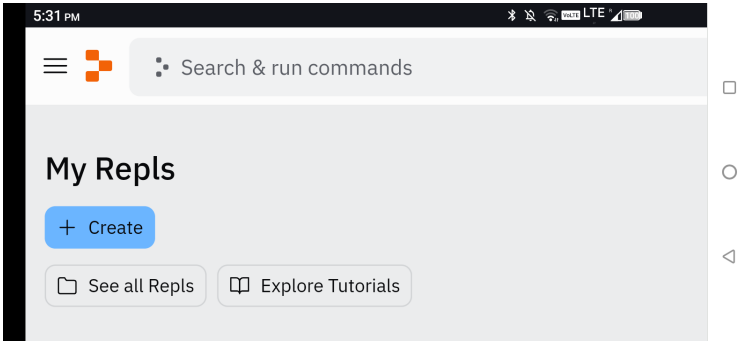
Now, let's try our new development environment, Replit, using a simple Lua program.

Customarily, we almost always use the so-called "Hello World" program, a program that simply prints out a text *Hello World*, for initial test purposes, for no other reason than it is sort of a "tradition", which started over 40 years ago!

When a new baby is born, if the baby were able to speak, he or she would probably say this greeting, *HELLO, WORLD!* as their first words, to the world. So, it is pretty natural to use "Hello World" as our first program. ☺

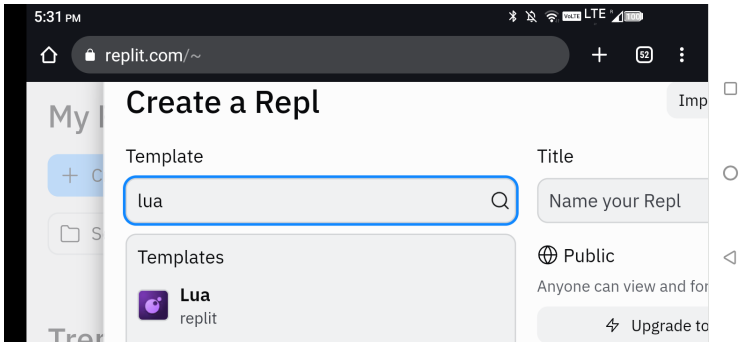
2.2.1. Creating a Hello World program

Go to the Replit website, replit.com. Click on the big (blueish) + **Create** button.

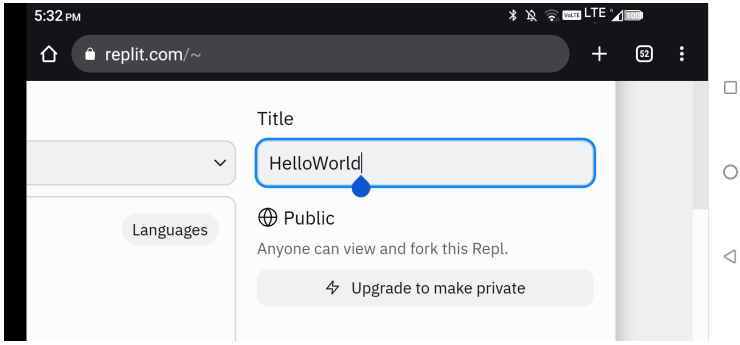


The screenshots are taken from Replit's mobile website. But, you will most likely want to use desktop/laptop computers for coding.

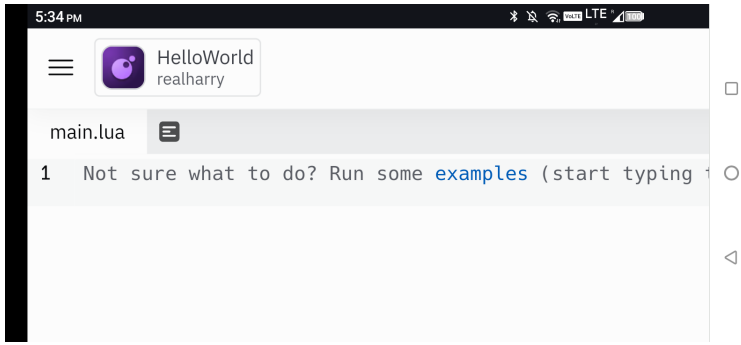
Type "lua" in the template search box, and select the purplish "Lua" template.



Enter any meaningful project name, such as *HelloWorld* or *MyFirstProgram*, etc., in the **Title** box, and then press the + **Create Repl** button.



Replit uses the word *Repl* to refer to a workspace, which includes a set of Lua source files, among other things. In this newly created Repl, you will see an editor with the file name set to *main.lua*. Lua files usually end with the *.lua* extension. The name *main.lua* is just a convention, which is given to the "main" code file in a program.



Type the following code in the editor box.

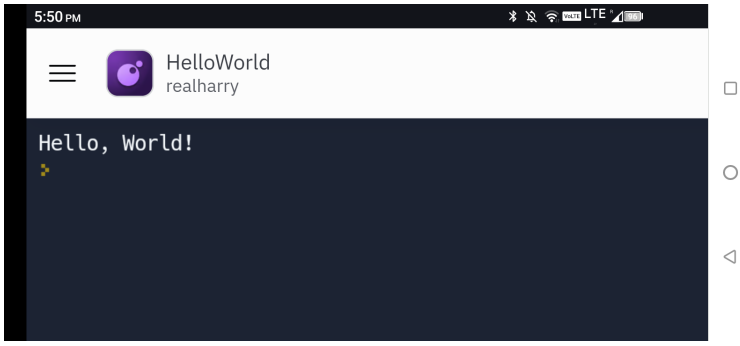
```
print("Hello, World!")
```

Now, your Repl screen might look like this:

That's it. We just created our first Lua program. We will explain what this code means in the next lesson, but this is indeed a hello-world program since, as we will see next, it prints the text "Hello, World!".

2.2.2. Running a Lua program

Replit has this greenish "play" (or, run) button at the bottom. Press the button. Replit will then execute your code. The result is shown in the "Console" tab. Do you also see the text, *Hello, World!*?



Congratulations! You just ran your first Lua program.

We will only use the most basic features of Replit in this book, namely, code editing and running, or "interpreting", Lua code. Do not worry about other things, for now.

2.3. Comments

Programs often contain text that is not directly used by the computer. They are called the "comments". They are primarily used as notes, to yourself and/or to other fellow programmers.

In Lua, we use a sequence of two dashes, `--`, to start a comment. The comment continues until the end of the given line. (There are other ways to write comments in Lua, but we will not discuss them in this book. As a general rule, we will try to be as narrow and as specific when it comes to Lua syntax. On the other hand, when we discuss the "theories", we will try to explain them more broadly so that the knowledge can be useful even when you decide to learn and use a different programming language.) For example,

```
-- I am a comment.                          ①
-- Lua ignores me.
print("Hello, World!")         -- Huh?     ②
```

① These two lines are comments.

② Comments can be placed after a valid program statement. In this example, `-- Huh?` is a comment. As we will discuss in the next chapter, white spaces are *mostly* ignored by Lua. Hence, this program really consists of one statement `print("Hello, World!")`, as before. All others are insignificant as far as the Lua interpreter (e.g., Replit) is concerned.

One thing to note is that the computers are not the only audience of your programs. As stated, people also may, possibly, read your programs. As you practice more programming down the line, you will learn that it is

rather important to be able to write cleaner and more readable code, in terms of code formatting and what not, and not just the code that "works". This topic is, however, beyond the scope of this book.

Exercises

We learned, in this pre-coding lesson, if you will, how to print, or more accurately stated, how to tell the computer to print, a text "Hello, World!". Although we haven't explained what this program means, other than how it works, let's try and do our first exercise. If you are an absolute beginner, and if these exercises do not make sense, then you can skip them, and maybe come back later. See the author's note below.

Our single line program contains a couple of dozen characters, and it may look gibberish to you. But, you may still be able to see some structures in this program. Now, our first question is, what would you do if you wanted to print *!Hola, Mundo!* instead of *Hello, World!*?

Ex 1. !Hola, Mundo!

Write a Lua program that prints a text *!Hola, Mundo!*. Do this on Replit, by creating a new Repl with a name "HolaMundo", using the Lua template. Try running your program, e.g., using the green "play" button. Does it print the desired text to the console?

Author's Note

"Backtracking" - Or, How to Use This Book

Last night, I noticed that a *huge* flying insect somehow got into the bathroom, and it was trying to get out through the bathroom window. Obviously, it was not possible. I opened the window a bit wider, thinking that this wasp-like insect can find its way out more easily, and I completely forgot about it.

This morning, I noticed this guy or gal still on the same side of the closed window, trying to fly out by desperately flapping its wings. I could see that it was rather exhausted after having tried for so long, likely for all night. (I almost saw bulging muscles in the bases of its wings, just like those you see in the arms of body-builders. ☺)

I had to intervene. I moved this guy a little bit further away from the window, just an inch or two. And, that was all it needed. Once it was away from the closed window, it was immediately able to find a better way out, that is, through the open side of the window.

In computer science, we often deal with this ubiquitous, and rather broad, category of problems called search. They are everywhere, and we are all familiar with one particular use case, namely, Web search, for instance. Finding a way out from a maze is another classic example of search.

This wasp-like creature was trying to find a way out. He was *searching*, but he was just too close to the window. The outside world seemed so close to it, and yet it was not able to reach it. In a sense, moving away from the window, away from the destination, the outside world, might seem somewhat counter-intuitive, but that was what ultimately led to his successful escape.

This is somewhat tantamount to a technique called the "backtracking". Once you get stuck along one search path, you will need to *backtrack* through the currently taken path to eventually, and hopefully, find a different, and better, path(s) forward. Unfortunately, this wasp look-alike did not know that, at least initially.

So, why are we telling you this long-winding, and seemingly irrelevant, story? We have been writing a few programming books for the last couple of years. Unfortunately, *some* readers do not seem to know how to really learn, e.g., using books like

this. They tend to expect too much, and they end up blaming the books when they fail to meet their unrealistic expectations.

First of all, books are just a tool. It is you who use the books. Second, no books are perfect. They are bound to have some (hopefully minor) errors and typos and what not. Third, books like this one are written for broad audience. Not just for people with different experiences, but even with different thought processes and personalities. And hence, some readers may find this particular book, for instance, too easy, or some may find it too difficult. It is rather important to accept this premise that all people are different and you are possibly different from everyone else. In fact, no readers will find this book perfectly tailored to them. Learning, using an imperfect resource (*imperfect* to you) due to various reasons, is a skill, which you can get better at with some conscious effort, as we emphasized in the beginning.

While going through this book, if, that is, *if*, you find yourself frustrated because the book seems too difficult for you to understand, or because the book *seems* full of errors and typos which makes it rather hard to learn anything in your mind, or for whatever reason, then here's a quick advice for you, if you are inclined.

First, take five. Breathe. Don't take it personally. It's not your fault. It's not the book's fault. And, just remember, you are not alone. This is called "learning". Everyone goes through this, more or less. Learning is hard, *by definition*. (Otherwise, everyone will know everything. ☺) There is no magic pill that will teach you programming, or anything for that matter, just by swallowing it. Learning is a trial-and-error process, which you have to keep trying, that is, if you are willing.

Now, you do not have to understand every word or every sentence in any given lesson. If you get the gist of the lesson, then you can move on to the next lesson. There is no point of fixating on the things that do not make sense to you, blaming yourself or others, or dwelling on the half-empty glass (or, the "bad book"), if that's how you view the world. Some explanations in the book may, and will likely, make more sense as you gain more experience, through learning and practice.

In fact, this is where the theme of this story, "backtracking", comes in. Finally. ☺ Although it is hard to quantify the mental process, let's suppose hypothetically that you have understood 80% of Lesson 5. Now, if you move on to the next chapter, Lesson 6, you will likely end up learning even less, for example, because some concepts may depend

on the previous lesson. That is perfectly all right. Let's suppose that you have understood 70% of Lesson 6, and you move forward to Lesson 7. Now you get 50% of what this lesson is teaching. And, you keep going, until you reach the point where you feel like you are getting less out of reading than the effort you are putting in.

