
Lua

Booklet 1 of 2: Foundations to tables

Chapters 1 to 4

Fall 2026

Contents

Chapter 1: Foundations	1
Lua Basics	1
Setting Up the Environment	3
Basic Data Types	5
Chapter 2: Variables and Control Flow	9
Variables in Lua	9
Control Flow Statements	12
Conditional Statements	15
Chapter 3: Functions	19
Function Basics	19
Function Arguments	22
Return Values	25
Chapter 4: Tables	29
Introduction to Tables	29
Table Operations	32
Tables as Arrays and Dictionaries	36

Chapter 1: Foundations

Lua Basics

Lecture

Why this comes first

Most introductions to a language start with syntax. This one starts with the design brief, because Lua's syntax is unusually easy and its design choices are unusually consequential. Almost everything that will later feel surprising, and almost everything that makes Lua worth learning, follows from a single decision made at the start: the language had to be small enough to embed inside somebody else's program.

The official description is one sentence, and every word in it is load-bearing:

"Lua is a powerful, efficient, lightweight, embeddable scripting language." (lua.org, About)

By the end you should be able to:

- Say what Lua is for, in terms of the constraint that shaped it.
- Give concrete evidence for the smallness claim rather than repeating the word.
- Explain what "embeddable" means as a technical property, not a marketing one.
- Predict, from the design brief, why the language has one data structure and no built-in class system.
- Say which Lua version this book teaches, and why the version is worth checking.

1. Smallness, with numbers

Every tutorial calls Lua small. That is worth converting into something you can check, because the number is the argument.

On 64-bit Linux, the Lua interpreter built with all the standard libraries takes **293K**, and the Lua library itself takes **484K** (lua.org, About, which gives these figures for the current release, Lua 5.5.1, as checked in October 2026). That is the whole language: parser, virtual machine, garbage collector and standard library.

The portability claim rests on the same discipline. Lua "is distributed in a small package and builds out of the box on all platforms that have a standard C compiler" (lua.org, About), which is why it turns up on Unix, Windows, phones and embedded devices without a per-platform port.

Hold onto the size figure. It is the reason for design decisions that would look like omissions in a larger language.

2. What "embeddable" actually means

Embeddable is not a synonym for "small". It is a statement about who is in charge.

In a normal program, your language runs the show and calls out to libraries. In an embedded scripting language, a **host** program written in another language (usually C) is in charge, and it creates a Lua state, loads a piece of Lua code, and calls it. Lua is a guest.

The official framing is that Lua "is a fast language engine with a small footprint that you can embed easily into your application", with "a simple and well documented API that allows strong integration with code written in other languages" (lua.org, About).

Two consequences worth carrying forward:

- **The host decides what Lua can see.** A Lua script in a host application can only touch what the host chose to expose. That is what makes it safe to let users script your application.

- **The language cannot afford opinions.** A language meant to be dropped into any host cannot assume a threading model, a filesystem, an object system, or a garbage collection strategy that the host might disagree with.

That second point is the key to the rest of this book. Chapter 8 returns to embedding from the host's side, once you have the language to make sense of it.

3. What the design brief predicts

Before you have seen any syntax, you can predict several of Lua's features by asking "what would a language do if it had to fit in 293K and offend no host?"

- **One compound data structure, not five.** Shipping arrays, lists, records, dictionaries and sets separately costs code size and forces the host to carry all of them. Lua ships exactly one, the *table*, and lets it play every role. This is Chapter 4, and it is the single most important idea in the language.
- **No class keyword.** An object system is an opinion. Lua supplies a lower level mechanism (*metatables*, Chapter 5) from which you can build the object system you want, and declines to pick one for you.
- **A small standard library.** Chapter 6 tours it.
- **Cooperative rather than pre-emptive concurrency.** Pre-emption requires assumptions about threads that a guest cannot make. Lua provides *coroutines* (Chapter 7), which yield control explicitly.

None of these are quirks to memorise. They are one decision, seen from four angles. The Evolution of Lua, the language authors' own history paper, is the place to read the reasoning first-hand.

4. Where you will actually meet Lua

The manual describes Lua as an extension language that works embedded in a host program (Lua 5.4 Reference Manual, section 1). The history paper lists the kinds of application that use it, among them game development, robotics, image processing, extensible text editors, Ethernet switches and web development (Ierusalimsky, de Figueiredo and Celes, HOPL III, section 1).

That has a practical consequence. **Embedded Lua runs inside the host's rules:** the host decides which globals, callbacks and libraries a script can use. Learning the language is necessary and not sufficient; reading a host's API is the other half.

5. Which Lua this book teaches

This is one piece of administrative detail that is worth settling before the first example.

- **Lua 5.4** is the version this book teaches. Its reference manual is the authority for anything where the exact rule matters.
- **The first edition** of *Programming in Lua* is freely available online. Each of its pages carries this note: "This first edition was written for Lua 5.0. While still largely relevant for later versions, there are some differences." (lua.org/pil) We use it for teaching order and explanation, and we check every precise claim against the 5.4 manual.

Other Lua releases differ from 5.4 in places, so an example that works here may behave differently on another release. The manual has a section on incompatibilities, section 8 "Incompatibilities with the Previous Version", but it lists only the changes from Lua 5.3 to Lua 5.4 (Lua 5.4 Reference Manual, contents). For any other release, the place to look is that release's own manual.

6. What to take away

Lua is a small, fast language designed to be a guest inside a larger program. The size is real and measurable. The guest status explains the feature set far better than any list of features would: one data structure, no class system, a

small library, cooperative concurrency. Everything in the next seven chapters is either a piece of that language or a consequence of that constraint.

Key Terms

- **Embeddable:** designed to run as a guest inside a host program written in another language, where the host creates the Lua state, loads the code and calls it.
- **Host program:** the application that embeds Lua and decides what the script is allowed to see. This is what makes user scripting safe.
- **Lua footprint:** on 64-bit Linux the interpreter with all standard libraries is 293K and the Lua library is 484K, according to the lua.org About page for Lua 5.5.1, which is the concrete evidence behind the word lightweight.
- **Scripting language:** a language intended to drive or extend an existing application rather than to be the whole of it.
- **Lua 5.4:** the version of the language this book teaches. The reference manual for it is the authority on exact semantics.
- **Programming in Lua, first edition:** the freely available edition of the book, quoted here for its explanations. Its own pages say it was written for Lua 5.0 and is still largely relevant for later versions, with some differences, so every precise claim is checked against the 5.4 manual.

Summary

Lua is a small, fast scripting language designed to be a guest inside a larger program, and the smallness is measurable rather than rhetorical. That single constraint predicts the feature set: one compound data structure instead of five, no built-in class system, a small standard library, and cooperative coroutines rather than pre-emptive threads. This book teaches Lua 5.4. Other releases differ in places, so the version is the first thing to check when an example behaves differently on your machine.

Setting Up the Environment

Lecture

Why this comes second

You cannot learn a language by reading about it. This lecture exists to get a Lua interpreter onto your machine and a program running inside it, so that every claim in "Basic Data Types" and afterwards is something you can check for yourself in under ten seconds.

By the end you should be able to:

- Install Lua by whichever of the three routes suits your machine.
- Write a program in a file and run it with the standalone interpreter.
- Say which Lua you have installed, and why that question is not pedantic.
- Know where the definitive reference for the interpreter's behaviour lives.

1. Getting an interpreter

Lua's own getting-started page describes three ways to get Lua (lua.org, Getting started):

- **Pre-compiled binaries.** The page says pre-compiled Lua libraries and executables are available at LuaBinaries.
- **A package manager.** The page says that on Linux or macOS, Lua is either already installed or there is a Lua package for it.

- **Building from source.** The page calls Lua "very easy to build and install" and shows a terminal session that downloads a release and builds it with make. The About page says Lua builds out of the box on all platforms that have a standard C compiler.

Which you choose is a matter of what your machine already has. The page recommends the latest release, which it names as 5.5.1 when checked in October 2026. This book teaches 5.4, so check which version you ended up with (section 4) and expect differences if it is a later release.

2. Writing and running your first program

Once an interpreter exists, running a program is one command. *Programming in Lua* states it plainly:

"If you are using the stand-alone Lua interpreter, all you have to do to run your first program is to call the interpreter (usually named `lua`) with the name of the text file that contains your program." (Ierusalimsky, *Programming in Lua* 1st ed., ch. 1)

Which in practice looks like this:

```
LUA
prompt> lua hello.lua
```

So the whole workflow is: put your code in a text file, save it, and hand the filename to `lua`. Create `hello.lua` containing a single line:

```
LUA
-- Two hyphens start a comment that runs to the end of the line.
-- print is available without importing anything.
print("Hello from Lua")
```

Then run it with `lua hello.lua`. If you see the greeting, your environment is finished and you are ready for "Basic Data Types".

If you see an error instead, read it before changing anything. Lua's error messages name the file and the line, and getting into the habit of reading them now will save far more time later than it costs today.

3. Choosing an editor

Lua's getting-started page says to use your favourite text editor to write your Lua programs and to save them as plain text, and it names ZeroBrane Studio as an IDE to try if you want one (lua.org, Getting started).

Any editor with syntax highlighting is enough for this book. Nothing in this book requires a particular tool.

4. Which Lua did you just install?

Ask this now rather than in Chapter 5. As "Lua Basics" set out, this book teaches Lua 5.4, and other releases differ from it in places. You can ask the interpreter which version it is:

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

A package manager may give you a different release, and the version you have determines which examples run unmodified. The global `_VERSION` holds the running version (Lua 5.4 Reference Manual, §6.1). When something in a later lecture does not behave as described, the version is the first thing to check, and the manual's section 8, "Incompatibilities with the Previous Version", records the changes from Lua 5.3 to Lua 5.4 (Lua 5.4 Reference Manual, contents).

5. What this lecture deliberately does not cover

Two things you would reasonably expect in a setup lecture are missing, and the honest reason is that this book does not write anything it has not read.

- **The interactive interpreter (the REPL) and the command-line flags.** These are real and useful, and they are defined in section 7 of the reference manual, "Lua Standalone" (Lua 5.4 Reference Manual, contents). That section is named here as the place to look, and nothing about its contents is paraphrased. Read it directly; it is short.
- **Embedding Lua in a C or C++ host.** The getting-started page covers this, and so does Chapter 8 of this book, at the point where you have the language needed to make sense of it.

If you want the REPL before Chapter 8, the manual section above is a five-minute read and everything in it will make sense already.

6. What to take away

Install by whichever of the three routes fits your machine, put code in a file, and run it with `lua yourfile.lua`.

Check which version you got, because that answer will explain a surprise later. The reference manual is the authority on the interpreter itself, and it is worth knowing where that authority lives before you need it.

Key Terms

- **Standalone interpreter:** the lua command-line program that runs a Lua file directly, invoked as `lua yourfile.lua`.
- **Lua Standalone:** section 7 of the Lua 5.4 Reference Manual, the definitive reference for the interpreter's command-line options and behaviour.
- **Incompatibilities with the Previous Version:** section 8 of the Lua 5.4 Reference Manual, which lists the incompatibilities between Lua 5.3 and Lua 5.4. Worth checking when an example does not behave as written.
- **Comment:** text ignored by the interpreter. A line comment starts with two hyphens and runs to the end of the line.
- **print:** a function available without importing anything, which writes its arguments to standard output. The fastest way to check a claim about the language.

