Lua 101

Crash course in awesomeness!

Contents

- Before we start
- Numbers
- Memory
- Text
- Truth and logic
- Control flow
- Bonus stuff
- Exercices

Before we start

Running the interpreter

- Lua source usually in `source.lua` files
- From the command line: `lua source.lua`
- From a menu (or shortcut) in a Lua-friendly text editor
  - `Tools -> Build` or `Ctrl-B` in Sublime Text
- One program = one Lua file (for now)
- Interpreter reads the whole file and executes it line by line

For very small tests: interactive Lua interpreter

- Windows: Start menu -> search for Lua -> "Lua (Command Line)"
- Others: run `lua` in a terminal

Comments and whitespace

Comments: start with `--`, end at the end of line

- documenting
- separating sections in code
- disabling code temporarily

```
-- Here we print some things:
print("Hello world") -- the first thing
print("This is great") -- the second thing
-- print("Not this!")
```

Spaces, tabs and new lines ignored:

```
print(        "Hello world"
)print("This is great"            )
```

Good style: one instruction per line, consistent spacing

Basic output: `print()`

An instruction to print something to the console. Used mostly to:

- explain what is going on
- show the result of a computation

```
print("Hello, user!")
print("I will now print 5:")
print(5)
print("I'm done, bye!")
```

One `print()` instruction: one line of output

Numbers

> " If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is. (John von Neumann)

Numbers

The most basic *type* of data you can use in Lua:

```lua
-- let's print some numbers:

print(123)
print(-2)
print(45.024)
print(-999999999.9)
```

Mini exercise: printing numbers

Write a program that gives the following output:

```
245
-4000
19.56
-0.333
```

Not that fascinating... Let's actually use the CPU.

Basic number operations

Operation	Math notation	Symbol
Addition	+	+
Substraction	-	-
Multiplication	× or ·	*
Division	/, ÷ or —	/

```lua
-- Lua as a simple calculator:

print(3 + 18.5)
print(20 - 5)
print(2763 * 47684)
print(60 / 4)
```

Mini exercise: computing things

Write a program that outputs the answers to these questions:

- A group of 23 people goes to the movies. Each ticket costs 13€. How much do they pay in total?
- You have 75 cookies. You eat 54. How many cookies are left?
- A pirate crew of 45 mates finds a treasure worth 15030€, they split it evenly. How much booty does each pirate receive?
- A wedding reception has 36 lady guests and 29 gentlemen guests. How many guests in total?

Bigger operations

Can chain the operations as much as you want:

```
print(2 + 4 + 6 - 7 + 10 - 8 + 9)
print(3 * 4 * 10 / 2 * 5 / 10)
```

Like in math, * and / are applied before + and -:

```
print(3 + 4 * 6)      -- 27
```

Like in math, use parentheses (and) to force the order:

```
print((3 + 4) * 6)   -- 42

-- average of 5 numbers:
print((2 + 7 + 4 + 10 + 6) / 5)
```

Mini exercise: bigger operations

Write a program that computes the answer to the following question, using a single `print()` instruction:

For your birthday, your 23 school friends gave you each 5€, your 3 siblings gave you each 7€, and your significant other gave you 55€. How much money did you get in total?
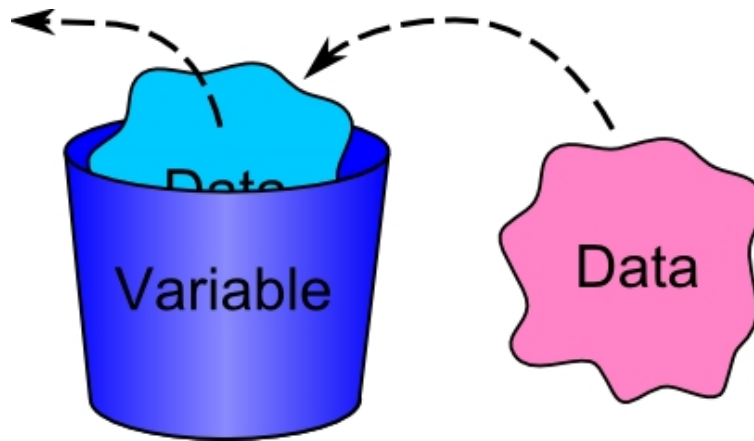
Memory

> A clear conscience is the sure sign of a bad memory. (Mark Twain)

Variables: using memory

A variable:

- is a space in memory
- it has a name
- we can store a value (like a number) in it
- we can retrieve its content later to use other computations

Storing data in variables

To store a value in a variable, we use the `=` symbol:

```
aVariable = 10
anotherVariable = 2 * 3 + 4
```

The computer will compute ("evaluate") what is on the right-hand side, and store the result in the variable whose name is on the left-hand side.