Then, it is time to BACKTRACK! You do not have to go back to Lesson 1. Go back to the earliest lesson where you felt like you understood *most* of it. In this arbitrary example, that could be Lesson 5, Lesson 6, or even Lesson 4 or earlier. This criterion is really subjective, and it is really up to you. Anyways, go through the book again starting from this lesson. *What's the point of this?*

Let's suppose that you start your second reading from Lesson 5. Now, you will realize that this lesson is a bit easier to follow. This is clearly because you have understood 80% of its content already. But, more importantly, you will most likely end up understanding more of Lesson 5 this second time around. How is that possible? That is because, from the time you first went through Lesson 5, *you have learned more,* e.g., by going through the later lessons, even though you may have felt like you understood and remembered very little of these lessons.

Now, suppose that you have understood 85% of Lesson 5. You keep moving forward, 75% of Lesson 6, 60% of Lesson 7, etc. At some point, again you may reach the point of negative return, in terms of your time and effort invested vs. the benefits you are getting. Then, again you do backtracking, say to Lesson 6 or Lesson 7, or any later lesson. You repeat this metaphorical *two-step* forward and *one-step* backtracking process until you finish the book, or until you feel like you have learned enough from this book.

Here's one example of how you might end up reading this book. It is important to note that it is what you might end up doing, in hindsight, and it is not a plan or recommendation.

```
L1 -> L2 -> L3 -> L4
   -> L2 -> L3 -> L4 -> L5
         -> L4 -> L5 -> L6 -> L7
               -> L5 -> L6 -> L7
```

This is a general advice, not necessarily specific to learning programming using this book. You will obviously have to take it with a grain of salt.

Lesson 3. Anatomy of Hello World

Although this is not a biology class, we will look at the "anatomy" of the hello world program in this chapter. A more appropriate term would have been "analysis". This is one of the commonly-used techniques in learning, and in any other studies.

When you try to understand a system that consists of multiple parts, it is often helpful to do what is known as *analysis and synthesis*. That is, roughly speaking, you break the system into smaller components, try and see if you can understand each component, and then put them together again to understand the whole system.

As we have emphasized a few times already, learning is a skill, an important skill, which you can learn and get better at. You are currently teaching yourself how to program. This book is just a tool, a helper, and it is ultimately you who have to teach programming to yourself.

It will be beneficial to you, in the long run, if you consciously think about how to learn better, rather than just focusing on the particular subject matter at hand, and details (that you may end up easily forgetting anyways).

3.1. Hello World, Again!

Here's our hello world program from the previous lesson, which consists of one line.

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

First of all, if you have never seen any program before, it is just a text. The code includes characters, that is, letters and punctuation symbols. And, it can also include numbers and what not, as we will see throughout this book. But, ultimately, a program is just a sequence of characters.

If you haven't seen a program like this, however, you

cannot easily tell what these characters mean or what they indicate. Without explaining exactly why or how, this one line program consists of the following four components:

- `print`,
- `(`,
- `"Hello, World!"`, and
- `)`.

This is the lowest-level "anatomy" of our hello world program. Can you see it? More or less? This is called the "lexical analysis" in programming, using another big, *technical,* word, but the terminology is not that important, at this point. These components are comparable to atoms and molecules, if you will. As you know, all things in this world are made of atoms and molecules. Likewise, all programs are made of these small components.

One thing to note is that white spaces like space, tab, or newline characters are not part of these components, and they can be (in general) freely mixed with these components without changing the meaning of the program. For example,

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

This is *exactly* the same program as our original hello world program. They do *exactly* the same thing.

To state differently, the white spaces are ignored between these small components. (These components or atoms are often called "tokens", or lexical tokens, in programming.) Note, however, that that is not the case *inside a component*. For instance, `p r int` is not the same as `print`. In fact, white spaces, along with some punctuation symbols, are generally used as "separators" between these components. Hence, `p r int` would have been three separate components, `p`, `r`, and `int` (although they may not have been valid tokens of Lua, as we will discuss in the next section on syntax).

The third component, `"Hello, World!"`, is somewhat special. Although it contains a space in the middle as well as two double quotes `""` in the beginning and end, it is still considered one component. This is generally called a *string* in programming. The opening and closing double quotes, as a pair, indicate that this is just one component, a string. Spaces in strings are significant. `"Hello, World ! "`, for instance, would have been a different string.

Now, can you see how our hello world program consists of four components? Scanning from left, we see five letters `p`, `r`, `i`, `n`, and `t`, and they form a single component since there are no white spaces between them. The next character `(` acts like a white space or

separator, and it also forms a component by itself. Next, we see a character `"`, which is a start symbol of a string. Hence, we proceed until we see another (closing) `"`, and all these characters form a single string component. After that, the last character `)` is another separate component.

If you do not fully understand this, that is perfectly all right. You will be able to recognize these almost *subconsciously* in no time once you start programming for a little while. One important takeaway from this section is that different classes of characters like letters, numbers, and punctuation symbols, as well as white spaces, play different roles in the program source code. Details are not as important as you might think, at least in the beginning.

3.2. How to Create a Sentence

So, what do these components mean? As we will see, it's not just the meaning of each of these components, but more importantly it's their arrangement that is important in programming. This is where the aforementioned *syntax* comes in.

Just like words have to be arranged in a certain way to form a valid sentence in English, these components have to be arranged in a certain way to be a valid "statement" in programming.

Going back to our hello world program,

```
print ( "Hello, World!" )  ;
```

This is a statement. (In Lua, a statement can *optionally* end with a semicolon, `;`, but semicolons are rarely used in practice.) In general, a statement does something. It is an instruction to the computer to do what the statement indicates (e.g., its "semantics").

Again, without explaining exactly why or how, this statement is a "function call". The first word `print` is the name of a (built-in) function, and the pair of parentheses, `()`, is used to *call* this function. The string in the middle, `"Hello, World!"`, is called the "argument" (of this function call).

We will discuss functions in more detail later in the book, but a function does something when it is *called*. What's important at this point is its syntax, or "form": A function name, followed by a pair of parentheses, and the argument(s) inside the pair of parentheses.

A programming language like Lua includes many syntactic rules, and you will have to learn most, if not all, of them to be proficient in programming with that language.

> But you do not have to memorize them like you memorize the multiplication table. With some practice, you will get gradually more *familiar* with them over time, and eventually they will become natural to you.

3.3. What's the Meaning of All These?

So, what does this statement, `print("Hello, World!")`, do? Well, we already saw what it did. It printed the string argument `"Hello, World!"` to the console, *without the double quotes*.

That is more or less the *semantics* of this statement.

This functionality is implemented in the function `print`, which is a *builtin* function in that it is built into any Lua interpreter. In other words, when you use Lua, you can always rely on this function being available for your use. (One cannot say the same for other library, or user-defined, functions, in general.)

Just as an illustration, if you happen to be using the command line Lua interpreter, e.g., instead of Replit, then you can execute our hello world program as follows. (You can skip this if you do not know what this means at this point.)

```
$ lua                                                ①
Lua 5.4.4  Copyright (C) 1994-2022 Lua.org,
PUC-Rio
> print("Hello, World!")                            ②
Hello, World!                                        ③
>                                                    ④
```

① The dollar sign *$* represents the shell prompt. We start the Lua interpreter using the *lua* command in this example.

② The greater-than sign > represents the Lua prompt. We run our favorite print hello world statement by typing it in one line and pressing the Enter key.

③ This is the output.

④ Another Lua prompt, waiting for the next command.

3.4. Do This? *And Then* Do That?

In programming languages like Lua, in fact, in the vast majority of languages, a program is essentially a sequence of statements. As indicated, *statement* is a general term that we commonly use to refer to a command or an instruction to a computer, as in "do this". (Well, *please* is optional. ☺). Lua includes a number of different kinds of statements, which are all essential to be able to write any non-trivial Lua program. We will look at a few of them in this book.

But, the more important thing to remember is that, as a general rule, a program is executed (more or less) sequentially, from top to bottom, *one statement at a time*. This is known as "sequential processing". Languages like Lua are often called *imperative* programming language.

The only statement that we have learned so far is the `print` statement, which is also, and more precisely, a *function call expression*, as we mentioned earlier. Let's try writing more than one statement using `print`.

```
print("It's like the ticking crocodile, isn't
it?")  ①
print("Time is chasing after all of us.")
```

① Due to the fixed paper size (for the paperback version) and device size (for the ebook version), the text may wrap around. But, this is one line. If you type it on a computer, you will need to put it in one line. Unfortunately, this comment applies throughout this book. This is just a limitation of the medium that we use. In practice, many people use big screen monitors for programming.

This is one program, comprising two statements. Although it's not absolutely necessary, we tend to write statements in separate lines, e.g., one statement per line. What do you think this program will do?

When we give this to Lua, it first executes (or, "interprets") the first statement, which will print the text *It's like ...* to the console. It *then* executes the second statement, which will print the text *Time is ....* Here's a sample output, if you run it on Replit, for instance,

```
It's like the ticking crocodile, isn't it?
Time is chasing after all of us.
```

Now, what do you think we will have to do to execute 3 statements or 4 statements, or more, one after another? Here's a sample program that includes 5 statements:

```
print("I'm the first.")
print("I'm the second..")
print("I'm the third...")
print("I'm the fourth....")
print("I'm the fifth.....")
```

When we run this program, it will print out the following output. You can try it yourself, if you'd like.

```
I'm the first.
I'm the second..
I'm the third...
I'm the fourth....
I'm the fifth.....
```

3.5. The Great Escape

We have seen so far, although we haven't explicitly stated, that *strings* can contain sequences of various characters. For example, `"Never is an awfully long time."` is also a string since, why?, since the characters are enclosed in a pair of double quotes `""`. So is `"blah blah abc xyz 123 !@#$%^&* I'm gibberish (blah) 42 yak yak"`.

One *small detail* we didn't mention is that the opening `"` and closing `"` double quotes of a string must be in the same line, e.g., in a program source code. (Remember, that's just "syntax", or rules. You cannot argue with the rules. "Officer, why is the speed limit of this road only 25 miles per hour? I can drive much faster here." ☺)

Most characters can be included in a string. But, there are some exceptions. For example, the newline characters (or, linebreaks, formfeeds, etc.) cannot be directly included in a string because, for example, that will end up putting the two quotes of a string on two different lines. (Again, "why" is not important, at least for now. At the end of the day, it's just a rule. Whether the rule is reasonable, or it's necessary, is a different question. You'll eventually have to learn, and *follow*, all, or most of, these rules, aka the *syntax* (of a particular programming language).)

Then, how do you include newlines in a string? You can "escape" the newline characters. It is a special syntax. Lua supports a few different "escape characters", and all of them start with the backslash character, `\`. For example, a newline is represented as `\n`. Another example is a tab, which is represented as `\t`. Note that these escape characters represent single characters in a program although they are written with two characters (including the backslash prefix) in a source code. (They are sometimes called the "escape sequences" since they lexically involve more than one character.)

Incidentally, this escape syntax was originated from C. In fact, programming languages all have very similar syntax although some of them look very different. It's like humans and pigs share 98% of the same genes. (We think that) we are, and look, so different from pigs, and yet we are really cousins. There are dozens of different programming languages that are currently in wide use, and at the end of the day, the differences between them, at least syntactically, are not that great. ("So, why do we have so many languages?" is for another day. ☺)

Getting back to the newline escape character, what do you think the following program will do?

```
print("If you cannot teach me to fly,\nteach
me to sing.")
```

Yes. It outputs something like this to the console:

```
If you cannot teach me to fly,          ①
teach me to sing.
```

① The `\n` character in a string in a source code is replaced with the real newline in the output.