Summary

Install Lua by pre-compiled binary, package manager or source build, whichever suits the machine, then put code in a text file and run it with `lua yourfile.lua`. Check which version you were given, because that answer explains surprises later. The reference manual's standalone section is the authority on the interpreter itself, and it is worth knowing where that authority lives before you need it.

Basic Data Types

Lecture

Why this comes third

You now know what Lua is for and you have an interpreter to test claims in. This lecture gives you the vocabulary of values, which is the foundation every later chapter builds on. Two of the eight types introduced here get an entire chapter to themselves later: tables in Chapter 4, and threads (coroutines) in Chapter 7.

By the end you should be able to:

- Name all eight basic types and predict what `type()` returns for a value.
- State exactly which values are false in a condition, and defend the answer.

- Explain why `nil` is both "no value" and the way you delete something.
- Say how integers and floats differ inside the number type, and what each one prints.

1. There are exactly eight

The manual is precise about this, in section 2.1 "Values and Types". The eight basic types are:

nil, boolean, number, string, function, userdata, thread, table

Two of those will look unfamiliar and are worth naming now even though they come later:

- **userdata** holds raw memory owned by the host program. This is the type that makes embedding work: when a C host hands Lua an object, that object arrives as userdata. The manual says userdata values "cannot be created or modified in Lua, only through the C API" (§2.1).
- **thread** is a coroutine, not an operating-system thread. "Lua Basics" predicted this from the design brief: a guest language cannot assume a threading model, so Lua supplies cooperative coroutines instead. Chapter 7.

Note what is *absent*. There is no separate array type, no list, no dictionary, no set, no class. "Lua Basics" predicted the first four of those from the design brief, and Chapter 4 shows how one type covers all of them.

You can check any value's type yourself, which is the habit worth forming today:

```
LUA
-- type() returns the type NAME as a string, which is itself a string value.
print(type(nil))      --> nil
print(type(true))     --> boolean
print(type(42))        --> number
print(type("hello"))  --> string
print(type(print))     --> function
print(type({}))        --> table
```

2. The falsity rule, stated exactly

This is the single rule in this lecture most likely to cost you time if you carry an assumption in from another language, so here it is verbatim from the manual:

"Both **nil** and **false** make a condition false; they are collectively called *false values*. Any other value makes a condition true." (Lua 5.4 Reference Manual, §2.1)

Read the second sentence again, because it is the surprising half. **Any other value**. That includes `0`. That includes the empty string. Both are true in a Lua condition.

```
LUA
-- In C, Python and JavaScript, at least one of these is falsy.
-- In Lua, both branches below print. This is not a quirk; it is the rule above.
if 0 then print("zero is true in Lua") end
if "" then print("empty string is true too") end
```

If you are coming from Python or JavaScript, this is the assumption to unlearn first. A guard written as `if count` `then` is testing whether `count` exists, not whether it is non-zero, and those are different questions.

3. nil: absence, and deletion

`nil` is the type with exactly one value, also written `nil`, and it means "no value". A variable you have never assigned to is `nil` rather than an error.

That much is ordinary. The part that is specific to Lua is what `nil` does when you assign it *into* a structure: assigning `nil` removes the entry. There is no separate delete operation, because assignment of `nil` is the delete operation.

This is a genuine two-for-one in the language design and it is consistent with the smallness argument from "Lua Basics": one concept covers "absent" and "remove", so the language does not need a second one. It is also why "the key is present but its value is nil" is not a state Lua can be in, which will matter when you meet tables in Chapter 4.

4. Booleans, and why the rule in section 2 is not redundant

The boolean type has two values, `true` and `false`.

It is worth being clear about why booleans and the falsity rule are separate ideas. `false` is a *value* of the boolean type. A *false value* is anything that makes a condition fail, which is `false` and `nil` and nothing else. So the set of false values is larger than the boolean type, and smaller than most languages' falsy sets. Both halves of that sentence catch people out.

5. Numbers: integers and floats

This is the type with one feature that changes what your programs print, so it is worth seeing directly.

Lua 5.4 distinguishes between integers and floats *within* the single number type. The manual says so in the same section as the eight types: "The type number represents both integer numbers and real (floating-point) numbers, using two subtypes: integer and float." (Lua 5.4 Reference Manual, §2.1)

The practical consequence is in what gets printed. An integer prints without a decimal point and a float prints with one, and the two compare equal with `==` when they hold the same number. Division with `/` always produces a float, so even a whole-number quotient prints with a decimal point.

Run this yourself rather than taking anyone's word for it:

```
LUA
-- Integers and floats are two subtypes of the one number type.
print(1)                --> 1
print(1.0)              --> 1.0
print(1 == 1.0)         --> true
print(math.type(1), math.type(1.0)) --> integer float
print(10 / 2)           --> 5.0
print(10 // 2)          --> 5
```

`math.type` reports which subtype a number has. The line `10 / 2` gives `5.0` because `/` always produces a float, while `10 // 2` gives the integer `5` because both operands are integers.

6. Strings and functions, briefly

Strings are sequences of bytes and are immutable: every operation that looks like it modifies a string returns a new one. Chapter 6 covers the string library, including Lua patterns, which are its own matching notation and are not regular expressions.

Functions are a basic type, which is the same as saying functions are values. They can be stored in variables, passed as arguments and returned. This is the foundation of Chapter 3, and it is also why the design brief could get away without a class system: if functions are values, you can build the abstractions yourself.

7. What to take away

Eight types, two of which (userdata, thread) exist because Lua is a guest in someone else's program. Only `nil` and `false` are false, so `0` and `""` are true. `nil` means absent and assigning it deletes. And the number type has two subtypes, integer and float, which print differently.

Next chapter: variables and scope, then the control structures that use the falsity rule you just learned.

Key Terms

- **Basic types:** Lua has exactly eight, namely nil, boolean, number, string, function, userdata, thread and table.
- **nil:** the type with a single value meaning no value. An unassigned variable is nil rather than an error, and assigning nil into a structure removes the entry.
- **False value:** nil or false, and nothing else. The Lua 5.4 manual states that both nil and false make a condition false and any other value makes it true, so 0 and the empty string are both true.
- **userdata:** a type holding raw memory owned by the host program. It is how a C host hands its own objects to Lua, and it cannot be created from Lua code, only through the C API.
- **thread:** the type of a coroutine, which is cooperative and not an operating-system thread. Covered in Chapter 7.
- **type():** a function returning a value's type as a string, and the fastest way to check any claim in this lecture against your own interpreter.
- **Integer subtype:** the distinction between integers and floats inside the single number type. The integer 1 prints as 1 and the float 1.0 prints as 1.0, and the two compare equal.
- **math.type:** a function that reports whether a number is an integer or a float.
- **Immutable string:** a string that cannot be changed in place, so every operation that looks like a modification returns a new string.

Summary

Lua has eight basic types, two of which exist because Lua is a guest inside another program. Only nil and false are false in a condition, which means zero and the empty string are true and any guard borrowed from Python or JavaScript needs rereading. nil means absent and assigning it is how you delete. The number type has integer and float subtypes that print differently, so a division such as `10 / 2` prints 5.0.

Chapter 2: Variables and Control Flow

Variables in Lua

Lecture

Why this comes first

Chapter 1 covered what values are. This chapter covers where they live and how control moves between them, and it starts with variables because Lua's answer to "what is a global variable" is unusually specific and unusually useful.

Most languages treat globals as a separate mechanism you are told to avoid. Lua tells you exactly what one is, in a single sentence, and once you have that sentence three separate facts collapse into one explanation.

By the end you should be able to:

- State the scope rule for a local variable precisely.
- Explain what a global variable actually is in Lua, mechanically.
- Say why locals are faster than globals, from that mechanism rather than as folklore.
- Explain why a mistyped variable name is a silent `nil` rather than an error.
- Use multiple assignment without being surprised by the extra or missing values.

1. local, and the exact scope rule

The manual states the rule precisely, and the precision is worth having:

"The scope of a local variable begins at the first statement after its declaration and lasts until the last non-void statement of the innermost block that includes the declaration." (Lua 5.4 Reference Manual, §3.5 Visibility Rules)

Two consequences fall straight out of that wording.

The scope begins AFTER the declaration. So in `local x = x`, the `x` on the right is the *outer* one, because the new local does not exist yet at that point. That is a genuinely useful idiom for capturing an outer value, and it looks like a typo until you know the rule.

The scope ends with the innermost block. A block, per *Programming in Lua*, is "the body of a control structure, the body of a function, or a chunk (the file or string with the code where the variable is declared)" (PiL 1st ed., §4.2).

```
LUA
local x = 1
do
  local x = 2      -- a DIFFERENT variable, shadowing the outer one
  print(x)         --> 2
end
print(x)           --> 1, the outer x was never touched
```

The manual adds one more sentence that will matter in Chapter 3: "each execution of a **local** statement defines new local variables" (§3.5). Each pass through a loop therefore creates a *fresh* variable rather than reusing one. That is what makes closures created inside a loop capture separate values, as the manual's own example in section 3.5 shows.

2. What a global actually is

Here is the sentence that does the work:

"any reference to a free name (that is, a name not bound to any declaration) var is syntactically translated to `_ENV.var`." (Lua 5.4 Reference Manual, §2.2 Environments)

A *free name* is a name not bound to any declaration. So when you write `print`, the name is translated to `_ENV.print`, which is an ordinary table index. The manual adds that every chunk is compiled in the scope of an external local variable named `_ENV`, so `_ENV` itself is never a free name, and that `_ENV` is "a completely regular name" (§2.2).

The manual confirms the consequence: the global environment is a special table, and "by default, free names in Lua code refer to entries in the global environment and, therefore, they are also called *global variables*" (§2.2).

A global variable is not a separate mechanism. It is table indexing on `_ENV`, and by default `_ENV` holds the global environment. That is the whole thing, and it explains three things at once.

3. The three things it explains

Why locals are faster. *Programming in Lua* gives the advice and the reason: "It is good programming style to use local variables whenever possible. Local variables help you avoid cluttering the global environment with unnecessary names. Moreover, the access to local variables is faster than to global ones" (PiL 1st ed., §4.2). Now you know *why* rather than just *that*: a local is a variable the compiler resolves ahead of time, and a global is a table lookup done at run time.

Why a typo is silent. Indexing a table with a key that is not present returns `nil` (Chapter 1). Since a global read is a table index, a misspelled global is not an error. It is `nil`.

```
LUA
-- No error. This creates a new global named countr. The variable we meant,
-- counter, is still nil, so a later counter + 1 fails somewhere else.
local total = 0
countr = total + 1    -- meant `counter`
```

This is the strongest practical argument for `local`. Lua does not flag this, because an undeclared name is just a free name, and section 2 showed that a free name is a global read. Declaring variables with `local` by default keeps the real globals few, so a stray one stands out.