Variable names

Anything you want with the following restrictions:

- contains only letters, numbers or underscores ("_")
- cannot start with a number
- no special letters (like ä, å, ö, é, etc)

Traditional styles for multiple words:

- camel case: `thisIsMyVariable`
- underscores: `this_is_my_variable`

Hint: in small programs, small variables are OK (`a`, `b`, etc.). In bigger ones, try to give them meaningful names, to remember what they represent.

Retreiving values from variables

Simply use the variable name directly (in `print()` or in an operation):

```
-- storing
myVariable = 10

-- using
print(myVariable)
print(myVariable + 2)
print(myVariable * myVariable - 1)
```

Retreiving values from variables

You can use as many variables as you need, and store new results in new variables:

```
-- simulation of buying a TV
money = 1000
priceOfTelevision = 200
moneyLeftAfterBuying = money - priceOfTelevision

print(moneyLeftAfterBuying)
```

Mini exercice: using variables

Using variables named `amountOfGold`, `pricePerKilo`, and `totalValue`, write a program that answers the following question:

Gold is nowadays worth 36452.84€ per kilo. If you happened to find 4.56 kilograms of gold in your backyard, how much would that be worth?

Replacing values in variables

The name "variable" means that the value in a variable can change. You can reuse a variable as much as you want:

```
a = 10
print(a)     -- 10
a = 20
print(a)     -- 20, the 10 was replaced
```

A common idiom:

```
a = 100      -- or whatever
print(a)     -- 100

a = a + 1    -- this is called "incrementing"
print(a)     -- 101
```

Mini exercice: reusing variables

Using a single variable, write a program that prints your daily amount of cookies, knowing that:

- You start with 24 cookies
- On the first day, you eat 13 cookies
- On the second day, you eat 5 cookies
- On the third day, you buy 45 cookies
- On the fourth day, you give 32 cookies to your friend
- On the fifth day, you eat 10 cookies

Mini exercice: reusing variables

Write a program that swaps the content of two variables. For instance, complete the following program:

```
a = 10
b = 5

-- insert your code here

print(a)
print(b)
```

so that it outputs:

```
5
10
```

Absence of value

A variable that was never used doesn't contain any value. We say it "is nil", or "contains nil":

```
print(a)      -- nil
a = 5
print(a)      -- 5
```

Trying to do math with nil will make the interpreter angry:

```
a = 5
print(a + b)      -- whoops, b is nil
```

nil is a special value that means "no value", or "this variable is empty".

Forgetting values

The nil value can be assigned to a variable to erase it, using the `nil` keyword

```
a = 5
print(a)     -- 5
a = nil
print(a)     -- nil
```

In general, you don't need to worry about saving memory or "recycling" variables.

A bit of vocabulary

These symbols:

```
+ - * /
```

are called "operators". They "operate" on numbers. When, in the source code, we combine numbers and/or variables with operators, we form an "expression":

```
a + 4 * (b - 200) / c
```

Note: a variable by itself, or a number by itself is also an expression:

```
aVariable
100
```

When the interpreter encounters an expression in the code, it will "evaluate" it (compute its value by applying the operators, ie. "do the math").

Expressions

We can put an expression directly in `print()` (to print its value):

```
print(200 * 246 - 2)
```

or in the right-hand side of an assignment (to store its value in a variable):

```
result = 200 * 246 - 2
```

Later, we'll see other places where we can use expressions.

Text

> There is nothing to writing. All you do is sit down at a typewriter and bleed. (Ernest Hemingway)

Text

- "character strings" or "strings"
- another type of data that Lua programs can manipulate.
- sequences of characters
- in the source, they are written between " (double quotes)

We've already used them for output messages:

```
print("This is a string!")
print("These are ok: äåöÈ")
```

We use them any time we need to manipulate text:

- user input
- body of an email
- URL in your web browser
- ...

Mini exercice: strings

Write a program that outputs the following:

```
Welcome to the wonderful world...
Of programming!
Isn't this great?
```

Strings are values

Like any value, strings can be stored in variables and retrieved:

```
myString = "Hello"
otherString = "World"
print(myString)
print(otherString)
```

Mini exercice: storing strings

Using a variable, write a program that prints your name 5 times in a row.

For instance, mine would ouput:

Noé
Noé
Noé
Noé
Noé

String details

You can also use single quotes `'`:

```
a = 'This is also valid'
```

Convenient if you need to put quote characters *in* the string:

```
print("It's nice!")
print('What is this "string" you speak of?')
```

Strings can be empty:

```
print("")    -- well, that was useful...
```

String concatenation

Concatenate is fancy talk for "stick 'em together". It is done using the `..` operator:

```
print("Hello, " .. "world!")
```

It works, of course, with variables that contain strings:

```
name = "Clarice"
print("Good evening, " .. name)
```

And like the math operations, you can chain as many as you want:

```
name = "Dave"
action = "can't"

print("I'm sorry " .. name .. ", I'm afraid I " ..
    action .. " do that.")
```

Mini exercice: string concatenation

Like before, store your name in a variable. Then complete your program so that it outputs your name three times in a row, but on a single line, with commas and an exclamation mark.