An astute reader might have realized that this could also have been done with two statements. No?

```
print("If you cannot teach me to fly,")
print("teach me to sing.")
```

This program will print out exactly the same output. Although these two programs are different (e.g., they use different syntax, etc.), they do *the same thing*.

Now, here's an important lesson. You can achieve the same goal in life using different means. Oh, we are talking about programming, so, formally, there is a many-to-one correspondence from syntax to semantics. OK, this statement is just a total gibberish, but to put it differently, you can write programs in many different ways to do the same task, more or less.

3.6. String Additions

What? You can *add* strings? Yes, you can do anything in programming. (Or, maybe not.) You can add two strings to create a new string, e.g., using an "operator" `..` (two dots) in Lua. They are more formally called the "string concatenation". For example,

```
print("Absence makes the heart grow fonder...
" .. "or forgetful.")
```

This will print the following text in one line.

```
Absence makes the heart grow fonder... or
forgetful.
```

The two strings `"Absence ..."` and `"or forgetful."` are added, as indicated by the string concatenation operator `..` between them. This program is the same as the following, using one string:

```
print("Absence makes the heart grow fonder...
or forgetful.")
```

Again, syntactically these two programs look (slightly) different, but they are semantically equivalent. They do the same thing.

So, why do we have so many different ways to do the same things? So, how do we choose one method over others? ... These are not easy questions to answer, especially, in the beginner's book. But, let us just say that as you learn more you will know when to use certain things and when not to use certain things. You will also develop your own preference, over time. Sometimes, doing certain things in one way might be "better" than others, in some criteria.

Note that, for example, concatenated strings can be put into multiple lines unlike in the case of using only one string.

```
print(
"To live will be an awfully big adventure.\n"
..
"To die will be an awfully big adventure." )
```

If you will remember, from the beginning of this lesson, spaces are generally ignored between the lexical tokens in Lua. Newlines and tabs are also syntactically considered white spaces in many programming languages. For instance,

```
print(
  "Wendy, " ..
       "one girl is more use than "
  ..
  "twenty boys."
)
```

Although this program looks "ugly" ☺, it will print out the following text to the console:

```
Wendy, one girl is more use than twenty boys.
```

Exercises

As indicated, all exercises are optional, and in fact you may want to skip all, or some, of them in your first reading of this book.

Ex 1. Lexical analysis

The program source code "components" that we mentioned earlier in this lesson, which are usually called the *lexical tokens*, form the lowest level constituents in programming. That is, programs are made up of tokens, and not of individual characters. The syntactic grammar is defined with tokens.

Here's a simple Lua "module" that defines an `add` function. You do not have to, and you are not expected to, understand this code. (You don't have to know what a module is.) The question is, how many tokens are there in this source code?

```
local funs = {}

function funs.add(a, b)
  return a + b
end

return funs
```

The same question with the following program, which computes the greatest common divisor (GCD) of two numbers, `30` and `12`.

```
function gcd(a, b)
  while b > 0 do
    a, b = b, a % b
  end
  return a
end

local a, b = 30, 12
local v = gcd(a, b)
print("gcd =", v)
```

Ex 2. Syntactic analysis

Here's a simple Lua program with two statements:

```
print("Hello, Peter!")
print("Hello, Wendy!")
```

As is generally the convention, we put these two statements in two lines. Can you think of a way to make it a one-line program? Write a program that does the same thing as this but occupies only one line. (There can be more than one way to accomplish this.)

Ex 3. Semantic analysis

Do a Web search, and find five different ways to say "Hello, world!", say, in five different languages. Write a program that prints out these five different greetings in five lines.

Ex 4. Lua programs

We've learned a few important programming concepts. First, a program source code can be analyzed at three different levels, *lexical*, *syntactic*, and *semantic*.

Furthermore, a program in languages like Lua consists of one or more statements. The statement is what makes the computer do things. If you do not fully understand

these concepts, no worries. Again, what's important is the gist, and not every detail. We will go through these important concepts, again and again, throughout this book. Statements in a program are, in general, executed sequentially, e.g., from top to bottom. Now, write a program that prints the following output.

```
Dreams do come true,
  if only we wish hard enough.
You can have anything in life,
  if u will sacrifice everything else for it.
```

Ex 5. Escape characters

Write a Lua program that prints the following output, without using space characters. ☺

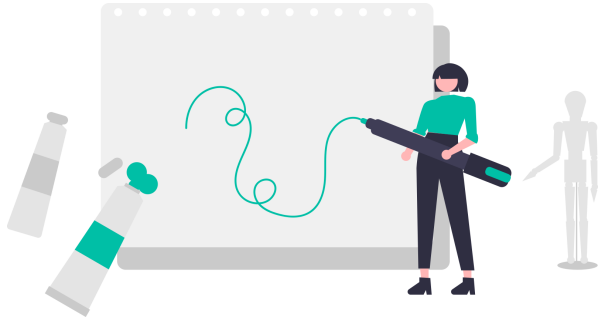
```
Apple   Pear    Orange
Lemon   Mango   Strawberry
Kiwi    Tomato  Melon
```

Ex 6. Double quotes in a string?

This is a difficult question since we haven't discussed this yet. Write a program that prints the following:

```
Huh? Is "Peter Pan" a true story?
```

Lesson 4. Is Programming Art?



Computers, or more precisely computer programs, can do so much these days. One of the most popular uses of the modern AIs has been generating images. These AI models can create pretty good looking, and sometimes rather realistic, images. There are some controversies surrounding the method and data they use to train these generative models, but it is still amazing to see how much a computer software can do.

Before the age of computer graphics and graphical user interfaces, we only used character-based terminals. There is one technique, or more like a tradition, that persists till today, called the ASCII art. The word *ASCII* in this context simply refers to the fact that we only use characters for "drawing". For example, here's an *ant*. Does it look like an ant to you? ☺

```
\(")/
 (|)
/(_)\                                    ①
```

① All artworks used in this book are in the public domain, in our understanding. Otherwise, their copyright belongs to the respective owners.

Let's continue this tradition by practicing some ASCII art in this chapter. But, first, a little digression.

4.1. Values and Some Such

Let's try running (or, interpreting) the following code on Replit:

```
print (1 + 2)
```

It prints out *3*. This is because `1 + 2` evaluates, or computes, to the number `3`. These components like `1`, `2`, and `3` are called *values* in programming. Sometimes, terms like "objects" are also used (e.g., in possibly slightly different contexts), but we will stick to the word *value* in this book.

We have already seen some other values in this book. For example, `"Hello, World!"` is also a value. Interestingly, this does not look anything like `3` or `300`,

for instance. But, it is still a value (in programming). The difference between strings like `"Hello, World!"` and numbers like `300` is that they have different "types". In Lua, `"Hello, World!"` has a type, `string`, whereas numbers like `1`, `2`, and `300` have a type, `number` (naturally).

It is not really important to know what exactly `type` is at this point. You can think of it as something like "category", "class", "group", or "set", etc. Values like `5` or `500` belong to one category, or one `type`, and other values like `"Hello, World!"`, `"will be an awfully big adventure"`, or `"X"` belong to another category or `type`.

Let's try the following code:

```
print(type("Value is What You Get"))
```

This will print *string*. On the other hand,

```
print ( type ( 420000 ) )
--  5    0    5    0    5                    ①
```

① This is a comment.

This will print *number*. What are we doing here? Let's try dissecting, or analyzing, this last statement, for practice. First, this statement consists of 7 components, or tokens. Can you see that? This is what we called the "lexical analysis" earlier, and most programmers do this without even realizing it, e.g., almost subconsciously.

Syntactically, as we explained in the previous lesson, a pair of *matching* parentheses is used to call a function in Lua, e.g., the function, or name, that precedes the pair. Remember? Hence, as before, `print` is the name of a function (since it precedes the pair of parentheses at positions 7 and 25).

Parentheses can be used for other purposes as well in Lua, and in programming in general. But, we will try not to overly complicate things by trying to be too accurate in our explanations in this book. The readers might ask "then, how do you know if these parentheses are used for function calling, and not for something else?", for instance. That is syntax. We already mentioned that syntax is about structure, as in an English sentence, and not just individual tokens or symbols. This is something you will have to learn to recognize over time.

Using the same logic, we can tell that the token `type` refers to a function, e.g., because it precedes the pair of parentheses at positions 14 and 23. This is purely based on syntax. The fact that you may have never seen this particular function before has no relevance. Syntactically, `type` has to be a function name. Otherwise, it is an incorrect program statement. (You will see, and most likely have seen, many "syntax errors" while programming.)

In fact, `type` happens to be another builtin function in Lua. It returns the `type` of the given argument. (And, hence the name.) In this particular example, its argument is `42000`. The argument of `print` is `type(42000)`, whose value is `number`. Hence, the whole statement ends up printing *number* to the console.

Going back to our original example, `1 + 2`, in the beginning, this is one example of what we call an "expression". An *expression* is something that evaluates to a value in programming. Since `1 + 2` evaluates to a value `3`, it is an expression. So is `1 + (2 * 3)` since it evaluates to a value `7`. (The `*` operator is used for multiplication in Lua.) Values are expressions since they (trivially) evaluate to the same values.

Strings are values, as just stated, and hence they are also expressions. String concatenations are also expressions since they result in strings, which are values (and, expressions).

As we stated in the very beginning, computers do "computation". Therefore, as one can easily imagine, values, and hence expressions, play a fundamental role in programming. Roughly speaking, the statements, the building blocks of Lua programs, consist of values and expressions, and possibly other (nested) statements.

4.2. What's in a Name?

Now, although we primarily deal with values and expressions in programming, sometimes it is not always convenient to deal with them directly. For example, let's consider the following Lua code.

```
print("Oh, the cleverness of me!")
print("Oh, the cleverness of me!")
print("Oh, the cleverness of me!")
```

We are printing the same string value, `"Oh, the cleverness of me!"`, three times in this program. As we will explain a bit later, there is a "better", or at least different, way to do this. But, regardless, this program looks a bit odd, for example, because we are repeatedly typing the same value again and again, and again.

Just like you and I have names, values can have names. We can give this particular string value a name, and use that name instead of the value itself. For instance,

```
local message = "Oh, the cleverness of me!"
print(message)
print(message)
print(message)
```

In this program, we give the string a name, `message`, and we use that name in the following three `print` statements. In programming, the names like `message` are generally called "variables". In Lua, we declare a new name using the special keyword `local`. Note the syntax. It starts with the keyword `local`, followed by a name, an equality sign `=`, and finally the target value.

A variable declaration statement like this is generally (and, often erroneously) called *assignment*, and the equality sign `=` is called the assignment operator. This implies that we are "assigning" the value on the right hand side of `=` to the variable on the left hand side. In Lua, however, this interpretation, and the terminology, is not entirely accurate. The variables in Lua are just names. In this example, the name `message` refers to the value, `"Oh, the cleverness of me!"`.

Functions are also values, and they also have names. We used the names `print` and `type` earlier, which refer to their respective function values. Note that we do not know what exactly they are, but that does not matter. As long as we know their names, we can use them.

Names have a lot more uses in programming. Just like it is unimaginable for us not to have names, it is (almost) unimaginable to program without using names.

Here are a few more examples.

```
local part1 = "To be, "
local part2 = "Or not to be, "
local part3 = "That is the question!"
local question = part1 .. part2 .. part3

print(question)                          ①
```

① What would be the output of this statement?

Names have to follow certain lexical rules, but we will not cover them here. In general, a name has to start with a letter, and it can include letters and numbers.

We do not cover all syntax in this book. For example, there is a different way to declare a variable in Lua. We do not include it in this book. This comment generally applies throughout this book. For example, we may explain one way to do "iteration", but not all different ways to do it. We do not cover all the syntax of Lua. That is not the purpose of this book.