Why sandboxing is possible. If globals are entries in a table, then swapping the table changes what a chunk can see. That is the mechanism behind letting a host application run untrusted scripts, which is the Chapter 8 subject and the reason the "Lua Basics" lecture called Lua a guest.

4. `_G`, and reading the environment as data

Because the environment is an ordinary table, you can inspect it.

```
LUA
-- These are the same thing, by the translation rule in section 2.2 of the manual.
x = 10
print(x)          --> 10
print(_G.x)       --> 10

-- Which means you can ask questions you could not ask in most languages.
for name, value in pairs(_G) do
  print(name, type(value))
end
```

Useful for exploring, and a reasonable way to see how small the standard library really is. Not something to build on: writing through `_G` is the same as creating globals, with the same problems.

5. Declare late, and initialise

PiL's second piece of advice is about placement rather than scope: "By declaring a variable only when you need it, you seldom need to declare it without an initial value... Moreover, you shorten the scope of the variable, which increases readability" (PiL 1st ed., §4.2).

This lands differently in Lua than in a language with block-scoped declarations everywhere, because an uninitialised local is `nil`, and `nil` is a value that flows onward silently rather than failing at the point of the mistake. Declaring at the point of first use removes the window in which that can happen.

6. Multiple assignment

Lua assigns lists to lists, and the rule for a length mismatch is mechanical rather than an error.

```
LUA
local a, b = 1, 2      -- the ordinary case
a, b = b, a            -- swap, with no temporary, evaluated before assigning

local x, y, z = 1, 2    -- extra names get nil:      x=1  y=2  z=nil
local p, q = 1, 2, 3    -- extra values are dropped: p=1  q=2

-- The idiomatic use: a function returning several values.
local ok, err = pcall(risky)
```

The pattern to internalise is that **too few values give `nil` and too many are discarded, silently in both directions**. Combined with the falsity rule from Chapter 1, that means a missing return value does not announce itself; it arrives as `nil` and behaves like a legitimate "absent". Chapter 3 goes into this properly when it covers functions returning multiple values.

7. What to take away

A local's scope starts after its declaration and ends with its innermost block, and each execution of a `local` statement makes a new variable. A global is not a separate mechanism at all: a free name is rewritten to `_ENV.var` and looked up in a table, which is simultaneously why locals are faster, why typos are silent, and why sandboxing works. Use `local` by default, declare at first use, and expect multiple assignment to pad with `nil` and drop the excess without complaint.

Next: the control structures that use Chapter 1's falsity rule.

Key Terms

- **Local scope rule:** the scope of a local begins at the first statement AFTER its declaration and lasts to the last non-void statement of the innermost block containing it.
- **Block:** the body of a control structure, the body of a function, or a chunk, meaning the file or string in which the variable is declared.
- **Free name:** a name not bound to any declaration. The manual says any reference to one is syntactically translated to `_ENV.var`, which makes it a table index.
- **_ENV:** a regular name for the external local variable that every chunk is compiled in the scope of. Free names are looked up in the table it holds, which by default is the global environment. Swapping that table is what makes sandboxing a script possible.
- **_G:** a global variable initialized with the global environment table, so the environment can be read and iterated like any other table.
- **Why locals are faster:** a local is a compiler-resolved slot, while a global is a runtime lookup in a table. The speed difference is a consequence of the `_ENV` translation, not a separate fact.

- **Silent typo:** a misspelled global is not an error, because indexing a table with an absent key returns nil. It fails later, somewhere else.
- **local x = x:** a legitimate idiom capturing the OUTER x, because a local's scope begins only after its declaration, so the right-hand x is still the outer one.
- **Fresh locals per execution:** each execution of a local statement defines new local variables, which is what lets closures made in a loop capture separate values.
- **Multiple assignment:** values are matched to names positionally; missing values become nil and extra values are discarded, silently in both directions.

Summary

A local's scope starts after its declaration and ends with its innermost block, and each execution of a local statement makes a new variable. A global is not a separate mechanism: a free name is rewritten to `_ENV.var` and looked up in a table, which simultaneously explains why locals are faster, why typos are silent, and why sandboxing works. Use local by default, declare at first use, and expect multiple assignment to pad with nil and drop the excess without complaint.

Control Flow Statements

Lecture

Why this comes second

Lua's control structures are, in *Programming in Lua's* own words, "a small and conventional set... with **if** for conditional and **while**, **repeat**, and **for** for iteration" (PiL 1st ed., §4.3).

Small and conventional is accurate, and it means this lecture can do something unusual: cover the complete set, not a selection. The manual's §3.3.4 and §3.3.5 enumerate every control form the language has, which lets us be certain about what is present *and* about what is absent. One notable thing is absent, and the second half of this lecture is about living without it.

By the end you should be able to:

- Write all four loop forms and choose between them.
- Explain what the generic `for` actually does, and what stops it.
- Say why `repeat ... until` can see variables declared in its own body.
- Handle the absence of `continue`, using `goto`.

1. The complete set

Per §3.3.4 Control Structures and §3.3.5 For Statement, Lua has: `while`, `repeat-until`, `if-then-else` with optional `elseif` branches, `for` in two varieties, `goto` with labels, and `break` to terminate a loop. `return` exits a function.

That is all of it. Compare the list you would write for almost any other language and notice what is not there: no `switch`, no `do-while` in that spelling, no `foreach` keyword, no `continue`.

2. while and repeat until

```
LUA
-- while: test BEFORE the body, so the body may run zero times.
local i = 1
while i <= 5 do
  print(i)
  i = i + 1
end
```

```
-- repeat until: test AFTER the body, so the body always runs at least once.
-- Note the inverted sense. `until` continues while the condition is FALSE.
local n = 0
repeat
  n = n + 1
until n >= 5
```

The `until` sense is the trap. In a `while` you write the condition for *continuing*; in a `repeat` you write the condition for *stopping*. Reading a `repeat` loop as though it were a `while` gets the logic exactly backwards.

A genuine oddity worth knowing. The `until` condition can refer to locals declared inside the loop body. The manual says so directly: "In the repeat-until loop, the inner block does not end at the until keyword, but only after the condition. So, the condition can refer to local variables declared inside the loop block." (§3.3.4) That is a real difference from `while`, where the condition sits outside the body, and a real convenience:

```
LUA
repeat
  local line = io.read()
until line == nil or line == "quit" -- `line` is still in scope here
```

3. The numerical for

The numerical `for` repeats a block while a control variable moves through an arithmetic sequence, with an initial value, a limit, and an optional step (§3.3.5).

```
LUA
for i = 1, 5 do print(i) end      -- 1 2 3 4 5, limit INCLUSIVE
for i = 1, 10, 2 do print(i) end -- 1 3 5 7 9
for i = 5, 1, -1 do print(i) end -- counts down
```

Two things that catch people arriving from other languages:

- **The limit is inclusive.** `for i = 1, #t` covers the whole of a 1-based array with no off-by-one adjustment. This is Lua being internally consistent: 1-based indexing and an inclusive limit fit together.
- **The control variable is local to the loop.** The manual calls it "a new variable local to the loop body" (§3.3.5), so it does not exist after the loop. The manual also says "You should not change the value of the control variable during the loop," so treat it as read-only.

4. The generic for

This is the form you will use most, and it is worth understanding rather than memorising. The manual is exact:

The generic `for` "works over functions, called *iterators*. On each iteration, the iterator function is called to produce a new value, stopping when this new value is `nil`." (Lua 5.4 Reference Manual, §3.3.5)

So a generic `for` is not a special syntax for tables. It is a protocol: call a function repeatedly until it returns `nil`.

```
LUA
-- ipairs: 1, 2, 3, ... stopping at the first nil. Array-like traversal.
for index, value in ipairs(t) do print(index, value) end

-- pairs: every key. Order is NOT specified.
for key, value in pairs(t) do print(key, value) end
```

The `nil-stops rule is load-bearing`, and it is the source of a bug you will eventually hit. Because `ipairs` stops at the first `nil`, a table with a hole traverses only up to the hole:

```
LUA
local t = {1, 2, nil, 4}
for i, v in ipairs(t) do print(i, v) end -- prints 1 and 2, then stops
```

That is not a bug in `ipairs`. It is the documented protocol doing exactly what it says. Chapter 4 returns to this when it covers the length operator, which has the same underlying cause.

Since the protocol is just "a function returning values until nil", you can write your own iterators, and Chapter 7 shows how coroutines make that easy.

5. break, and the missing continue

`break` terminates the innermost loop, and it is in the §3.3.4 list.

`continue` is not. The enumeration in §3.3.4 and §3.3.5 is complete, so this is a fact about the language rather than an omission in this lecture.

The `idiomatic replacement` is `goto` with a label at the end of the loop body:

```
LUA
for i = 1, 10 do
  if i % 2 == 0 then goto continue end
  print(i) -- odd numbers only
  ::continue::
end
```

`goto` is not the only way. The other form inverts the condition instead, so the rest of the body runs only when the iteration should continue:

```
LUA
for i = 1, 10 do
  if i % 2 ~= 0 then -- same result, no goto
    print(i)
  end
end
```

6. Choosing between them

Situation	Use
A known count or a numeric range	numerical <code>for</code>
Walking an array-like table in order	generic <code>for</code> with <code>ipairs</code>
Walking every key of a table	generic <code>for</code> with <code>pairs</code>
Looping until a condition changes	<code>while</code>
The body must run at least once	<code>repeat until</code>

Reach for the numerical and generic `for` first. They make the iteration explicit and cannot be broken by forgetting to advance the counter, which is the classic `while` bug.

7. What to take away

The complete set is `while`, `repeat until`, numerical `for`, generic `for`, `goto` and `break`. `repeat` tests after the body and its condition can see the body's locals. The generic `for` is a protocol, not a syntax: call an iterator until it

returns `nil`, which is why `ipairs` stops at a hole. There is no `continue`; use `goto ::continue::`, or invert the condition.

Next: the conditionals, and the two operators that do far more work in Lua than their names suggest.

Key Terms

- **The complete control set:** while, repeat-until, if-then-else with elseif, for in two varieties, goto with labels, and break. There is no switch and no continue.
- **while versus repeat:** while tests BEFORE the body so it may run zero times; repeat tests AFTER so the body always runs at least once.
- **until sense inversion:** a while condition says when to CONTINUE, an until condition says when to STOP. Reading a repeat as a while inverts the logic.
- **repeat scope oddity:** an until condition can refer to locals declared inside the loop body, because the local's scope runs to the end of the innermost block and the until is part of it.
- **Numerical for:** repeats while a control variable moves through an arithmetic sequence with an initial value, a limit, and an optional step. The limit is INCLUSIVE.
- **Generic for:** a protocol rather than a syntax. It calls an iterator function repeatedly to produce a new value, stopping when that value is nil.
- **ipairs stops at nil:** because the generic for stops on nil, ipairs traverses only up to the first hole in a table. This is the documented protocol, not a bug.
- **pairs order is unspecified:** pairs visits every key but guarantees no particular order, so any code depending on iteration order is relying on an accident.
- **No continue:** absent from the language. Use a goto with a `::continue::` label, or invert the condition instead.

Summary

The complete set is while, repeat-until, the numerical for, the generic for, goto and break. repeat tests after the body and its condition can see the body's locals. The generic for is a protocol rather than a syntax, calling an iterator until it returns nil, which is exactly why ipairs stops at a hole. There is no continue: use a goto label, or invert the condition.

Conditional Statements

Lecture

Why this comes third

The `if` statement is five minutes of work. The rest of this lecture is about `and` and `or`, which in Lua are not really boolean operators at all, and which do far more work in idiomatic Lua code than their names suggest.

Get the two definitions in section 2 into your head exactly, and two useful Lua idioms stop being things to memorise and become things you can derive, including the one that has a documented failure case.

By the end you should be able to:

- Write conditionals using Chapter 1's falsity rule correctly.
- State exactly what `and` and `or` return.
- Derive the `x = x or default` idiom, and say when it is wrong.
- Derive the `a and b or c` ternary, and say precisely when it breaks.

1. if, and a reminder about truth

The conditional form is `if-then-else` with optional `elseif` branches (§3.3.4). There is no `switch`.

```
LUA
if x > 10 then
  print("big")
elseif x > 5 then
  print("medium")
else
  print("small")
end
```

Everything here rests on Chapter 1's rule, which is worth restating because this is where it starts costing money:

"Both **nil** and **false** make a condition false; they are collectively called *false values*. Any other value makes a condition true." (Lua 5.4 Reference Manual, §2.1)

So `0` is true and `""` is true. A guard written as `if count then` tests whether `count` exists, not whether it is non-zero. If you want non-zero, say `if count > 0 then`. These are different questions and Lua will not confuse them for you.

2. The two definitions everything else follows from

Here they are exactly, and this is the whole lecture:

and "returns its first argument if this value is **false** or **nil**; otherwise, **and** returns its second argument." **or** "returns its first argument if this value is different from **nil** and **false**; otherwise, **or** returns its second argument." (Lua 5.4 Reference Manual, §3.4.5 Logical Operators)

Read what those say. **They return one of their operands**. Not `true`, not `false`, but one of the two things you gave them. Both also short-circuit, so the second operand is not evaluated unless it is needed.

```
LUA
print(1 and 2)      --> 2      (1 is truthy, so the second is returned)
print(nil and 2)    --> nil    (first is false-y, so the FIRST is returned)
print(1 or 2)       --> 1      (first is truthy, so the first is returned)
print(nil or 2)     --> 2      (first is false-y, so the second is returned)
print(false or "hi") --> hi
```

If you expected `true` or `false` from any of those, reread the definitions. Everything below is these two sentences applied.

3. The default-value idiom, derived

```
LUA
local function greet(name)
  name = name or "stranger"
  print("Hello, " .. name)
end
```

Why it works, from the definition: if `name` is nil or false, `or` returns its second argument, which is `"stranger"`. Otherwise it returns `name` itself, and crucially it returns the *value*, not `true`. That is why the idiom can be used for assignment at all.

And here is where it is wrong. Because only `nil` and `false` are false values, the idiom cannot distinguish "not supplied" from "supplied as `false`":

```
LUA
local function configure(enabled)
    enabled = enabled or true    -- BUG: can never be false
    return enabled
end
print(configure(false))        --> true, which is not what the caller asked for
```

For a boolean option, test for `nil` explicitly:

```
LUA
if enabled == nil then enabled = true end
```

Note that `0` and `""` are *not* affected, since both are truthy in Lua. That is the reverse of the trap JavaScript and Python programmers carry in, where `0` would be eaten by the equivalent idiom. In Lua the danger is `false`, and only `false`.

4. The ternary idiom, and exactly when it breaks

Lua has no ternary operator. The conventional replacement is:

```
LUA
local status = (x > 10) and "big" or "small"
```

Derive it. If the condition is true, `and` returns `"big"`; then `"big" or "small"` returns `"big"` because it is truthy. If the condition is false, `and` returns the false condition; then `false or "small"` returns `"small"`. Correct in both cases.

Now the failure. The derivation assumed the middle value is truthy. If it is `false` or `nil`, the `and` yields a false value and the `or` falls through to the third operand regardless of the condition:

```
LUA
local result = true and false or "fallback"
print(result)    --> fallback, NOT false
```

The condition was true and the answer is still the else-branch. This is not a quirk to memorise: it follows directly from the two quoted definitions, which is the point of learning them exactly. When the middle value can be `false` or `nil`, write a real `if`.

5. Guard clauses and short-circuiting

Short-circuiting makes `and` a safe way to chain checks that depend on each other:

```
LUA
-- The index is only attempted if `user` is truthy, so this cannot error
-- on a nil user.
if user and user.name then
    print(user.name)
end

-- Same idea for a value, using the fact that `and` returns its second operand.
local city = user and user.address and user.address.city
```

That last line is Lua's version of optional chaining, built out of nothing but the operator definition. If any link is `nil`, the whole expression is `nil`, and no indexing of a nil value is ever attempted.

6. Comparison operators, briefly

```
LUA
==  ~=      -- equal, NOT equal. It is ~= and not != .
<   >   <=  >=
```

Two notes. `~=` is the inequality operator, and a C-style `!=` is a syntax error. And comparison operators return actual booleans, unlike `and` and `or`, which is why `x > 10` genuinely gives `true` or `false` while `x and y` gives you back one of your operands.

7. What to take away

`if`, `elseif`, `else`, and no `switch`. Only `nil` and `false` are false, so a truthiness guard tests existence rather than magnitude. `and` and `or` return one of their **operands**, not a boolean, and short-circuit. From that single fact you can derive the `x or default` idiom and its `false` blind spot, and the `a and b or c` ternary and its failure when `b` is falsy. Learn the two definitions and you will not need to memorise either idiom.

That closes Chapter 2. Chapter 3 takes functions seriously, and the closures it introduces rest directly on the "Variables in Lua" lecture's scope rule.

Key Terms

- **and**: returns its FIRST argument if that value is false or nil, otherwise it returns its second argument. It short-circuits.
- **or**: returns its FIRST argument if that value is different from nil and false, otherwise it returns its second argument. It short-circuits.
- **Operators return operands**: `and` and `or` hand back one of the two values given to them, never a manufactured true or false. This is why they can be used for assignment.
- **Default-value idiom**: `x = x or default` works because `or` returns `x` itself when `x` is truthy, and the default otherwise.
- **The false blind spot**: `x = x or default` cannot distinguish an absent argument from one explicitly passed as false. Zero and the empty string are unaffected, because both are truthy in Lua.
- **and-or ternary**: `cond and a or b`, which is correct only while `a` is truthy. If `a` is false or nil the `and` yields a false value and the `or` falls through to `b` regardless of the condition.
- **Guard chain**: `user and user.address and user.address.city`, Lua's optional chaining, which yields nil if any link is nil and never indexes a nil value.
- **Not-equal operator**: written `~=` in Lua, not `!=`. A C-style `!=` is a syntax error.
- **Comparison operators return booleans**: unlike `and` and `or`, the relational operators genuinely produce true or false.

Summary

Lua has `if`, `elseif` and `else`, and no `switch`. Only `nil` and `false` are false, so a truthiness guard tests whether something exists rather than whether it is non-zero. The `and` and `or` operators return one of their operands rather than a boolean, and short-circuit. From that one fact both the `x or default` idiom with its false blind spot, and the `and-or` ternary with its failure on a falsy middle value, can be derived rather than memorised.

Chapter 3: Functions

Function Basics

Lecture

Why this comes first

Chapters 1 and 2 covered values, variables and control flow. This chapter is where Lua starts to feel different from most languages you have used, because a function is not a special kind of thing. It is a value, exactly like a number or a string, and everything in this lecture follows from taking that literally.

By the end you should be able to:

- Read all of Lua's function-declaration forms and know they are the same underlying construct.
- Explain why a function can be stored in a variable, passed as an argument, and returned from another function without any special syntax.
- Write an anonymous function and know when you would reach for one.
- Explain what a closure is, in terms of upvalues, and predict what a closure captures.

1. Functions are values, and declaration is just assignment

The manual's function-definition syntax is `functiondef ::= function funcbody`. That is the whole grammar: the keyword `function`, followed by a parameter list and a body. Nothing in that grammar produces a name. The name you are used to seeing comes from a separate, ordinary assignment, and the manual spells out the translation directly (Lua 5.4 Reference Manual, §3.4.11):

"`function f () body end` translates to `f = function () body end`"

That single line is the key to this whole lecture. `function f () ... end` is not a distinct kind of declaration. It is sugar for creating an anonymous function value on the right and assigning it to the name `f` on the left, using the same `=` you already know from Chapter 2. The manual confirms the same pattern for a dotted path: `function t.a.b.c.f () body end` translates to `t.a.b.c.f = function () body end`.

```
LUA
-- These two statements are IDENTICAL, per the manual's own translation rule.
function greet(name)
    return "hello, " .. name
end

greet = function(name)
    return "hello, " .. name
end
```

`local function` is the one form worth pausing on, because it looks like the same sugar but is not quite. `local function f () body end` declares the local `f` *before* defining the function body, specifically so that the function can call itself by name. This matters the moment you write a recursive local function: if the local were declared only after the assignment (as Chapter 2's ordinary `local x = x` rule would otherwise imply), the name inside the body would refer to some outer `f` or to nothing at all.

2. Functions as values: stored, passed, returned

Once you accept that a function is a value, the rest is just Chapter 2's variable mechanics applied to a new kind of thing.

```
LUA
-- Stored in a table, exactly like a number would be.
local ops = {
  add = function(a, b) return a + b end,
  sub = function(a, b) return a - b end,
}
print(ops.add(2, 3))    --> 5

-- Passed as an ordinary argument.
local function apply(f, x, y)
  return f(x, y)
end
print(apply(ops.add, 4, 5))  --> 9

-- Returned from another function.
local function adder(n)
  return function(x) return x + n end
end
local add10 = adder(10)
print(add10(5))    --> 15
```

Nothing above needs new syntax. A function is passed and returned the same way a number or a table is: by value, where the "value" for a function happens to be a reference to its code plus whatever it needs to run, which is exactly what Section 4 covers next.

3. Anonymous functions

An anonymous function is simply a `function ... end` expression that is never assigned a name at all. You reach for one whenever the function is used exactly once, right where it is defined, most often as an argument.

```
LUA
local nums = {5, 3, 8, 1}

table.sort(nums, function(a, b) return a > b end)  -- descending
```

There is nothing structurally different about this function from `greet` above. It has the same `functiondef` grammar. It only happens to be written in a position where a name was never needed.

4. Closures: functions plus their upvalues

Here is where the scope rule from Chapter 2 gets cashed in. The manual defines the term directly (§3.5):

"A local variable used by an inner function is called an *upvalue* (or external local variable, or simply external variable) inside the inner function."

And the manual's visibility rule (§3.5, carried from Chapter 2) is what makes using an upvalue safe rather than accidental:

"Because of the lexical scoping rules, local variables can be freely accessed by functions defined inside their scope."

Programming in Lua gives the plain-language version and the term this lecture is named for: "a closure is a function plus all it needs to access its upvalues correctly" (PiL 1st ed., §6.1). The `adder` function above is already a closure: the inner `function(x) return x + n end` closes over `n`, the upvalue from the enclosing scope.