4.3. ASCII Art

Now, with all theories aside, let's go back to our ASCII art. We already know at least two different ways to print the ASCII art. Let's use our small ant as our model:

```
\(")/
 (|)
/(_)\
```

4.3.1. Print, print, and print.

This particular drawing occupies three lines, and hence we can print it in three separate print statements.

```
print("\\(\")/")
print(" (|)")
print("/(_)\\")
```

A few things to note here. First, the backslash characters are used for escaping special characters, and hence they cannot be used directly in a string. The backlash character itself has to be escaped, that is, using another backslash, e.g., `"\\"`.

Likewise, since the double quotes are used as the start and end symbols of a string, the double quote character needs to be escaped. E.g., `"\""`. (Although we didn't

mention, and we don't use them in this book, Lua strings can also use a single quote pair instead of a double quote pair. In strings using single quotes, the double quote character need not be escaped, and vice versa.)

We can alos use variables.

```
local antHead = "\\(\")/"
local antBody = " (|)"
local antTail = "/(_)\\"

print(antHead)
print(antBody)
print(antTail)
```

In this particular example, using variables seems more cumbersome.

4.3.2. String concatenation

This can also be done using string concatenations, For example,

```
print("\\(\")/\n" ..
      " (|)\n" ..
      "/(_)\\")
```

Alternatively, using a variable,

```
local ant =
"\\(\")/\n" ..
" (|)\n" ..
"/(_)\\"

print(ant)
```

Note that we add newlines `"\n"` at the end of each of the top two lines. Why do we need to do that? Obviously, we need to do that since the ant is drawn in three lines. What is confusing is, in fact, why didn't we add the newlines in the previous examples? Why do we not add the newline at the end of the third line in these two examples?

Although we didn't explicitly mention it, an observant reader might have noticed it. The `print` function, by default, automatically adds a newline after the string argument when printing. Now, does it make sense that we have two newlines but not three in these two examples?

(You can change the default behavior of `print`, but we will leave it to the reader to find out how. Remember that the book covers only a portion of the Lua grammar, and that there are, or can be, multiple ways to achieve the same thing.)

4.4. Multiline Strings

Lua also supports a syntax for "multiline strings".

```
print([[
  Never say goodbye
  because goodbye means going away
  and going away means forgetting.
]])
```

If we run this program, it will print the sentence in three separate lines, as shown. There are a few things to note here. First, syntactically, a multiline string starts with `[[` and ends with `]]`. Second, a multiline string *can* span multiple lines. Third, newlines in a multiline string need not be escaped. The newlines in the string are directly printed out to the output. Fourth, likewise, special characters like backslashes need not be escaped. We can also use variables for multiline strings:

```
local wisdom = [[
  All of this has happened before,
  and it will all happen again.
]]
print(wisdom)                                    ①
```

① Readers are encouraged to try it out to see what kind of output they get.

4.5. Back to the Ant!

Here's the ant using the multiline string syntax:

```
print([[
\(")/
 (|)
/(_)\]])
```

If your run this program, it will print exactly the same ant to the console.

The "ant" in this form (e.g., without escape characters) seems a bit easier to recognize. In general, multiline strings can be useful in many circumstances, especially in places where many escape characters would otherwise have been required.

You can also use a name for this multiline ant string:

```
local ant = [[
\(")/
 (|)
/(_)\]]

print(ant)                                    ①
```

① If you'd like to follow along, you can try this code on Replit, for example.

Exercises

Ex 1. Hello you!

The "hello world" program is a terrible program since it does only one thing. ☺ Let's fix that. (Not right now though. We will do it throughout the rest of this book.)

Here's an example program that prints *Hello, Peter*.

```
local name = "Peter"
local greeting = "Hello, " .. name
print(greeting)
```

Write a Lua program that prints the following:

```
Hello, Peter Pan
Hello, Tinker Bell
Hello, Wendy Darling
Hello, John Darling
Hello, Michael Darling
Hello, Tiger Lily
Hello, Tick Tock
Hello, Captain Hook
```

(Remember that there is no single right answer. You can do this in many different ways.)

Ex 2. Everything in one line?

Write a Lua program that prints the same *ant*, but put everything in one.

(Obviously, the program has to work. ☺)

Ex 3. More ASCII arts

Here are some more *small* ASCII arts that we found on the Internet. (We believe that they are all in the public domain.)

Write a Lua program that prints these character-based drawings to the console.

Butterfly

```
 (\o/)
(  |  )
 (/!\)
```

Star

```
__/\__
\ ,, /
/_  _\
  \/
```

Bird

```
  ___
 (@ @)
(  V  )
--m-m--
```

Spider

```
 !  !
 \()/
/(__)\
!    !
```

Floppy

```
 ______
| |__| |
|  ()  |
|______|
```

Glass

```
\~~~/
 \ /
  V
  |
 ---
```

House

```
  /^^\
 |  # |
 |====|
_||__||_
```

Airplane

```
    __!__
_____(_)_____
    !   !
```

Sword

```
     /| ____________
0|==|* >____________>
     \|
```

Person

```
       ,,,
      (o o)
---oOO--( )--OOo---
```

Ex 3. Bird * 10

Print the bird ten times in a row.

Ex 4. A snowman

Use your creativity, or artistic skills (or, your Web search skills ☺), to draw a snowman using only ASCII characters. Write a Lua program that prints the snowman to the console.

Ex 5. Ant and butterfly

Write a program that prints an ant and a butterfly side by side, that is, something like the following:

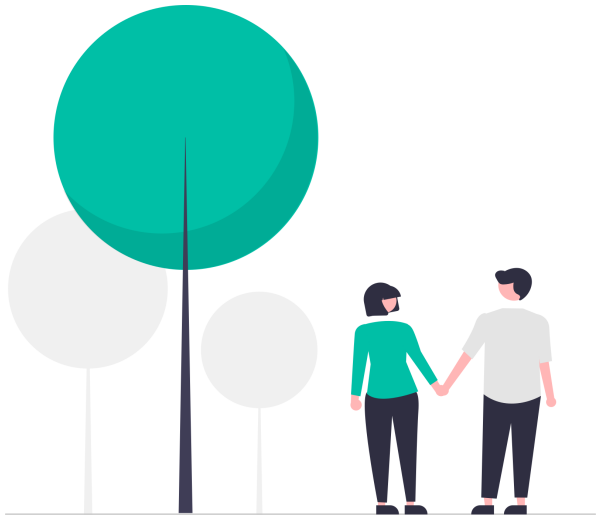
```
\(")/    (\o/)
 (|)    (  |  )
/(_)\    (/!\)
```

These are not easy problem, depending on how you going to do it. Remember, "getting things done" (by any means ☺) is the first step. Doing well, doing better, doing more efficiently, etc., only matter only when you can do it in some way in the first place.

Ex 6. Build five houses

A similar problem. These exercises are meant to make you think rather than to test your programming skills. Print five houses side by side, not one after another. Or, how about 3 birds on the same wire? Or, can you draw 10 crawling spiders side by side? ☺

Lesson 5. Hello, All the Moons in the World!



We learned in the previous lesson that Lua programs deal with values like numbers and strings. These values are somewhat special in that they are *simple*.

Programs often deal with values which are more *complex*. For example, a value may contain other values, some of which may contain other values, and so on. In programming, terms like containers, collections, or data structures, etc. are generally used in this context. More specifically, many programming languages support special types like arrays, lists, and dictionaries (or, maps), etc.

Lua supports this kind of complex values using a special syntactic construct called the "table". `table` is a type, and its values are also called tables. Lua does not include any other collection types. (As briefly alluded before, a table *type* is a set of all table *values*.)

Tables play crucial roles in many different aspects of Lua programming. Although the use of the term `table` is somewhat unique to Lua, Lua tables roughly correspond to "objects", as this term is generally used in other programming languages like JavaScript.

We will learn some simple uses of tables in this lesson. In particular, we will use tables as *arrays* and *maps*. Furthermore, we will learn how to "iterate" over tables. In general, iterating over elements of a collection, for various purposes, is one of the most common tasks in programming. In a somewhat abstract term, iteration is one way to control the flow of a program. As mentioned, programs are generally executed from top to bottom. As we will see shortly, the iteration allows us to repeat the same part of code multiple times.

5.1. Tables as Arrays

Lua provides a special syntax for representing table values. It is rather similar to the way string values are represented by the special syntax, e.g., starting with a double quote `"` and ending with a double quote `"`.

Likewise, a table starts with an opening curly brace `{` and it ends with a closing brace `}`. Zero, one, or more "elements" of the table are included between these pair of braces, separated by commas `,`. This is also similar to the way the (real) string content is enclosed within the given pair of double quotes. For instance,

```
local tab1 = { 1, 2, 3 }
```

In this example, the right hand side of the assignment is a table. This table `{ 1, 2, 3 }` contains three elements, `1`, `2`, and `3`, each of which is a value, or more precisely a number in this particular example.

Again, this is the "syntax", or rules. There is no "why?". This is how the Lua programming language is designed. Learning, in this context, means *getting familiarized* with this syntax, and hopefully remember it when you need to use it. Don't try to "understand" it.

An *array* is a general term commonly used in programming that, roughly speaking, refers to a sequence of values (as in "1 *and then* 2 *and then* 3"). A Lua table used in this way is an array. (Different programming languages may use different syntax, but the general concept is the same.)

Why would anyone need, or use, arrays? That's too broad a question to answer in a short beginner's book, and you will have to learn that through experience.

As a general advice, however, the more syntax you learn that is supported for a particular construct (like `table`), the more of their uses you will come to realize yourself, and learn, over time, without having to be told explicitly what they are.

In case of Lua `table`, it supports two special syntax, among others, for "indexing" and "iterating". Before we get to them, let's start from the basics.

5.1.1. Creating an array

Since you have seen *one* example so far ☺, how would you create an empty array, that is, an array with no elements?

Well, you are right. Just put nothing inside the braces:

```
local emptyArray = {}                    ①
```

① This also represents an empty *map,* as we will see in the next section. In fact, it is just an empty table. Syntactically, there is only one construct, `table`, in Lua. We are just *using* it in certain ways, and we are providing particular *meanings* to them.

Can you now create an array with four string elements, `"apple"`, `"orange"`, `"pear"`, and `"avocado"`, and give it a name, `fruits`? Maybe, something like this?

```
local fruits = {
  "apple", "orange", "pear", "avocado"
}
```

Or, maybe, something like *this*?

```
local fruits = {
    "apple",
    "orange",
    "pear",
    "avocado",
}
```

As emphasized a few times before, "formatting" is not that important in Lua, unlike in some other languages, and white spaces, including newlines, can be liberally used to format the source code to your liking.

5.1.2. Accessing an element in an array

Each element in an array can be accessed using "indexes". The index of a given element refers to the position of the element in the array, starting from left. That is, the first element has an index `1`, the second

element has an index 2, and so forth. For instance, using the above example, `fruits[1]` refers to `"apple"`, `fruits[3]` refers to `"pear"`, etc. Note the syntax. It uses the square brackets after the array name, in which the index (a number) is enclosed.

Despite the syntactic difference, the index notation is essentially just a name. The element `"avocado"` can be likewise referred to as `fruits[4]`. Here's another example array, using our three most favorite insects ☺,

```
local bugs = {                          ①
  "ant", "bee", "cockroach",            ②
}
```

① The new array on the right hand side is given a name `bugs` in this example.

② Note that the elements are separated by commas, and the trailing comma is optional, as you might have noticed from the earlier examples.

Now, what do you think the following program will do, using the above `bugs` array?

```
print(bugs[1])
print(bugs[2])
print(bugs[3])
```

This program is, effectively, the same as the following:

```
print("ant")
print("bee")
print("cockroach")
```

That is, `bugs[1]`, `bugs[2]`, and `bugs[3]` are essentially just names that refer to the first, second, and third elements of `bugs`, respectively,

5.1.3. Iterating over an array

One of the nice things about collecting multiple values into a single array is that one can easily *iterate* over these values, e.g., to do something with those values.