The classic worked example is a counter, adapted from PiL §6.1:

```
LUA
local function newCounter()
  local i = 0
  return function()
    i = i + 1
```

```

    return i
  end
end

local c1 = newCounter()
local c2 = newCounter()
print(c1()) --> 1
print(c1()) --> 2
print(c2()) --> 1, a SEPARATE i, not shared with c1

```

`c1` and `c2` are two different closures over two different `i` variables. This only makes sense because of the sentence Chapter 2 planted and never used: "each execution of a **local** statement defines new local variables" (Lua 5.4 Reference Manual, §3.5). Every call to `newCounter()` runs `local i = 0` again, and each run creates a fresh `i`. If Lua reused one shared `i` across calls, `c1` and `c2` would step on each other. They do not, because scoping guarantees a new variable per execution, not per declaration in the source text.

The same mechanism supports a genuinely practical pattern: a closure can restrict access to something by keeping the only reference to it private. PiL §6.1 gives the example of redefining `io.open` inside a closure so that the original, unrestricted function is reachable only through a wrapper that checks permissions first. The original `io.open` becomes an upvalue that no outside code can reach directly.

5. What to take away

A function is a value. `function f () ... end` is sugar for `f = function () ... end`, confirmed by the manual's own translation rule, and `local function` differs only in declaring the name before the body so the function can recurse. Functions are stored, passed and returned with the same mechanics Chapter 2 taught for every other value. An anonymous function is just a function expression with no name attached. A closure is a function plus the upvalues it needs, and Chapter 2's "fresh locals per execution" rule is exactly what lets two closures created from the same code capture independent state.

Next: what happens to the arguments a function is actually called with, which turns out to be far less strict than most languages you have used.

Key Terms

- **Function value:** a function is an ordinary Lua value, storable in a variable or table field, passable as an argument, and returnable from another function, with no special syntax for any of those.
- **Function-statement sugar:** 'function f () body end' is defined by the manual as translating to 'f = function () body end', an assignment of an anonymous function to a name.
- **local function:** declares the local name BEFORE defining the function body, specifically so the function can call itself by that name recursively.
- **Anonymous function:** a 'function ... end' expression with no name attached, most often written inline as an argument such as a comparator passed to `table.sort`.
- **Upvalue:** the manual's own term for a local variable from an enclosing function that an inner function uses. Also called an external local variable.
- **Closure:** a function plus everything it needs to access its upvalues correctly. Not a separate language feature, a consequence of lexical scoping.
- **Lexical scoping:** the rule that a function defined inside another function's scope can freely access that function's locals, which is what makes upvalues possible.
- **Fresh locals per call:** each execution of a local statement creates a NEW variable, which is why two closures built from the same function (e.g. two calls to `newCounter()`) capture independent state rather than sharing one.

Summary

A function is a value, and 'function f () ... end' is sugar for an ordinary assignment, per the manual's own translation rule. Functions are stored, passed and returned with the same mechanics as any other value. An anonymous function is just an unnamed function expression. A closure is a function plus the upvalues it captures from an enclosing scope, and Chapter 2's rule that each execution of a local statement makes a fresh variable is exactly why two closures from the same code (like two counters from `newCounter()`) do not interfere with each other.

Function Arguments

Lecture

Why this comes second

"Function Basics" established that a function call is just another value flowing through the language. This lecture is about what happens at the boundary of a call: how Lua matches the values you pass against the parameters a function declared, and it turns out that boundary is far looser than in most languages you have used.

By the end you should be able to:

- State the arity rule precisely, and predict the result of a call with too few or too many arguments.
- Use `...` to write a function that accepts any number of arguments.
- Use `select("#", ...)` and `select(n, ...)` correctly, and know the difference from `#`.
- Simulate named and optional arguments with a table.

1. Arity is never enforced

Lua does not check how many arguments a call passes against how many parameters a function declared. It adjusts, using exactly one rule stated in the manual's section on lists of expressions, for non-variadic functions (Lua 5.4 Reference Manual, §3.4.12):

"If there are more values than needed, the extra values are thrown away; if there are fewer values than needed, the list is extended with nil's."

That is one rule, not two, and it is the *same* rule Chapter 2 already taught for multiple assignment (`local a, b = 1, 2, 3` drops the `3`; `local a, b, c = 1, 2` leaves `c` as `nil`). A function call is not special: the argument list is adjusted to the parameter list exactly the way any list is adjusted to any other list in Lua.

```
LUA
local function greet(name, greeting)
  return (greeting or "hello") .. ", " .. name
end

print(greet("Ada"))           --> hello, Ada      (greeting is nil)
print(greet("Ada", "hi"))     --> hi, Ada
print(greet("Ada", "hi", "extra")) --> hi, Ada      ("extra" is dropped, silently)
```

No error in either direction. A missing argument is `nil`, which is why the `greeting or "hello"` idiom from Chapter 2 "Conditional Statements" works here without any special-casing for "argument not supplied." An extra argument simply vanishes. This is the single fact this lecture is built on: there is no arity checking to opt into, so any validation you want has to be written by hand, usually with `type()` checks or an assertion at the top of the function.

2. Varargs: ...

A function can decline the fixed-arity contract entirely by ending its parameter list with `...`. The manual is explicit about what that changes (§3.4.11):

"A variadic function does not adjust its argument list; instead, it collects all extra arguments and supplies them to the function through a vararg expression, which is also written as three dots."

```
LUA
local function sum(...)
  local total = 0
  for _, v in ipairs({...}) do
    total = total + v
  end
  return total
end

print(sum(1, 2, 3))    --> 6
print(sum())           --> 0
```

`...` can be mixed with named parameters, in which case the named ones are filled first and `...` collects whatever is left:

```
LUA
local function describe(label, ...)
  print(label, select("#", ...))
end

describe("scores", 90, 85, 77)  --> scores 3
```

3. select: counting and slicing the extra arguments

`...` on its own is an expression, not a value you can index or measure with `#` in a meaningful, guaranteed-safe way, because `#` on a list of values with possible `nil` holes is exactly the undefined-length trap Chapter 4 covers for tables. The tool built for this job is `select`, and this lecture checked its behavior against the real Lua 5.4 source. The actual implementation, `luaB_select` in `lbaselib.c`, is unambiguous:

- `select("#", ...)` returns the count of the extra arguments: it reads `n - 1`, where `n` is the total argument count to `select` itself (the selector string plus every vararg value), so subtracting the selector slot gives exactly the vararg count.
- `select(i, ...)` for a positive `i` returns every argument from position `i` onward, as multiple results.
- A negative `i` counts back from the end, like the negative positions in `string.sub` (the manual says `-1` is the last argument): `i = n + i` internally, before the range check.

```
LUA
local function report(...)
  print("count:", select("#", ...))
  print("from 2nd:", select(2, ...))
  print("last one:", select(-1, ...))
end

report("a", "b", "c")
-- count:      3
-- from 2nd: b  c
-- last one: c
```

The `#` versus `select("#", ...)` distinction, stated plainly: `#` measures the length of a table or string and is undefined in the presence of holes (nils in the middle). `select("#", ...)` measures the actual number of arguments a variadic call received, `nil` values included, because it counts call-time argument slots rather than walking a sequence looking for a border. `sum(1, nil, 3)` passed through `select("#", ...)` reports `3`; the same values dropped into a table and measured with `#` can report `1` or `3` depending on the table's internal representation. This is exactly why `select("#", ...)` is the correct tool for "how many arguments did I get," and a table plus `#` is not.

4. Named arguments via a table

Lua has no named-parameter syntax; a call is always positional. Programming in Lua's workaround (PiL 1st ed., §5.3) is a small piece of leverage rather than new syntax: a single table-constructor argument to a function needs no parentheses, so a table literal at a call site *reads* like named arguments even though it is really one ordinary positional argument.

```
LUA
local function rename(t)
    return os.rename(t.old, t.new)
end

rename{old = "temp.Lua", new = "temp1.Lua"}
-- No parentheses needed: `rename{...}` is a call with ONE table argument.
```

This scales to optional parameters with defaults, which is exactly the case where positional arguments get unreadable:

```
LUA
local function Window(t)
    local title = t.title or "untitled"
    local background = t.background or "white"
    local border = t.border
    if border == nil then border = true end
    -- ... build the window from title, background, border, t.width, t.height
end

Window{ x = 0, y = 0, width = 300, height = 200, title = "Lua" }
```

PiL states the practical reason directly: "this style of parameter passing is especially helpful when the function has many parameters, and most of them are optional" (§5.3). Note the `border == nil` check above rather than `border` or `true`: Chapter 2's falsity rule means `t.border == false` would be silently overridden by `or true`, so an explicit `nil` check is needed whenever `false` is a meaningful, distinct value from "not supplied."

5. What to take away

Arity is never enforced: one rule, extra values dropped and missing values padded with `nil`, identical to ordinary multiple assignment. `...` opts a function out of that adjustment and collects every extra argument. `select` is the tool for working with that collection: `select("#", ...)` for the count, `select(n, ...)` for a positional slice, negative `n` counting from the end, all confirmed against the real Lua 5.4 source. Named and optional arguments are simulated with a single table argument and the brace-call sugar, checking explicitly for `nil` wherever `false` needs to survive.

Next: what a function hands back, and why grabbing "all of it" depends entirely on where the call sits in an expression.

Key Terms

- **Arity is never enforced:** one rule for every call, extra arguments thrown away and missing arguments become `nil`, identical to Chapter 2's multiple-assignment rule.
- **Varargs (...):** a function ending its parameter list with three dots does not adjust its arguments; it collects all extra values into a vararg expression instead.
- **select with '#':** returns the COUNT of the extra arguments received, confirmed against the real Lua 5.4 source code (lbaselib.c) rather than the manual's prose.
- **select with a position:** returns every argument from position `n` onward, as multiple results; a negative `n` counts back from the end, with `-1` as the last argument.

- **# versus select count:** # measures a table or string and is undefined with holes; select('#', ...) counts actual call-time argument slots, nils included, and is the correct tool for counting varargs.
- **Named arguments via a table:** since Lua has no named-parameter syntax, a function taking one table argument can be called as f{key = value, ...} without parentheses, reading like named arguments.
- **Explicit nil check for booleans:** when simulating optional arguments with defaults, x or default silently overrides an explicit false; use if x == nil then x = default end whenever false is a meaningful value.

Summary

Arity is never enforced: one adjustment rule, extra arguments dropped and missing ones padded with nil, the same rule Chapter 2 taught for multiple assignment. Varargs (...) opt a function out of that adjustment and collect everything extra; select('#', ...) counts them and select(n, ...) slices them, both confirmed against the real Lua 5.4 source. Named and optional arguments are simulated with a single table argument and brace-call sugar.

Return Values

Lecture

Why this comes last

"Function Arguments" was about what a function receives. This lecture is the mirror image: what it hands back, and specifically the fact that a Lua function can hand back more than one value at once, with a rule for how many of them you actually get that depends entirely on *where* the call sits in your code.