Lua provides a special syntax for iterating over arrays, which is called the `for` statement. The `for` statements typically span multiple lines, and they may look very complicated, more complicated than anything we have seen so far, but, remember, it's just a *form* (as in "beauty is only skin deep"). Here's an example:

```
local pets = { "Dog", "Cat", "Bird" }    ①
for i, pet in ipairs(pets) do            ②
  print(i .. ": " .. pet .. ".")         ③
end                                      ④
```

① A 3 element array. The number of elements is called the "length" of the array. The length of `pets` is `3`.

② Scanning this line from the beginning, you see (i) `for`, which happens to be a Lua keyword, (ii) two names `i` and `pet` separated by a comma (`,`), (iii) another Lua keyword `in`, (iv) a function named `ipairs` with an argument `pets`, and (v) finally another keyword `do`. Clearly, if you have never seen the `for` statement before, it will not be easy to recognize all these, but at least you can (roughly) tell which are tokens, e.g., based on white spaces and punctuation symbols.

③ The `for` statement can include other statements. In this example, it includes one `print` statement. Note that we use the variable `pet` in this statement, which is declared in the previous line.

④ The keyword `end` matches the opening keyword `do`. Together, they form a sort of braces. The statements like `print` in this example are enclosed within this `do` - `end` block, if you will.

There are so many things going on in this sample code. But, you do not have to be overwhelmed. Remember, "analysis and synthesis". Everything is simple once you break it down into smaller pieces. Everything is easy once you get to know it. (Also, remember, it takes time, and most likely many repetitions, to get to know something, or someone.)

Let's go over the syntax first. We mentioned four new keywords, `for`, `in`, `do`, and `end`. (Keywords are just names that have special meanings to Lua.)

```
for ... in ... do ... end
```

That's the overall structure of the `for` statement. Between `for` and `in`, we need (that is, syntactically) two variables, separated by a comma (`,`). In the example, we used `i` and `pet`, but these names are arbitrary as long as they are lexically correct.

Between `in` and `do` comes a function call. Lua allows two builtin functions to be used in this context, `pairs` and `ipairs`. In this example, e.g., when a table is used as an array, they are *more or less* the same. We will use `pairs` in the next section.

If you will re*call* (no pun intended), a function call expression has this syntax, a function name followed by an argument list in a pair of parentheses. The `ipairs` function takes one argument, which must be a `table`. In this example, we use an array, `pets`, as an argument, which has three elements.

What does the `ipairs` function do? That is, in fact, a part of the `for` statement semantics. In each iteration, it selects an element, in the specified order in the array, and it returns a pair of its index and its value.

Between `do` and `end`, we can include zero, one, or more statements (even other `for` statements). This is where the magic happens. The `for` statement repeatedly executes this series of statements, by going through each element of the array argument of the `ipairs` function, from the first element to the last element.

More specifically, it takes an element of the given array, one at a time, and assigns it to the second variable between `for` and `in`. The first variable, e.g., `i` in this particular example, gets the index of the chosen element, in this particular usage of the `for` statement.

Next, it runs the enclosed statements, sequentially. If it is done with the current element, it picks the next element in the array and again executes the enclosed statements. Once it runs out of the elements in the given array, the `for` statement stops executing, and program flow continues to the next statement, if any.

Let's go through the earlier `for` statement together.

```
for i, pet in ipairs{"Dog", "Cat", "Bird"} do
  print(i .. ": " .. pet .. ".")
end
```

First iteration: We pick `"Dog"`, the first element, and assign it to `pet`. `i` is `1`. Then, we execute the enclosed `print` statement.

This is roughly equivalent to the following.

```
local i, pet = 1, "Dog"                    ①
print(i .. ": " .. pet .. ".")
```

① This statement declares *two* variables, `i` and `pet`, and initializes them with `1` and `"Dogs"`, respectively. This is essentially a generalization of the variable declaration statement that we learned earlier.

Or, even

```
print(1 .. ": " .. "Dog" .. ".")
```

And hence, this will output, in its first iteration,

```
1: Dog.
```

Can you see it?

Now that it is done with the first element, it next picks the second element, which is `"Cat"`. In this case, the index `i` becomes `2`.

```
local i, pet = 2, "Cat"
print(i .. ": " .. pet .. ".")
```

Or, in effect,

```
print(2 .. ": " .. "Cat" .. ".")
```

This will, therefore, output, in its second iteration,

```
2: Cat.
```

Then, it repeats the same steps by selecting the third element, `"Bird"`, with the index `3`. In the third iteration, the enclosed statement(s) may effectively act like this:

```
print(3 .. ": " .. "Bird" .. ".")
```

This will end up printing

```
3: Bird.
```

At this point, since there are no more elements in the `pets` array, the `for` statement is done. The overall output may look as follows:

```
1: Dog.
2: Cat.
3: Bird.
```

If you have seen this before, it should be easy. Otherwise, you can go through this again, and make sure that you understand this. This is called the "iteration" or "looping" in programming, and it is one of the most important concepts (regardless of the particular language syntax) that you will have to learn as a beginner. This will eventually become your second nature, but you will not get there unless you practice.

5.2. Tables as Maps

A map (aka, a hashmap, hashtable, or dictionary) is another important data structure that is commonly used in programming. (As we briefly mentioned earlier, a "data structure" is just a *complex* value, which can include, or contain, other values, for example.)

A map is a data structure that is essentially a collection of key-value pairs. That is, each entry in a map consists of two values, a *key* and a *value*. One of the most important characteristics of the map data structure is that it lets you *easily* find an entry based on its key. ("Easily" is the *key* word here, but we will not get into algorithms and data structures too deep in this book. ☺)

In Lua, the same syntactic construct `table` can also be used as a map. In fact, using tables as arrays or maps has rather similar syntactic structures. Let's start from the beginning, as in the previous section.

5.2.1. Creating a map

An empty table is really neither an array nor a map. Whether it is an array or a map depends on what kind of elements we add to the table. Here's an empty "map",

```
local emptyMap = {}
```

Now, you can create a map with initial values using a few different syntax in Lua. We will use one method in this book. In particular, each initial entry can be specified with an assignment-like syntax. For instance,

```
local capitals = {
  UK = "London",
  France = "Paris",
  Germany = "Berlin",
}
```

In this example, we create a map with three entries and assign it to a variable named `capitals`. Each entry is a pair of two values, `UK` → `"London"`, `France` → `"Paris"`, and `Germany` → `"Berlin"`. As we noted earlier, `UK`, `France`, and `Germany` are called the keys, and `"London"`, `"Paris"`, and `"Berlin"` are the corresponding values. Note that there are some kind of implicit *mappings* from keys to values, in a *map*.

5.2.2. Looking up an element in a map

Arrays use numeric indices, and their elements can be accessed using the corresponding indices. In case of maps, their elements can be accessed using their keys, but using a rather similar (square bracket) syntax. For instance, using the above `capitals` example,

```
print(capitals["UK"])
print(capitals["France"])
print(capitals["Germany"])
```

This program will print out the following:

```
London
Paris
Berlin
```

In an array, its indices are sequentially numbered, and hence (naturally) they are all different. In a map, there are no intrinsic ordering among the entries, but their keys still have to be all different.

5.2.3. Iterating over a map

A map can be iterated over just like an array, but using their keys rather than indices. Let's use the following map for illustrating iteration.

```
local countryCode = {
  USA = 1,
  Canada = 1,
  Mexico = 52,
}
```

There are a couple of things to note. First, as indicated, all keys have to be different. But, that does not mean all values have to be different too. In this particular example, `USA` and `Canada` happen to have the same country code. Second, the types of the values can be arbitrary. In the previous example, the type was `string`, and in this example, it is `number`.

To iterate over a map, we can use the same `for` statement, but now with the `pairs` function, instead of `ipairs`. The difference is that the `pairs` function, in each iteration, returns a pair of key and value of a given entry when used in the context of maps, rather than a pair of index and its element as in case of arrays using the `ipairs` function.

Some readers might have noticed that these `pairs` and `ipairs` functions are rather different from the functions we used before, like `print` and `type`, and the functions that we will work on in later lessons.

We will not get into why `pairs` and `ipairs` functions work the way they do, or what is the precise difference between these two functions, etc., in this book. Curious readers can look up relevant information on the Web, or do some experiments on their own, e.g., by swapping `ipairs` and `pairs`, etc.

For example,

```
for k, v in pairs(countryCode) do
  print("Dial +" .. v .. " first to call a
number in " .. k)
end
```

This will output

```
Dial +1 first to call a number in USA
Dial +1 first to call a number in Canada
Dial +52 first to call a number in Mexico
```

As we did for an array iteration, we can also go through a map iteration, by picking one entry at a time. If it is not clear to you how iteration over a map works, the readers are encouraged to go through the same exercise, e.g., using this `countryCode` map.

5.3. Hello, the Moon!

OK, *back to the Moon!* Let's pay homage to our Moon before we start. Our solar system has eight planets (sorry, Uranus), and some of them have moons. The earth has only one moon, but some bigger planets like Jupiter and Saturn have dozens of moons each, and possibly even more depending on how you exactly define the "moon".

To our one and only one, and *special*, Moon,

```
print("Salute, the Moon!")
```

5.4. Hello, the Moons of Mars!

Our next door neighbor, Mars, has two moons, Phobos and Deimos.

```
local moons = { "Phobos", "Deimos" }
```

Let's say hi to them.

```
for _, moon in ipairs(moons) do
  print("Hello, " .. moon .. "!")
end
```

The underscore `_` is a special name predefined in Lua, which can be written to but cannot be read from. It simply means that we are going to ignore the indexes of the elements in the array (or, the keys of a map). We only care about their values, or the names of moons in this example, denoted by the variable `moon`. (In other words, although, syntactically, two names are required, we only care about the second name, and hence we use `_` for the first one. This is sometimes called a "discard" variable.)

This program will print,

```
Hello, Phobos!
Hello, Deimos!
```

Do you see how we would get this output? If not, go through the iteration steps, one element at a time. Start from the first element, `"Phobos"`, and so on. We will leave it as an exercise to the readers.

5.5. Hello, the Moons of Both Planets!

For practice, let's try saying hello to the moons of both planets, Earth and Mars. We will say hello to each of the three moons, with their parent planet.

For example, here's an expected output. (Let's suppose that the order is not important, to make our job easier.)

```
Hello, Moon of Earth!
Hello, Phobos of Mars!
Hello, Deimos of Mars!
```

How would you go about doing this? If you have never seen this before, it will be rather difficult to figure this out on your own. Don't worry about it.

Here's one way to do this.

```
local lunas = {                                    ①
  Moon = "Earth",
  Phobos = "Mars",
  Deimos = "Mars",
}

for luna, planet in pairs(lunas) do                ②
  print("Hello, " .. luna .. " of " .. planet
.. "!")
end
```

① Notice that `lunas` is a map, and not an array, unlike `moons` in the previous example.

② This particular `for` loop uses `pairs`, and not `ipairs`. These two would produce different results.

As mentioned, complex types like maps are known as data structures in programming. In this example, the local variable `lunas` refers to a map, a data structure. This particular map includes three entries, and the keys, `Moon`, `Phoobos`, and `Deimos`, point to the values, `"Earth"`, `"Mars"`, and `"Mars"`, respectively.

As we went through earlier, this `for` statement with the `pairs` function will iterate over all entries of `lunas`. In each of the three iterations, the values of `luna` and `planet` will be `Moon → Earth`, `Phobos → Mars`, and `Deimos → Mars`. That's how we get the desired output.

It is somewhat hidden in the notation, but the reason why we have chosen this particular data structure is because of the inherent mapping structure, from a key to its value, within a map data structure. For example, in particular, `lunas[Moon]` is `"Earth"`, and `lunas[Phobos]` is `"Mars"`, and so forth.

The `for` statement, in fact, hides these details (by design), but if you think about it, the value of `lunas[luna]` is always the same as `planet`, *in each iteration*.

Note that variables like `luna` and `planet` are often called the "loop variables", e.g., because they are used in the loop. As indicated, their names are largely arbitrary, to a large extent, except that, as a general principle, good names make the program easier to read.

By the way, if you run this program on Replit, or on your computer, the output might look a bit different. This is because, in a map, the orders are not fixed. (This can be a difficult concept to grasp at this point, depending on your prior exposure to programming, and if it does not make sense to you, you can ignore it. Unless you are too busy ☺, practicing on your own, e.g., using example code from this book, will help you understand better. We can include only so much in the book. *Don't be a passive learner*, who just complains that the book does not make sense. ☺)

As we emphasize throughout this book, there are more than one way to do the same thing. For this particular problem, here's an alternative way:

```
local satellites = {
  Earth = { "Moon" },
  Mars = { "Phobos", "Deimos" },
}

for planet, moons in pairs(satellites) do
  for _, moon in pairs(moons) do          ①
    print("Hello, " .. moon .. " of " ..
planet .. "!")
  end
end
```

① One can use either `pairs` or `ipairs` in this context.

Note that the variable `satellites` is a map. On the other hand, the values of each of its entries are arrays. This is one of the more complicated programs we use, and see, in this book. But, we already learned everything that is need to understand this program. There is a *nested* `for` loop (e.g., a `for` statement within another `for` statement), which is "new" in a sense. But, then again, it's not really new, if you understand how the `for` statement works. We will leave it as an exercise to the readers to go through this nested loop.

And, remember, if you don't understand this program, that's not the end of the world. You can ignore it and move on. (It's better to *live to fight another day* than burn yourself out. Also remember that you can always "backtrack", and you will surely have a better chance the next time around.)

Exercises

This lesson has been somewhat technical, and depending on your prior coding experience, you may have found it rather difficult. That's quite all right. As stated, 99% of learning is just getting familiarized, and that *takes time* and *repetitions*. Just to remember to backtrack and iterate, as you feel necessary, and as often as possible.

Ex 1. Hello, all!

Since we know how to do iteration, let's say hi again to some people.

- First, create an array of people's names. For example, in the previous section, we used the names of the characters from Peter Pan.
- Then, write a Lua script that goes through each of these names and say "hello" to them.

For example, here's an example output:

```
Hullo, Peter~~
Hullo, Wendy~~
Hullo, Tinker Bell~~
```

Ex 2. Art map

We included around 11 ASCII art samples in the previous lesson.

- Create a map using their names as keys and their multiline drawings as their values.
- Iterate over the map and print out all drawings vertically, one after another. You might want to add an empty line or two between the drawings so that they are not all merged together.

Ex 3. Galilean moons of Jupiter

In the history of human civilizations, we went though a few pivotal moments. The invention of the telescope was one of them. Pioneering scientists like Copernicus and Galileo, paved the way, using this new invention at the time, for the following, and still ongoing, scientific revolutions, for the next 400 years and more.

Galileo discovered four (biggest) moons of Jupiter using a telescope, over four centuries ago. They are often called the Galilean moons of Jupiter in honor of him. They were later given names, by Kepler, *Io*, *Europa*, *Ganymede*, and *Callisto*. Incidentally, as you may know, there is another "artificial moon" revolving around Jupiter in the last several years (and, probably for some more years to come), aptly named *Juno*.

Now, let's send some greetings to these four moons of Jupiter. First, create an appropriate data structure with these four moons. Then, iterate over this data structure, and print out something like the following to the console:

```
Hello, Jupiter's first moon, IO!
Hello, Jupiter's second moon, Europa!
Hello, Jupiter's third moon, Ganymede!
Hello, Jupiter's fourth moon, Callisto!
```

Lesson 6. Day of the Week

Programming languages specify the language grammar, both syntax and semantics. In addition, all modern programming languages also provide specifications for what is known as the "standard libraries". A library, in general, is a collection of predefined functions and custom types, etc. And hence, the standard library is a set of libraries whose content is determined by the programming language specification.

Learning a new coding language really means learning its standard libraries as well. In modern programming, one cannot effectively program without using the standard libraries.

6.1. Lua Standard Libraries

Some programming languages, like Python, provide an extensive set of standard libraries. Lua, on the other hand, includes only a bare minimal set. Clearly, there is a trade-off. In a sense, because of that, the importance of standard libraries is even bigger in Lua.

Lua's standard libraries are classified into a few different tables. (That is, the same `table` that we used for arrays and maps.)

For example,

- `string`: The `string` library includes functions for string manipulation and pattern matching.
- `table`: It includes functions for table manipulation such as concatenating two arrays.
- `math`: It includes basic mathematical functions. E.g., arithmetic and trigonometric functions.
- `io`: It provides functions for basic I/O.
- `os`: It provides core operating system facilities.

Going through them all is well beyond the scope of this beginner's book. But, as an illustration, we will look at a few concrete examples in this and the next lesson so that the readers can explore them further on their own, if needed, while learning programming in Lua.

6.2. Function Calls

Although we have been using some functions, especially the `print` function, it may be a good time to review Lua's function syntax in some more detail. In particular, we will look at how to call functions in this section, and how to define our own functions in the next.

A function call in Lua is an expression, and it has the following syntax:

```
FUNCTION-NAME ( ARG1, ARG2, ... , ARGn )
```

The argument list can include zero, one, or more values (or, more generally, expressions), and the number of required arguments is typically determined by the function definitions. In some special cases, like the builtin `print` function, one can call a function with different numbers of arguments. For example,

```
print(1, 2, 10)
print("I", "am", "a", "programmer")
```

This will print

```
1       2       10
I       am      a       programmer
```

6.3. Function Definitions

In Lua, one can define a custom function in a few different ways. We will use only one method in this book. Here's a common syntax:

```
function FUNCTION-NAME ( ARG1, ... , ARGn )
   STATEMENT-LIST
end
```

As in the previous section, all-capital-letter names are placeholders. On the other hand, `function` and `end` are Lua keywords. As before, the parentheses, `(` and `)`, are lexical tokens, which is part of this function definition syntax.

Hence, in plain English, a new function can be created by declaring its name after the keyword `function`, and the function definition ends with the keyword `end`. After the function name follows a parameter list within a pair of parentheses. A function can be declared with zero, one, or more parameters (denoted by `ARG1` through `ARGn` in the above syntax notation).

The `STATEMENT-LIST` represents a list of statements, as in the case of the `for` statement, and when this newly-defined function is *called*, this series of statements will be executed.

This explanation may seem too technical, but let's see some concrete examples.

```
function f1() end                          ①
function f2(a, b) end                      ②
function f3()                              ③
  return 42
end
function f4(a, b)                          ④
  return a + b
end
function f5(tab)                           ⑤
  for k, v in pairs(tab) do                ⑥
    print(k, v)
  end
end                                        ⑦
```

① The function `f1` accepts no arguments, and it does nothing. Note the (largely arbitrary) location of `end`. As stated, in Lua, formatting is not that important.

② The function `f2` accepts two arguments, denoted as `a` and `b`, and it just ignores them. This function again does nothing.

③ The function `f3` accepts no arguments, and it simply returns a number `42`. The `return` statement is used to return a value from a function, which becomes the value of the function call expression. That is, in this example, the value of the call `f3()` will be `42`.

④ `f4` accepts two arguments and returns its sum. That is, `f4` is an addition function.

⑤ `f5` accepts one argument of a `table` type, which can be seen from the implementation in the next three lines (aka, a "function body").

⑥ As we have seen earlier, this `for` loop syntax only works when the argument `tab` is a table. If not, it will fail. In *simpler* languages like Lua, this kind of type errors are more common. This discussion is, however, beyond the scope of this book.

⑦ This function `f5` does not return any (meaningful) value. The only reason why we would call this function is for its "side effects", namely, printing the key-value pairs of the given table in this example.

6.4. Date and Time Functions

6.4.1. The `os.date` function

The `date` function from the `os` library returns the date and time based on the given argument(s), either as a table or as a string. It can be used in a few different ways, and we will use the following two particular syntax in this lesson.

```
os.date("*t")                                    ①
os.date("*t", time)                              ②
```

① This function call returns today's date and time as a table. `"*t"` is a special format that indicates that `os.date` should return a table value.

② It works the same way, but it uses the given `time` instead of today's date. The `time` argument is a number, more specifically, the number of seconds since the "Unix epoch". See the next section.

The returned table will include the entries with keys, `year`, `month`, and `day`, etc. For our purposes, we are only interested in the value of the key `wday`, which returns a number between `1` and `7`, representing Sunday through Saturday. For example,

```
t = os.date("*t")
for k, v in pairs(t) do print(k, v) end
```

This code will print the following:

```
year    2023
month   3
day     16
wday    5
hour    12
...                                    ①
```

① Some output lines are omitted for brevity. `wday = 5` indicates that today is Thursday.

6.4.2. The `os.time` function

The `time` function in the `os` library returns a "Unix epoch time", that is, the number of seconds since *00:00 AM on January 1st, 1970, UTC*. The `os.time` function supports two different syntax.

```
os.time()                                ①
os.time(tab)                             ②
```

① It returns the Unix epoch time of "now", e.g., the current time at the time of the function call.

② It returns the Unix epoch time of the date and time specified by `tab`. This argument uses the same table format as those returned by the `os.date` call.

For example,

```
print(os.time())
```

This statement will print a number like *1678985623*. Likewise,

```
print(os.time{year=2023, month=1, day=1})
```

This will print a number *1672596000*.

6.5. Today, and Today Only

The problem of this lesson is to find the day of the week for a given calendar day. We will do this in two parts. First, we will just use today as the target date in this section. A slightly more general problem will be discussed in the next section.

In a sense, this is a very easy problem. You can just look at your smartphone, and it will tell you what day of the week it is today. But, although it may seem so trivial, this information is the result of *some computation* (e.g., done by your expensive smartphone). In Lua, we can do this using the aforementioned pair of library functions `os.date` and `os.time`. To be concrete, let's define the problem more precisely:

The program, when executed, should print out a string, Sunday, Monday, ... Saturday, based on today's date.

How would you do this?

First, we know that `os.time()` returns the Unix epoch seconds of today. Second, if we call `os.date("*t", time)` with the `time` argument, it will return the date and time in the particular table format. And, we know that the `wday` field includes the weekday information, coded in numbers `1` through `7`. All we have to do now is to map the returned number to the string value, a desired output.

Here's a map that encodes this weekday information.

```
dayOfWeek = {
  1 = "Sunday",
  2 = "Monday",
  3 = "Tuesday",
  4 = "Wednesday",
  5 = "Thursday",
  6 = "Friday",
  7 = "Saturday",
}
```

In Lua, this map is equivalent to the following array,

```
dayOfWeek = {
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
}
```

That is, when the keys of a map are consecutive integer numbers starting from 1, the map can be viewed as an array with their values as elements. They are semantically equivalent.

This may be somewhat of TMI (too much information) at this point, but it is important to note that we may still be using "mapping", conceptually, although at times we use the array syntax in Lua.

Next, we can encapsulate the part for computing today's day of the week in a function, since we now know how to define our own function. Here's a skeleton (that is, without a real implementation),

```
function weekday(today)                    ①
  return ""                                ②
end
```

① The `today` parameter is a Unix epoch time, that is, an integer number.

② This function will return one of the weekday string values, as defined above. For now, it's a placeholder.

The top-level logic of our program might look like this:

```
local today = os.time()
local day = weekday(today)
print("Day = " .. day)
```

Does this make sense to you? We first get today's Unix epoch time, by calling the library function, `os.time()`, and next we call our still-to-be-implemented `weekday`

function, which takes an epoch second and returns the weekday string. The final step is to output the result to the console, which we do with a `print` statement.

Now, here's the main logic of the `weekday` function.

```
function weekday(time)
  local d = os.date("*t", time)          ①
  return dayOfWeek[d.wday]               ②
end
```

① The `os.date("*t", time)` call returns a table, including the field `wday`.