By the end you should be able to:

- Write and call a function that returns multiple values.
- Predict exactly how many results survive when a multi-result call is not the last item in a list.
- State the mechanical definition of a tail call and identify one on sight.
- Explain, mechanically, what the object:method(args) sugar actually does, without yet building a class from it.

1. A function can return more than one value

There is nothing special about the syntax. return simply accepts a comma-separated list:

```
LUA
local function minMax(t)
  local lo, hi = t[1], t[1]
  for _, v in ipairs(t) do
    if v < lo then lo = v end
    if v > hi then hi = v end
  end
  return lo, hi
end

local low, high = minMax({4, 1, 9, 2})
print(low, high) --> 1 9
```

This is not a tuple, an object, or a wrapper of any kind. It is a genuine list of results, produced at the call site and consumed by whatever assigned, wrapped, or discarded them. How many of those results you actually receive is governed by one adjustment rule, and that rule is the entire subject of this lecture.

2. Truncation: only the LAST position sees everything

The manual's own statement of the rule (Lua 5.4 Reference Manual, §3.4.12, "Lists of Expressions, Multiple Results, and Adjustment") is the one to hold onto:

"When a multi-res expression is used as the last element of a list of expressions, all results from the expression are added to the list of values produced by the list of expressions." "When a multi-res expression is used in a list of expressions without being the last element, or in a place where the syntax expects a single expression, Lua adjusts the result list of that expression to one element."

In plain terms: a multi-result call only gets to keep *all* of its results when it is the very last thing in whatever list it appears in, an assignment's right-hand side, a table constructor, or a function call's argument list. Anywhere else in that list, it is squeezed down to exactly one value, its first result, and the rest are silently discarded.

Programming in Lua confirms this with a small, concrete example that is worth memorizing rather than reasoning through each time (PiL 1st ed., §5.1):

```
LUA
local function foo2()
  return "a", "b"
end

local x, y = foo2(), 20
print(x, y)    --> a 20
```

`foo2()` is *not* the last element of the list `foo2(), 20`, so it is adjusted to one value ("a"), and `y` gets `20` rather than `foo2()`'s second result "b". Move it to the last position and the behavior flips completely:

```
LUA
local x, y = 20, foo2()
print(x, y)    --> 20 a

local x, y, z = foo2()
print(x, y, z) --> a b nil  (foo2 is last, so ALL its results are used,
                           -- then padded with nil for z)
```

The same rule governs table constructors and argument lists, which is why `print(foo2())` prints both values but `print(foo2(), "!")` prints only "a" followed by "!". There is one rule, and its only condition is position in the list: last position keeps everything, any other position keeps exactly one value. This is the direct extension of Chapter 2's multiple-assignment adjustment and the "Function Arguments" lecture's arity rule; all three are the same mechanism applied to three different kinds of list.

3. Tail calls

A tail call is not about return values in general. It is about one specific shape of `return` statement, and the manual defines it by that shape alone (Lua 5.4 Reference Manual, §3.4.10):

"A call of the form `return functioncall...` is called a tail call."

Programming in Lua's framing makes the point vivid (PiL 1st ed., §6.3):

"A tail call is a kind of goto dressed as a call. A tail call happens when a function calls another as its last action, so it has nothing else to do."

```
LUA
local function f(x)
  return g(x)  -- a tail call: nothing happens after g returns
end
```

The value of this is concrete and confirmed: PiL states that "a proper tail call uses no stack space" (PiL 1st ed., §6.3), and the manual says "the called function reuses the stack entry of the calling function" (§3.4.10), so a chain of tail

calls, however deep, does not grow the call stack the way ordinary recursion does. But the definition is strict about the *shape* of the statement, and PiL gives three confirmed examples that look similar but are **not** tail calls:

```
LUA
return g(x) + 1    -- NOT a tail call: the addition happens after g returns
return x or g(x)   -- NOT a tail call: the result must still be adjusted to 1 value
return (g(x))      -- NOT a tail call: the parentheses force a 1-value adjustment
```

In each case there is still work left to do after `g` returns, an arithmetic operation, an adjustment forced by `or`, or an adjustment forced by parentheses, so the call cannot reuse the current stack frame. The manual's own wording narrows this further: `return functioncall` is a tail call only when it is "not in the scope of a to-be-closed variable," a detail that will not come up in this book's examples but is worth knowing exists. The rule to internalize is mechanical rather than a performance guess: **only `return g(...)`, written exactly that way, is a tail call.**

4. The colon-call sugar, previewed

The colon has a call side and a declaration side. This lecture covers the call side, because it is really about what a function call passes and gets back, not about objects; the declaration side arrives with objects in Chapter 5. The manual states the translation directly (§3.4.10):

"A call `v:name(args)` is syntactic sugar for `v.name(v,args)`, except that `v` is evaluated only once."

```
LUA
local t = { value = 10 }
function t.double(self)
    return self.value * 2
end

print(t.double(t))    --> 20, ordinary call, self passed explicitly
print(t:t.double())   --> 20, IDENTICAL, self passed implicitly as t
```

Mechanically, `t:double()` and `t.double(t)` are the same call, evaluating `t` exactly once. That is the entire fact this lecture needs. Why you would want a whole vocabulary of objects built around this sugar, with a `Class` table and an inheritance chain, is Chapter 5's subject once metatables are on the table. For now, hold onto the mechanical translation and nothing more: this sugar is about not repeating `self` explicitly, not about a `class` keyword Lua does not have.

5. What to take away

A `return` statement can produce a list of values, not just one. Whether a multi-result call keeps all of its results or gets truncated to one depends entirely on whether it is the last item in whatever list contains it, one rule shared with ordinary multiple assignment and function arguments. A tail call is defined by shape, `return g(...)` and nothing else, confirmed by three concrete non-examples where extra work after the call disqualifies it. And `v:name(args)` is confirmed sugar for `v.name(v, args)`, worth knowing mechanically now and worth revisiting properly in Chapter 5.

This closes Chapter 3. Chapter 4 moves to the one data structure Lua has, and functions come back almost immediately as the values a table's fields can hold.

Key Terms

- **Multiple return:** a return statement can list several values, comma-separated, with no wrapper object involved; the values are a genuine list produced at the call site.
- **Truncation rule:** a multi-result call keeps ALL its results only as the LAST element of a list (assignment, table constructor, or argument list); anywhere else it is adjusted to exactly one value, its first result.

- **Mid-list truncation example:** in 'local x, y = f(), 20', f() is NOT last, so it yields only its first result and y gets 20, not f()'s second value.
- **Tail call:** defined by shape alone, 'return g(...)' and nothing else. A tail call reuses the stack entry of the calling function, so there is no limit on the number of nested tail calls.
- **Non-tail-call trio:** 'return g(x) + 1', 'return x or g(x)', and 'return (g(x))' are confirmed NOT tail calls, because each forces work or an adjustment after g returns.
- **Colon-call sugar:** 'v:name(args)' is confirmed sugar for 'v.name(v, args)', evaluating v once. Mechanical fact only; the reason to want this is Chapter 5's subject.

Summary

A return statement can produce a list of values. Whether all of them survive or only the first one does depends entirely on position: last in a list keeps everything, anywhere else keeps one, the same adjustment rule as arity and multiple assignment. A tail call is defined mechanically as 'return g(...)' and nothing more, confirmed against three concrete non-examples that add work after the call. The colon-call sugar, v:name(args) as v.name(v, args), is confirmed here mechanically and previewed for the object system Chapter 5 builds on top of it.

Chapter 4: Tables

Introduction to Tables

Lecture

Why this comes first

Lua has exactly one compound data structure, and it is the table. There is no separate array type, no separate dictionary type, no separate object type, no separate set type. Everything you will build for the rest of this book, arrays, records, matrices, and eventually (Chapter 5) sets and objects, is a table wearing a different hat. This chapter is where you meet the thing itself, before it starts wearing any of those hats. Get comfortable with what a table actually is now, and every later chapter is just a new way of using it.

By the end you should be able to:

- Build a table with a constructor, in record style, list style, or a mix of both.
- Explain why a table can be indexed by almost any value, not just integers.
- Use `nil` to delete a key from a table, and explain why that is the same mechanism as never having set it.
- Predict what happens when you assign one table to a second variable, and explain why it is not a copy.

1. Tables are made with a constructor

A table constructor is an expression that builds a new table, most simply written as an empty pair of braces. The manual is direct about this (Lua 5.4 Reference Manual, §3.4.9):

"Table constructors are expressions that create tables. Every time a constructor is evaluated, a new table is created."

That last clause matters: a constructor is not a static template, it runs every time control passes through it, so a constructor written inside a loop or a function body produces a fresh table on every pass.

The constructor supports two field forms side by side. Record-style fields use a bare name:

```
LUA
-- Record style, per Programming in Lua §3.6.
local point = {x = 0, y = 0}
```

which the book states plainly is "equivalent to `a = {}; a.x=0; a.y=0`" (PiL 1st ed., §3.6), just written in one expression instead of three statements. List-style fields skip the name entirely and are numbered automatically, starting from 1:

```
LUA
-- List style. "the first element has always index 1, not 0" (PiL §3.6).
local days = {"Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"}
```

The manual's own grammar makes explicit what is really happening: a plain `exp` field is short for `[n] = exp`, where `n` starts at 1 and increases by one for every plain field in the constructor. A general field can also use any expression as its key, `[exp1] = exp2`, which is what makes non-string, non-sequential keys possible from inside a constructor at all.

Record and list style mix freely in a single constructor, and constructors nest. PiL's own example is a polyline built from four point-records in one list-style outer table:

```
LUA
-- Nested constructors: a list of records, per PiL §3.6.
local polyline = {
  color = "blue",
  thickness = 2,
  {x = 0,    y = 0},
  {x = -10, y = 0},
  {x = -10, y = 1},
  {x = 0,   y = 1},
}
```

Here `polyline.color` and `polyline.thickness` are record fields, and `polyline[1]` through `polyline[4]` are each "a table representing a record" (PiL §3.6): a table of tables, built in one expression. Trailing commas are always legal after the last field, which the book notes exists specifically so code that *generates* Lua tables never has to special-case the final entry.

2. Keys can be almost anything, including tables and functions

The manual's description of a table is not "an array" or "a dictionary." It is a general-purpose associative structure (Lua 5.4 Reference Manual, §2.1):

"The type table implements associative arrays, that is, arrays that can have as indices not only numbers, but any Lua value except **nil** and NaN."

Programming in Lua states the same rule with **nil** as the exception: a table can be indexed by "any other value of the language, except **nil**" (PiL 1st ed., §2.5). Assigning with a key that is **nil** or NaN raises an error ("table index is nil", "table index is NaN"), and reading with one just gives **nil**. Every other value works: strings, booleans, numbers, functions, and other tables are all legal keys.

```
LUA
local counts = {}
counts["apples"] = 3      -- string key
counts[true] = "yes"      -- boolean key
counts[print] = "a function key, unusual but legal"

local key_table = {}
counts[key_table] = "a table used as a key, also legal"
```

This is worth sitting with, because it is easy to arrive at tables already thinking of them as "Lua's array type" and miss that the array shape is a special case (all keys are consecutive positive integers) of something more general.