② We use our previously-defined mapping table to convert the number to the corresponding string. Note that the syntax `d.wday` is the same as `d[wday]`, and hence `dayOfWeek[d.wday]` is also the same as `dayOfWeek.d.wday`, for instance.

The readers are encouraged to combine all these pieces together and make a full functioning program.

Do not copy the code. Try to understand what each piece of code does, and *write* your own code from your memory. Your program may end up being slightly different from the code examples in the book, but that is perfectly all right.

6.6. Are You a Wednesday Child?

We will generalize this program so that it can be used with an arbitrary date, and not just with today.

Since we already know how to do this for a given Unix epoch time, that is, through the `weekday` function, all we have to do now is to get the Unix epoch time of a given date. And, we know how to that, e.g., by calling `os.time()` with a table that represents a particular date and time.

For practice, let's try and create another function, which takes `year`, `month`, and `day` as arguments, and returns the Unix epoch second.

```
function getTime(year, month, day)
  return os.time{                          ①
    year = year,
    month = month,
    day = day,
  }
end
```

① Note that, although we stated that the function call syntax requires the parentheses, they are optional in some cases. This is really a detail, but when a function takes a single table, or a single string, as an argument, the parentheses can be omitted.

Here's a complete program with an example date, the last day of the last century:

```
local year, month, day = 2000, 12, 31
local time = getTime(year, month, day)
local wday = weekday(time)
print("Day of the week = " .. wday)
```

In fact, we can make this whole program even "simpler" by using another function:

```
function printDayOfWeek(year, month, day)
  local time = getTime(year, month, day)
  local wday = weekday(time)
  print("Day of the week = " .. wday)
end
```

Now, our main program may look like this, using the author's birthday ☺ as an example,

```
local year, month, day = 2020, 1, 1
printDayOfWeek(year, month, day)
```

So, are you a Wednesday's child? Try coding a similar program on your own running it with your birthday. (About 14% of people are Wednesday's children. How do you think we came up with that number? ☺)

Exercises

We learned in this lesson

- The `os.date` and `os.time` library functions,
- How to call a function,
- How to define a custom function, and
- How to use Lua's `table` for mapping purposes.

Ex 1. Hello, function!

Write a function that

- Takes a string argument, named `name`, and
- Prints *Hello, name!*, where `name` is the argument of the function.

Ex 2. Length operator

Although we did not discuss in this book, Lua supports a length operator, `#`, which returns the number of elements in a table, e.g., an array or a map. That is, for example,

```
local a = { 5, 10, 15 }
local t = { a = "A", b = "B" }
print(#a, #t)
```

This code will print *3 2*. Now, write a function that takes a table as an argument and prints all elements of the given table and its length. For instance, here's a sample output, using `t` above as an example argument,

```
a       A
b       B
-------------------------
The number of elements: 2
```

This should work for both arrays and maps.

Ex 3. Conditional processing

Another important control flow statement is the `if` statement, also known as the conditional statement. The readers are encouraged to look up the general syntax of Lua's `if` statement on the Web, which can include keywords like `if`, `then`, `elsif`, and `else`. For the purposes of this exercise, let's take a look at a simple example.

```
local x = 10
if ( x > 5 ) then                          ①
  print("x is bigger than 5")
else
  print("x is equal to, or less than, 5")
end
```

① You may have never seen `if` statements before, but you can still recognize lexical tokens and what not. In this example, you can identify the overall form of this `if` statement, i.e., `if` - `then` - `else` - `end`, etc.

The expressions like `x > 5` are evaluated to Boolean values, e.g., `true` or `false`. In this particular example, since `x` (whose value is currently `10`) is bigger than `5`, it evaluates to `true`. In the `if` statement, if the Boolean expression is `true`, then the statements between `then` and `else` are executed. Otherwise, the statements between `else` and `end` are executed. Hence, if we run this example code, it will print out, *x is bigger than 5*.

BTW, the Lua's length operator `#` also works with strings, and it returns the number of characters in the given string. For example, `#"hello"` evaluates to `5`. Now, write a function that accepts an array of strings, and prints each string along with a label, *long* or *short*. More specifically, if a string is the same as, or longer than, 10 characters, then you use the label *long*. Otherwise, *short* is used. For instance, given an array, `{ "The moment", "you doubt", "whether", "you can fly," }`, here's a sample output:

```
The moment          long
you doubt           short
...
```

Ex 4. Mappings as a magic wand

We wrote a function earlier that prints out the day of the week as a string, given a date. Instead of printing the weekday strings, print one of the two strings, *Happy* and *Not so happy.* In particular, if the given date falls on a weekend, e.g., Sunday or Saturday, you print *Happy*. Otherwise, you print *Not so happy.* You can use the mapping, e.g, using a table, or you can use the conditional statement, if you like.

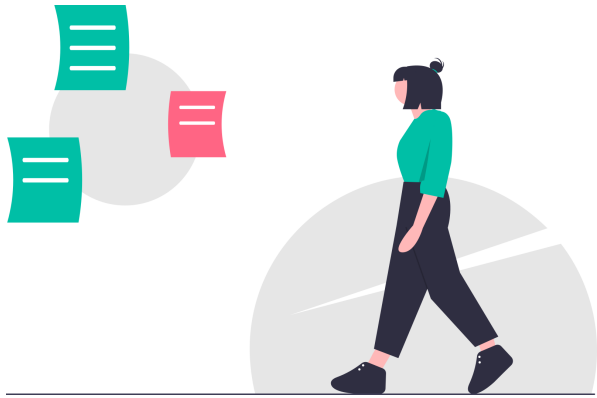
Ex 5. Is today a prime day?

In mathematics, a prime number is an integer that cannot be divided by any integers other than `1` and the number itself. Write a function that takes a date as an argument and returns `true` or `false` depending on whether the day (e.g., 1 through 31) of the given date is a prime number or not.

Ex 6. Back to the ASCII art

In an earlier lesson, we played around with some 11 ASCII art drawings. Write a function that takes the name of a drawing and prints out the corresponding drawing. For example, if you call your function with an argument `"ant"`, it should print the ASCII drawing of the ant.

Lesson 7. What is Your Sign?



Although we are now more used to using graphical user interface (GUI), and the apps specifically created for graphical interface, many useful and essential programs are still written for the so-called "command line" interface. Or, CLI for short. It is a character-based interface, and these CLI apps are typically used on terminals (or, consoles).

Let’s try building a slightly *more complete* program in this lesson that handles input and output via the command line interface. The fundamental ideas for handling user input and output will be also applicable when, and if, you decide to develop GUI-based programs in the future. First, some quick intro to the basic input/output in Lua…

7.1. Basic Input and Output

Lua includes basic command line-based input and output support, just like most other high-level programming languages.

The builtin `print` function that we have been using is mainly used for "debugging" purposes in Lua. The readers who have used JavaScript will be familiar with the `console.log` function. Their intended uses are more or less the same. They are primarily used, for example, to see what is going on inside the program, while the program is running.

In Lua, the standard input and output facilities are included in the `io` and `os` tables. (IO and OS stand for input/output and operating systems, respectively.) The IO functions defined in these tables are specially designed for user input and output processing, and they are not only more flexible than `print` but they can also handle special circumstances, for instance, if you need to run a Lua program in an environment where input and output have been redirected.

We will briefly discuss two functions, `io.read` and `io.write`, before we start working on our main program, which is sort of a (fake) astrology app. It reads the user input as a Zodiac sign, and prints out the corresponding horoscope.

7.1.1. The `io.write` function

First of all, notice that the function `write` is written as `io.write` with the `io` as a prefix (with a `.`). This is a common pattern. We used two other library functions, `os.date` and `os.time`, in the previous lesson.

Other than the builtin functions, most standard library and user-defined functions, and other values, are generally included in "modules", and their names need to be *qualified* with their module names. For example, in this case, `io` is the name of the module, and `write` is the name of the function.

The `io.write` function is defined as follows:

```
io.write(···)
```

The `io.write` function writes the given arguments to the standard output (e.g., the terminal). The `...` symbol represents an arbitrary number of function parameters, and they must be strings or numbers.

7.1.2. The `io.read` function

If you have used Python, Python includes a builtin function, `input`, for reading the user input from the command line. Lua's `io.read` is comparable to Python's `input` function.

The `io.read` function reads the user input from the standard input (e.g., the terminal), and it returns the value as a string or a number.

```
io.read()
```

The `io.read` function can take an argument that dictates the type and format of the return value, but we will only use the default format. In this simple use case, the `io.read` function returns the user input as a string.

7.1.3. I/O practice

Let's create a simple CLI program, for practice, which reads the user input from the terminal and prints out the result back to the terminal. This kind of program is often called "echo", for an obvious reason.

OK, so how would you go about creating a program like this? At a high level, we can easily think of a program flow, something like this:

1. Read the user input,
2. Store the value in a temporary variable, and
3. Write the value of the variable to the terminal.

The second step is optional. In fact, we can write the whole logic in one statement. But, as an exercise, let's do

this in a few distinct steps. It is a good practice to break a problem into smaller pieces and solve each small problem separately.

Since we learned how to define our own functions, let's create two functions for reading and writing.

```
function readText()
  return ""         -- tbd
end
```

```
function writeText(text)
  ;                 -- tbd
end
```

Note that, although these are just skeletons, we anticipate how these functions will ultimately work. For the `readText` function, it is supposed return a string value. For the `writeText` function, it takes a string argument, `text`. (The semicolon `;` represents an empty statement.) *Does this make sense to you?*

Now, without actually implementing these functions, we can write our main program as follows.

```
local input = readText()
writeText(input)
```

You read a user input, using the yet-to-be-implemented `readText` function, and we temporarily store it a local variable named `input`. Then, we write this `input` value to the terminal, or more precisely the standard out, using another yet-to-be-implemented function, `writeText`. This is the overall algorithm of our simple "echo" program.

All we have to do now is to implement these 2 functions.

```
function readText()
  io.write("What's your command? ")      ①
  return io.read()
end
```

① Before we read the user input, we write a "prompt" first so that the user knows that an input is expected.

```
function writeText(text)
  io.write("Your command is " .. text ..
"\n") ①
end
```

① Unlike `print`, `io.write` does not automatically append a newline at the end.

We will leave it as an exercise to the readers to code the entire program on their own and try running the program.

7.2. Game Plan

Let’s first review what we are going to build.

App Name

Horoscope Teller.

App Type

A CLI program/terminal app.

Requirements

- Ask the user for his/her astrology sign,
- Read the sign from the console,
- Generate a horoscope for the given sign, and
- Write the horoscope to the console.

7.2.1. Design

The word "design" often refers to graphic design. But, in programming, it also means software design. Suppose that we are assigned a task to build this *Horoscope Teller* app and we are given two hours.

What are we to do??? First, *don’t panic!* ☺

We just need some kind of a game plan. First, do we really understand the requirements? For example, what is an "astrology sign"? What is the "horoscope for the given sign"? How do we get the horoscope? etc. etc.

You can do some research and what not, and let's suppose that we understand *what* exactly we are supposed to build. For example, here's a list of zodiac signs.

```
local signs = {
  "aries",
  "taurus",
  "gemini",
  "cancer",
  "leo",
  "virgo",
  "libra",
  "scorpio",
  "sagittarius",
  "capricorn",
  "aquarius",
  "pisces",
}
```

Next, *how* are we going to build it? That's where the *software design* comes in. In some cases, drawing diagrams, etc., can be useful. For this task, let's just go through the requirements one by one and see how we can meet each of these requirements.

Ask the user for his/her astrology sign

Since it's a CLI app, we can use the `os.write` function to write the question to the console.

Read the sign from the console

We can use the `os.read` function to the read the user input. We will need to make sure that the read input is one of the valid signs. This is often called the error handling.

Generate a horoscope for the given sign

This is the core logic of our program. We will focus on this next.

Write the horoscope to the console

Again, we can use the `os.write` function to output the "generated" horoscope.