3. nil means "not present," and assigning it deletes

The manual states the deletion rule directly, as a consequence of what **nil** means for table membership (Lua 5.4 Reference Manual, §2.1):

"Any key associated to the value **nil** is not considered part of the table."

Programming in Lua gives the same rule as something you actively do: "you can assign **nil** to a table field to delete it" (PiL 1st ed., §2.5). These are two ways of stating one fact. There is no separate "delete this key" operation in Lua; deleting a key and setting its value to **nil** are the exact same act, because a table never distinguishes "this key holds **nil**" from "this key was never set."

```
LUA
local t = {a = 1, b = 2}
t.b = nil      -- b is now gone, not "present with value nil"
print(t.b)     --> nil, indistinguishable from a key that was never set
```

This is also why reading a missing key never raises an error the way it would in many languages: `t.b` after the delete above returns exactly the same `nil` you would get from `t.c`, a key that was never mentioned at all. The table cannot tell the two apart, and neither can you from the outside.

4. Tables are references, not values, and assignment never copies

This is the fact with the most consequences of anything in this lecture. The manual states it about tables (and functions, threads, and full userdata) together, as a category (Lua 5.4 Reference Manual, §2.1):

"Tables, functions, threads, and (full) userdata values are *objects*: variables do not actually contain these values, only *references* to them."

And directly after, the operational consequence:

"Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy."

Programming in Lua puts the same idea in plainer terms: "you may think of a table as a dynamically allocated object; your program only manipulates references (or pointers) to them. There are no hidden copies or creation of new tables behind the scenes" (PiL 1st ed., §2.5).

```
LUA
local original = {1, 2, 3}
local alias = original      -- NOT a copy; alias and original name the SAME table

alias[1] = 99
print(original[1])          --> 99, because there was only ever one table

local function mutate(t)
  t[2] = "changed"
end
mutate(original)
print(original[2])          --> "changed", the function received a reference, not a copy
```

Contrast this with a number or a string, which Chapters 1 and 2 already showed you behave the ordinary, copied way:

```
LUA
local n1 = 5
local n2 = n1
n2 = 10
print(n1)  --> 5, unaffected, because numbers are values, not objects
```

A table assigned to a second variable, passed into a function, or returned from one, is never duplicated. Both names refer to the one table that exists. If you actually want an independent copy, you have to build one yourself, field by field (or with a helper function you write for that purpose); the standard table library has no function that clones a table.

5. What to take away

A table is built with a constructor, which creates a genuinely new table on every evaluation and supports record-style, list-style, and freely nested fields in one expression. A table's keys can be almost any value except `nil` and NaN, which is the manual's own way of saying a table is not fundamentally an array or a dictionary, those are just shapes it can take. Assigning `nil` to a key removes it, because a table has no other notion of "not present." And a

table is an object: variables hold references to it, never the table itself, so assignment, passing it to a function, and returning it all share one table rather than making a copy. Keep that last sentence in mind through the rest of this chapter.

Next: what Lua's own `table` library does with these tables, and the real contract behind the `#` operator, including the one case where it genuinely does not have a defined answer.

Key Terms

- **Table constructor:** an expression, most simply `{}`, that creates a genuinely new table every time it is evaluated, confirmed by the manual's own wording.
- **Record-style field:** a table field written as `name = exp`, equivalent to a separate assignment statement after the fact.
- **List-style field:** a plain `exp` field inside a constructor, auto-indexed with consecutive integers starting at 1.
- **Nested constructor:** a table constructor whose own fields are themselves table constructors, such as a list of record-style tables.
- **Table as object:** the manual's own term for tables, functions, threads and full userdata, meaning a variable holds only a reference to one, never the thing itself.
- **Nil as deletion:** assigning `nil` to a table field removes that key, because the manual defines a `nil`-valued key as not part of the table at all, not as a special empty entry.
- **Reference semantics:** assignment, passing a table to a function, and returning a table from one all share the SAME table; none of those operations ever copies it.
- **Keys of any type:** a table may be indexed by any Lua value except `nil` and `NaN`, including strings, booleans, functions and other tables, not only integers.

Summary

A table is built with a constructor, which creates a new table on every evaluation and supports record-style, list-style, and freely nested fields. A table's keys can be almost any value except `nil` and `NaN`. Assigning `nil` to a key deletes it, because the manual defines a `nil`-valued key as absent, not present with an empty value. And a table is an object: variables hold references to it, never the table itself, so assignment, passing it to a function, and returning it all share one table rather than making a copy.

Table Operations

Lecture

Why this comes next

"Introduction to Tables" gave you the table itself: a constructor, keys of almost any type, `nil` as deletion, and reference semantics. This lecture gives you the tools Lua ships for working with tables that happen to be used as arrays, the `table` library, and it gives you the single most misunderstood operator in the language: `#`. By the end of this lecture you should never again write code that assumes `#t` means "the number of things in `t`," because that is not what it means.

By the end you should be able to:

- Use `table.insert` and `table.remove` to add and remove elements at either end or in the middle of an array-shaped table.
- Use `table.concat` to join an array's elements into a string.
- Use `table.sort` with and without a custom comparator.

- State the actual, formal contract of `#t`, including exactly why it is undefined for a table with holes, quoting the manual rather than a paraphrase of it.

1. `table.insert` and `table.remove`

Both functions operate on a table being used in the array shape: consecutive integer keys starting at 1. Confirmed directly from the real Lua 5.4 source (`ltablib.c`), `table.insert` has two forms. Called with two arguments, it appends:

```
LUA
local t = {10, 20, 30}
table.insert(t, 40)      -- append at the end
-- t is now {10, 20, 30, 40}
```

Called with three arguments, the middle one is a position, and the source confirms it validates that position is in range (an out-of-range position is an error, "position out of bounds") before shifting every later element up by one to make room:

```
LUA
table.insert(t, 1, 5)    -- insert 5 at position 1, shifting everything else up
-- t is now {5, 10, 20, 30, 40}
```

Programming in Lua describes the same function in the same terms: it "inserts an element in a given position of an array, moving up other elements to open space," and without a position argument "inserts the element in the last position of the array (and, therefore, moves no elements)" (PiL 1st ed., §19.2).

`table.remove` is the mirror image. Confirmed from source, it validates the position (when one is given) is within `[1, size + 1]`, reads the value at that position, shifts every later element down by one to close the gap, sets the now-vacant last slot to `nil`, and returns the value that was removed:

```
LUA
local last = table.remove(t)    -- removes and returns the last element
local first = table.remove(t, 1) -- removes and returns element 1, shifting the rest down
```

PiL's own description matches: `table.remove` "removes (and returns) an element from a given position in an array, moving down other elements to close space," defaulting to the last element with no position given, and the book flags plainly that removing from a low position is less efficient precisely because of that shift (PiL 1st ed., §19.2).

Both functions assume the array shape. Using them on a table with non-sequential or non-integer keys is not an error, but the "moving other elements" behavior only makes sense for a proper sequence, and mixing them with holes is exactly the kind of situation Section 4 below tells you to avoid relying on `#` for.

2. `table.concat`

`table.concat` joins a range of an array's elements into a single string, with an optional separator and an optional start and end index. Confirmed from source, every element in the joined range must be a string or a number; anything else raises an error naming the offending index and the operation ("invalid value ... in table for 'concat'").

```
LUA
local words = {"the", "quick", "brown", "fox"}
print(table.concat(words))      --> "thequickbrownfox"
print(table.concat(words, " ")) --> "the quick brown fox"
print(table.concat(words, ", ", 2, 3)) --> "quick, brown"
```

There is no equivalent library function for building a table by splitting a string; that direction goes through the string library's pattern matching, which is Chapter 6's subject.

3. table.sort

`table.sort` sorts an array-shaped table in place. Confirmed from source, it uses Lua's own `<` operator as the default order when no comparator is supplied, and when one is supplied it must be a function taking two elements and returning true when the first should sort before the second. PiL states the same contract in plainer language: the order function "receives two arguments and must return true if the first argument should come first in the sorted array" (PiL 1st ed., §19.3).

```
LUA
local nums = {5, 3, 8, 1}
table.sort(nums)
-- nums is now {1, 3, 5, 8}, using the default < comparison confirmed above

table.sort(nums, function(a, b) return a > b end)
-- nums is now {8, 5, 3, 1}, descending, using the supplied comparator
```

Confirmed from source (`ltablib.c`), the sort is a quicksort that rejects an array of `INT_MAX` or more elements ("array too big") and, when a partition turns out badly unbalanced, switches to a randomized pivot for intervals of 100 or more elements (the `RANLIMIT` constant in the source). None of that changes how you call it.

4. The real contract of

Here is the gotcha the whole lecture has been building to. The manual gives `#` a precise, formal definition for tables, using the idea of a *border* (Lua 5.4 Reference Manual, §3.4.7):

"a border in a table `t` is any non-negative integer that satisfies the following condition: (`border == 0` or `t[border] ~= nil`) and (`t[border + 1] == nil` or `border == math.maxinteger`)"

In plain terms, a border is a position where "something is here" (or you are at position 0) immediately followed by "nothing is here." A table with exactly one border is called a *sequence*, and for a sequence, `#t` reliably gives you that one border, which matches the intuitive "length of the array."

The trouble is tables with more than one border. The manual states the consequence directly:

"When `t` is not a sequence, `#t` can return any of its borders. (The exact one depends on details of the internal representation of the table...)"

And it gives its own concrete example of exactly this:

"The table `{10,20,30,nil,50}` has two borders (3 and 5), and therefore it is not a sequence. (The `nil` at index 4 is called a *hole*.)"

```
LUA
local t = {10, 20, 30, nil, 50}
print(#t) -- could print 3, could print 5. Both are valid borders.
          -- The manual explicitly does not promise which one you get.
```

This is not a bug. The manual defines `#t` this way and says the exact border returned depends on the internal representation of the table. A table with a hole in it has more than one border, so there is no single length for `#t` to report. **The practical rule:** only trust `#t` on a table you know is a genuine sequence (no holes, ever). If you build a table by appending with `table.insert(t, value)` and never assign `nil` into the middle of it, `#t` will behave exactly as you expect. The moment a `nil` lands anywhere but the very end, `#t` stops being trustworthy, silently, with no error to warn you.

5. Counting when a table has holes

There is no library function that reliably counts the elements of a table that might have holes. If you need the real count, the honest answers are: keep the table hole-free in the first place (the far better fix in almost every real case), or maintain your own explicit count field alongside the table, updated by hand at every insert and remove.

6. What to take away

`table.insert` and `table.remove` add and remove elements from an array-shaped table, shifting the rest to keep it contiguous, confirmed directly from the Lua 5.4 source. `table.concat` joins a range into a string, requiring every joined element to be a string or a number. `table.sort` sorts in place, defaulting to `<` and accepting a custom comparator function. And `#t` has a precise, manual-quoted definition, a *border*, that gives you exactly what you expect on a hole-free sequence and explicitly, by design, does not promise anything specific once a table has more than one border. There is nothing built in to count a table with holes, so keep the table hole-free or keep your own count.

Next: how `ipairs` and `pairs` actually walk a table, why they disagree with each other and with `#` about what a "hole" means, and how the same table does double duty as an array and a dictionary.

Key Terms

- **table.insert:** appends at the end by default, or inserts at a given position while shifting later elements up, confirmed against the real Lua 5.4 source.
- **table.remove:** removes and returns the last element by default, or an element at a given position while shifting later elements down, confirmed against the real Lua 5.4 source.
- **table.concat:** joins a range of an array's elements into one string with an optional separator, requiring every joined element to be a string or number.
- **table.sort:** sorts an array-shaped table in place, defaulting to the language's own less-than operator, or using a supplied comparator function that returns true when its first argument should sort first.
- **Border:** the manual's own formal term for a position where a value is present and the next position is not, which is exactly what `#` is defined in terms of for a table.
- **Sequence:** a table with exactly one border, the only case where `#` is guaranteed to give the answer you intuitively expect.
- **Holes trap:** once a table has more than one border, the manual states `#t` may return ANY of them, quoted directly rather than paraphrased, and gives its own example, the table `{10, 20, 30, nil, 50}`, having two borders.
- **No count for holey tables:** there is no library function that reliably counts a table with holes, so keep the table hole-free or maintain an explicit count field.

Summary

`table.insert` and `table.remove` add and remove elements from an array-shaped table, shifting the rest to stay contiguous, confirmed directly from the Lua 5.4 source. `table.concat` joins a range into a string. `table.sort` sorts in place, defaulting to `<` and accepting a custom comparator. And `#t` has a precise, manual-quoted definition, a *border*, that behaves exactly as expected on a hole-free sequence and explicitly does not promise anything specific once a table has more than one border. Nothing built in counts a table with holes, so keep tables hole-free or keep your own count.

Tables as Arrays and Dictionaries

Lecture

Why this comes last

The "Introduction to Tables" and "Table Operations" lectures gave you the table itself and the tools for treating one as an array. This lecture closes Chapter 4 by making the "one structure, many hats" idea concrete: the exact same table type is what you reach for whether you are building an ordered list or an unordered lookup, and Lua gives you two different ways to walk one, `ipairs` and `pairs`, that disagree with each other on purpose. Getting this straight now is what makes Chapter 5's "tables as sets" feel obvious rather than like a new structure.

By the end you should be able to:

- Explain 1-based indexing as a library convention rather than a rule the table type itself enforces.
- Choose `ipairs` or `pairs` correctly, and explain what each one actually does when it meets a hole.
- State plainly that `pairs`' iteration order is not something you may rely on.
- Build and read a nested table, a table whose values are themselves tables.

1. 1-based indexing is a convention, not a rule

"Introduction to Tables" already showed that a table can be keyed by almost anything. Nothing in the table type itself says an array-shaped table must start at index 1. *Programming in Lua* is direct about this being a choice, not a language constraint (PiL 1st ed., §11.1):

"it is customary in Lua to start arrays with index 1. The Lua libraries adhere to this convention; so, if your arrays also start with 1, you will be able to use their functions directly."

The book goes on to build arrays starting at 0 and even at negative indices, to demonstrate that the table itself does not care. What actually enforces 1-based indexing, in practice, is everything ELSE: the table constructor auto-indexes plain fields starting from 1 (confirmed in "Introduction to Tables", §3.4.9 of the manual), and `table.insert`, `table.remove`, `table.concat`, `table.sort` and `ipairs` (below) all assume index 1 is where an array-shaped table begins. Depart from that convention and every one of those tools either misbehaves or simply does not see your data; nothing raises an error to warn you. Use 1-based indexing not because the language demands it, but because the entire standard library was built assuming you will.

2. `ipairs`: a well-defined walk that stops at the first hole

`ipairs` is the tool for walking a table as an array, in order, from index 1. Its exact behavior was confirmed directly from the real Lua 5.4 source (`lbaselib.c`). The source shows `ipairs` returning an iterator function, the table, and a starting control value of 0; on each call, the iterator increments the index by one, reads that position, and stops the moment that read comes back `nil`.

```
LUA
local t = {"a", "b", "c", nil, "e"}

for i, v in ipairs(t) do
    print(i, v)
end
-- 1 a
-- 2 b
-- 3 c
-- STOPS here. "e" at index 5 is never visited, because ipairs
-- hits the nil at index 4 and halts, by design.
```

This is worth naming explicitly, because it resolves the "Table Operations" confusion cleanly: `ipairs` is not undefined on a holey table the way `#` is. It is completely well-defined; the manual says it iterates "up to the first absent index" (§6.1), so it stops at the first hole every time. `#`, by contrast, makes no promise at all about which border it reports once a table has more than one. Two different kinds of "does not do the whole array," and only one of them is actually a gotcha.

`ipairs` runs on the generic `for` statement's own iterator protocol, which the manual does confirm directly (Lua 5.4 Reference Manual, §3.3.5): the loop evaluates its expression list to produce "four values: an iterator function, a state, an initial value for the control variable, and a closing value," and then "at each iteration, Lua calls the iterator function with two arguments: the state and the control variable." `ipairs(t)` returns three values, "an iterator function, the table `t`, and 0" (§6.1), so it supplies the iterator, `t` as the state, and 0 as the starting control value, and the closing value is `nil`.

3. pairs: every key, in an order you may not rely on

`pairs` is the tool for walking a table's ENTIRE contents, array part and hash part alike, regardless of what the keys look like. Confirmed from source, `pairs` (absent a `__pairs` metamethod, which ordinary use does not need) returns exactly three values: the built-in `next` function, the table itself, and `nil` as the starting key. *Programming in Lua* gives the identical definition as literal code (PiL 1st ed., §7.3):

```
LUA
-- The book's own definition of pairs, quoted directly.
function pairs (t)
  return next, t, nil
end
```

```
LUA
local t = {"a", "b", nil, "d", x = 1, y = 2}

for k, v in pairs(t) do
  print(k, v)
end
-- Visits every key with a non-nil value: 1, 2, 4, x, y (in SOME order).
-- Unlike ipairs, pairs does not stop at the hole at index 3, and it
-- also visits the string keys x and y, which ipairs would never reach.
```

The order those keys come back in is the one thing about `pairs` you must never build logic around. PiL states this directly, describing what `next` itself does: "The call `next(t, k) ...` returns a next key in the table, in an *arbitrary* order" (PiL 1st ed., §7.3). The real Lua 5.4 table implementation gives a concrete reason this is true rather than just an arbitrary restriction: tables are stored, per the implementation's own comments, as "an array part and a hash part," and a traversal visits "all elements in the array part, then elements in the hash part." Hash-part order depends on internal details (hashing, table size, insertion history) that have nothing to do with the order you wrote the keys in, and nothing about that order is a promise you can lean on across different tables, different Lua builds, or even different runs of the same code with a table built a different way. If order matters, sort the keys yourself (`table.sort` on a list of keys you collect first) before using them; never assume `pairs` handed them to you in a useful sequence.

4. Choosing between ipairs and pairs

The choice is mechanical once you know what each one actually promises:

- Use `ipairs` when the table is (or should be) a genuine sequence and you want the array elements, in index order, stopping cleanly at the first gap. This is the right tool for "the list of things," where "the things" means index 1, 2, 3, ... with nothing skipped.

- Use `pairs` when you want everything a table holds, keys of any type, array-shaped or not, and you do not care about, or cannot rely on, the order you get them in. This is the right tool for "the record," "the lookup table," or "every field, whatever it turns out to hold."

Using `ipairs` on a table that is really a dictionary silently visits nothing (there is no index 1 to start from), with no error. Using `pairs` on a table you intended as a strict, ordered array works, but throws away the one guarantee (order) that made it useful as an array in the first place. Neither misuse crashes; both just quietly give you the wrong thing, which is exactly why naming the distinction explicitly, rather than reaching for whichever one comes to mind first, is worth the extra second of thought every time.

5. Nested tables: the same structure, one level deeper

Because a table's values can be anything, including other tables, nesting falls out of everything already covered this chapter with no new rule required. The "Introduction to Tables" lecture's polyline example already showed this: a list-style outer table whose elements are themselves record-style tables. The same idea extends to a dictionary of dictionaries, or any mix:

```
LUA
-- A nested structure: a dictionary of records.
local students = {
  alice = {age = 20, major = "CS"},
  bob   = {age = 22, major = "Art"},
}

print(students.alice.age)    --> 20
students.bob.major = "Music" -- reaches into the nested table and mutates it

for name, info in pairs(students) do
  print(name, info.age, info.major) -- order not guaranteed, per Section 3
end
```

Because tables are references ("Introduction to Tables"), `students.alice` is not a copy of Alice's record when you read it out; it is the same table `students` holds internally. Writing `local a = students.alice; a.age = 21` changes `students.alice.age` too, for exactly the reference-semantics reason "Introduction to Tables" covered: there is only ever one table for `alice`'s data, no matter how many variables currently point at it.

6. What to take away

1-based indexing works because the constructor and the whole standard library agree on it, not because the table type demands it. `ipairs` walks an array in order and is guaranteed, by source-confirmed design, to stop at the first hole; `#` merely does not promise anything once there is more than one hole, and those are different kinds of "incomplete." `pairs` visits every key a table has, in an order that is explicitly arbitrary per Programming in Lua's own definition of it, so never build logic that depends on that order. And nesting tables inside tables needs no special syntax at all: a table's value slot can hold another table exactly as easily as it holds a number, and reference semantics means there is still only ever one copy of the nested data no matter how you reach it.

That is Chapter 4. One structure, three lectures, and the entire rest of this book, sets, metatables and objects in Chapter 5, modules in Chapter 6, is going to turn out to be this same table doing one more job.

Key Terms

- **1-based indexing as convention:** nothing in the table type requires starting at index 1, but the constructor's auto-indexing and the entire table library assume it, so departing from it breaks those tools silently.
- **`ipairs`:** walks an array-shaped table from index 1 upward and is confirmed, from the real Lua 5.4 source, to stop the moment it reaches the first nil, a well-defined behavior rather than an undefined one.

- **pairs**: walks every key a table holds, confirmed from source to be built from next, the table, and nil as a starting point, matching Programming in Lua's own one-line definition of it.
- **Arbitrary iteration order**: Programming in Lua states directly that next visits keys 'in an arbitrary order', so code must never depend on the sequence pairs produces.
- **ipairs versus the holes trap**: ipairs and # both stop short on a table with holes, but ipairs does so in a guaranteed, source-confirmed way while the stopping point of # is explicitly undefined.
- **Nested table**: a table whose values are themselves tables, requiring no special syntax; reading and writing through the nesting uses ordinary dot or bracket indexing at each level.
- **Dictionary of records**: a common nested shape, a table keyed by name whose values are themselves record-style tables, such as a table of students each holding their own age and major.

Summary

1-based indexing works because the constructor and the whole standard library agree on it, not because the table type demands it. ipairs walks an array in order and is guaranteed, by source-confirmed design, to stop at the first hole; # merely does not promise anything once there is more than one hole, which is a meaningfully different kind of incomplete. pairs visits every key a table has, in an order Programming in Lua's own definition calls arbitrary, so that order must never be relied on. Nesting tables inside tables needs no special syntax at all, and reference semantics from "Introduction to Tables" means there is still only one copy of the nested data no matter how many variables currently reach it.