7.2.2. Core function

As we have seen, all requirements are mostly related to input and output handling, which may be viewed as somewhat peripheral. The exception is the one about "generating" a horoscope for a particular sign. That is the main part of the program.

So, how are we going to generate horoscopes?

7.2.3. Algorithm

"Algorithm" is another big word, which has many different definitions and subtly different meanings, e.g., depending on the contexts.

For our purposes, an algorithm is a *step by step* instruction to the computer. Here, "step by step" is the key word. Computers do not generally know how to do big tasks. For example, if you tell the computer to just create a horoscope app, it may not be able to do that. (Well, we'll have to wait and see how good these AI chatbots really are. ☺) They need to be told what to do in detail, step by step. That's the *algorithm*.

For this task, there can be many different ways to "generate a horoscope". But, we will just use one of the simplest methods, namely, a table lookup. We will create a map with the twelve signs as the keys and their corresponding horoscopes as the values. Then, when we need to generate a horoscope for a given sign, we can just look it up in the map. It is not completely realistic, but this will do for our purposes.

We will see, in more detail, how we go about doing this in the next section.

7.3. Horoscope Function

In practice, whether you really believe in astrology or not, the horoscope is based on the positions of the stars and the planets in the sky. But, we will just hard-code horoscopes in a single map. Here's an example:

```
local message = {
```

```
  aries = "Aries, you are very smart!",
  taurus = "Taurus, you are very
intelligent!",
  gemini = "Gemini, you are very witty!",
  cancer = "Cancer, you are very creative!",
  leo = "Leo, you are very astute!",
  virgo = "Virgo, you are very clever!",
  libra = "Libra, you are very shrewed!",
  scorpio = "Scorpio, you are very wise!",
  sagittarius = "Sagittarius, you are
ingenious!",
  capricorn = "Capricorn, you are very
knowledgeable",
  aquarius = "Aquarius, you are
enlightened!",
  pisces = "Pisces, you are brilliant!",
}
```

Then, we can (almost trivially) implement our horoscope function, which takes a sign as an argument and returns the corresponding horoscope. For instance,

```
function horoscope(sign)
  return message[sign]
end
```

That's it. Everything else is, as stated, about input and output.

7.3.1. Unit testing

The unit testing is another big concept in software engineering which you will have to learn over time. But, let's see briefly what unit testing is, in the present context.

First, we implemented this `horoscope` function with certain desired functionalities in mind. Although we did not nowhere explicitly mention it, if you understand the above implementation, this function *is supposed to work in a certain way*.

For example, when `horoscope` is called with an argument, `"aries"`, it should return a string, `"Aries, you are very smart!"`. Again, the key word here is *it should.* That's the *expected* behavior. But, in practice, programs may not work as expected. That is why we need "testing", e.g., to verify that the program works as expected.

How exactly we do testing is not that important at this point.

The unit testing is a special kind of testing. If you have paid attention, even though it is a small program, we deliberately divided it into two parts, the core part and the rest (mostly I/O). Now, testing the entire program may be a little bit more complex than testing each individual part. Testing a small part, or a *unit*, is what

we call the unit testing. This is, in a way, a divide-and-conquer strategy (which is also related to the analysis-synthesis method). The "unit" can mean different things in different contexts, but again that is not very important at this point.

In this particular example, we have this core function, separate from the rest of the program, and it can be *more easily* tested by itself. Just for illustration, let's write a simple "test case" for the `horoscope` function.

```
local sign = "leo"
local expected = "Leo, you are very astute!"
local result = horoscope(sign)

if (result == expected) then              ①
  print("Test passed")
else
  print("Test failed: result = " .. result)
end
```

① The symbol `==` is an equality operator. The expression `A == B` evaluates to `true` if `A` is the same as `B`. Otherwise, its value is `false`.

Can you see what's going on here? If not, don't worry. You will have plenty of chance to learn about unit testing, moving forward, if you continue to learn and practice programming.

7.3.2. Error handling

In this particular example, you can write twelve unit test cases that cover all possible input arguments. Or, more precisely, all *valid* input arguments.

What happens if the given argument is not one of the 12 valid signs? For our purposes, that's an error. We cannot provide a horoscope for an invalid astrology sign. *What do we do then?* This is called the error handling. There can be many different ways to handle invalid inputs, for instance. In our Horoscope Teller program, let's just print out an error message to the user if the given input is not valid.

In order to do this, the `horoscope` function needs to return a special value to indicate an invalid input. It so happens that the map access in Lua, e.g., `message[sign]`, returns a special value `nil` if the key `sign` does not exist in the map `message`. `nil` is a special value in Lua that represents an absence of a value. That is, `nil` is a value that is not a real value. ☺ Hence, the simplest way to handle errors in `horoscope` is just do nothing. It will naturally return `nil` if the given argument is invalid. Just for illustration,

```
print(message["moon"])
```

This statement will print out *nil*.

7.4. Horoscope Teller - CLI App

Here's a version of our main CLI program that includes the user input and output handling.

```
function playHoroscope()
  io.write("What is your sign? ")
  local sign = io.read()
  local msg = horoscope(sign)
  if (msg == nil) then
    io.write("Invalid sign: " .. sign ..
"\n")
  else
    io.write(msg .. "\n")
  end
end

playHoroscope()
```

7.4.1. Let's play!

Combine all the pieces together and make a complete program, as an exercise. Then, try to run the program.

Here's one example run.

```
What is your sign? cancer
Cancer, you are very creative!
```

Exercises

We learned in this lesson

- Basic input output, and
- How to write a simple CLI program.

In particular, we learned how to read the user input and how to write properly to the standard out, without relying on the debugging function, `print`. Also, we learned how to use the map data structure as a lookup table, among other things. Let's practice some of these concepts in our last exercise session.

Ex 1. Hello, what's your name?

Write a CLI program that does the following:

- Ask the user his/her name,
- Read the name, and
- Say, "Hello, <the user name>!".

Ex 2. The length of the user input

In the "I/O practice" section, we implemented a simple "echo" program. Modify this program so that, instead of echoing back the user's input, print out one of the following messages.

- If the length of the user input is longer than 42 characters, then write "Life is short. Use a shorter message." to the console.
- If it is shorter than 7 characters, then write "You are not very lucky!"
- Otherwise, write "You are just average."

Ex 3. Error handling

Add an error handling to the day of week function, `weekday(year, month, day)`, from the previous lesson. First, what kind of errors do you anticipate? Is a month `23` a valid input, for instance? Remember, there is no right and wrong answer.

Ex 4. Day of the week CLI program

Add input output handling to the day of week program from the previous lesson, and make it a complete CLI program.

Ex 5. A better horoscope app?

This is an open-ended question. Can you create a horoscope app that returns different horoscopes for different weeks? One last comment: All exercises are optional. ☺

Closing Remarks

> It is not in doing what you like, but in liking what you do that is the secret of happiness.
>
> — J.M. Barrie

We have covered very little in this book in terms of Lua syntax, and programming in general. You might say, we barely scratched the surface. But, then again, we have covered so much in this book.

The astute readers might have noticed, but the goal of this book was not to teach you all the details of programming or to go through the whole laundry list of programming language features. But, the goal of this

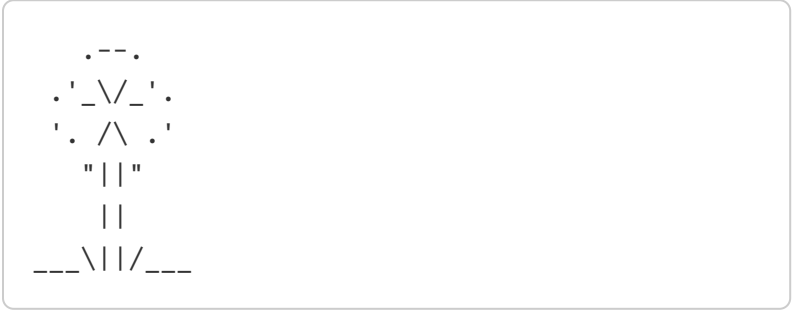
book was to teach you the fundamentals of programming. And more importantly, to teach you how to learn, or how to teach yourself, when it comes to programming.

We went through some big concepts (and, big words ☺) in this book, both technical and non-technical, so that you can learn much easier moving forward.

As the saying goes, "Give a man a fish, and you feed him for a day. Teach a man to fish, and you feed him for a lifetime", and teaching you to fish was the goal of this book. Whether we have succeeded or not, only *you* can tell.

For now, with the same spirit, try to teach yourself to fish rather than just focusing on details while learning programming, or while learning anything. That's a lifelong process. And, most importantly,

Have fun!



Credits

Images

All drawings used in this book are taken from undraw.co, an amazing service with an amazing open source license. Many thanks to the creator of the site: twitter.com/ninaLimpi!

Icons

All emoji icons used in this book are from fontawesome.com. Fontawesome is a very popular tool, probably used by almost everyone who does Web or mobile programming.

Typesetting

Here's another absolutely fantastic software, asciidoctor.org, which is used to create an ebook as well as paperback versions of this book. AsciiDoc [https://asciidoc.org] is like Markdown on steroid. You can follow them on Twitter: twitter.com/asciidoctor.

Other Resources

The author has relied on many resources on the Web in writing this book. If the book includes any material from these resources, then the copyright of those content belong to the respective owners.

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: www.instagram.com/codeandtips/
- TikTok: tiktok.com/@codeandtips
- Twitter: twitter.com/codeandtips
- YouTube: www.youtube.com/@codeandtips
- Reddit: www.reddit.com/r/codeandtips/

Other Programming Books by the Author

- The Art of Go - Basics: Introduction to Programming in Golang [https://www.amazon.com/dp/B08WYNG6YP]
- The Art of C# - Basics: Introduction to Programming in Modern C# [https://www.amazon.com/dp/B08X2SCG2Y]

Coding Lessons for Beginners

Learn programming, and new programming languages, using these books. The general target audience is from absolute beginners to programmers with a few years of coding experience.

- Learn Coding with Lua: A Slow and Gentle Introduction to Basic Programming for Non-Programmers [https://www.amazon.com/dp/B0BF19Q3DV/]
- Python for Serious Beginners: A Practical Introduction to Modern Python with Simple Hands-on Projects [https://www.amazon.com/dp/B09M7P9WCZ/]
- Python for Passionate Beginners: A Practical Guide to Programming in Modern Python with Fun Hands-on Projects [https://www.amazon.com/dp/B0BG9X5L8V/]

Programming Language References

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.

- Golang [https://www.amazon.com/dp/B09V5QXTCC/]
- Modern C# [https://www.amazon.com/dp/B0B57PXLFC/]
- Python [https://www.amazon.com/dp/B0B2QJD6P8/]
- Typescript [https://www.amazon.com/dp/B0B54537JK/]
- Rust [https://www.amazon.com/dp/B09Y74PH2B/]
- C++20 [https://www.amazon.com/dp/B0B5YLXLB3/]
- Modern Java [https://www.amazon.com/dp/B0B75PCHW2/]
- Julia [https://www.amazon.com/dp/B0B6PZ2BCJ/]
- Javascript [https://www.amazon.com/dp/B0B75RZLRB/]
- Haskell [https://www.amazon.com/dp/B09X8PLG9P/]
- Scala 3 [https://www.amazon.com/dp/B0B95Y6584/]
- Lua [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 and design patterns to machine learning. You can also find some sample code in the public code repositories on GitLab.

- www.codeandtips.com
- gitlab.com/codeandtips

Mailing List

Join our mailing list, join@codingbookspress.com, to receive coding tips and other news from **Coding Books**. If we find any significant errors in the book, then we will send you an updated version of the book (in PDF).

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 comments regarding the book, then please let us know. Although we cannot answer all the questions and emails, we will try our best to resolve the issues.

- feedback@codingbookspress.com

Revision 1.0.0, 2023-04-05

THANK YOU!!

Thanks for signing up to be part of our advance review team. It means so much to us. This effort wouldn't be possible without you.

-Harry