For instance, my program would output:

```
Noé, Noé, Noé!
```

Fun with strings

Like with any operation, you can store the results of concatenation back into variables:

```lua
s = "Building a "
s = s .. "string in several "
s = s .. "steps!"

print(s)
```

Numbers can be concatenated to strings (Lua automatically converts the number into its representation as a sequence of characters):

```lua
numberOfCats = 120 * 2

print("I have " .. numberOfCats .. " cats!")
```

Convenient for outputing results of computations, in a nicely formatted message.

Mini exercice: making output messages

Using the variables `name` and `age`, write a program that outputs something like:

`Hello, my name is Bob, and I am 34 years old.`

(You can lie about your name and your age ☺)

String length

To count the number of characters in a string, we use the # operator:

```
print(#"Hello!")          -- 6
print(#"This is great!")    -- 14
print(#"")                -- 0

-- works with variables that contains string too:

aString = "Yeah!"
print(#aString)           -- 5
```

The length operator # works only on strings:

```
a = 100
print(#a)    -- the length of a number??
```

Note: special characters might count as 2 characters (details are complex)

```
print(#"Å")      -- 2
```

Mini exercise: string length

Complete the following program, so that it prints the text on one line, and the number of characters in the text on the next line:

```
text = "This is an interesting text."

-- your code here
```

String length: just a number, really!

Note: the result of applying # to a string is itself a number, that can be stored and used in computations:

```
text = "This is some text."
length = #text

print("The text is " .. length .. " characters long.")
print("The double of that number is " .. length * 2)
```

Vocabulary reminder: .. and # are *operators*. Using them to combine strings and/or numbers forms *expressions*. Expression have *values*, which are computed when the interpreter encounters them (they are *evaluated*).

Types recap

So far, two kinds ("types") of values: numbers and strings, plus the special nil type (which means "no value").

The math operators (`+ - * /`) take numbers and produce new numbers:

```
10 + 20      -- the type of this expression is number
```

The concatenation operator `..` takes strings or numbers, and produces new strings:

```
"Begining and " .. "end" -- the type of this expression is string
"The result is " .. 100  -- the type of this expression is string
```

And the length operator `#` takes a string, and produces a number:

```
#"hello"      -- the type of this expression is number
```

Values have *types*: "what kind of data is this?". By extension, we say that expressions (which have values), and variables (which contain values) also have types.

Truth and logic

> The truth is rarely pure and never simple. (Oscar Wilde)

Comparing numbers

Checking if numbers are equal or different:

Comparison	Math notation	Symbol
Equal to	=	==
Not equal to	≠	~=

```
print(5 == 5)         -- true
print(4 ~= 10)        -- true
print(3 * 4 == 13)  -- false
```

Like any operators, `==` and `~=` also work on variables:

```
number = 10
print("Is the number equal to 100?")
print(number == 100)
```

The `==` and `~=` operators return a truth value: "is it true, or is it false?": it is the "boolean" type.

Booleans

Another type in Lua, named after mathematician and logician George Boole.

- numbers can take values like 1, -4, 5.5, -90.3, etc.
- strings can take values like "Yeah", "Super duper" or "I have 2 cats!"
- booleans can take exactly two values: true or false

Booleans are values like any other. You can print them:

```
print(3 * 10 > 100)     -- false
```

Store them in variables and retrieve them:

```
a = 5
b = 5
aVariable = a == b
print(aVariable)     -- true
```

And operate on them (later).

`true` and `false`

Just like:

- we can put numbers directly in the source (`a = 5`)
- we can put strings directly in the source (`print("Super!")`)

we can put booleans directly in the source, using keywords `true` and `false`:

```lua
print(true)     -- true
print(false)    -- false
a = true
print(a)        -- true
a = false
print(a)        -- false
```

Why we'd want to do that will become clear later :) Basically, any "yes/no" question translates somehow as a boolean in code.

Note: values that appear directly in the source are called "literals"

Ordering numbers

Checking which of two numbers is smaller or bigger:

Comparison	Math notation	Symbol
Smaller than	<	<
Greater than	>	>
Smaller or equal	≤	<=
Greater or equal	≥	>=

```
print(4 < 5)       -- true
print(4 > 5)       -- false
print(20 < 20)     -- false
print(10 <= 2 * 5) -- true
print(4 <= 10)     -- true
```

Vocab recap: `== ~= < > <= >=` are *operators*, they combine variables and numbers into *expressions*. Specifically, they take numbers and produce booleans (truth values).

Mini exercices: comparisons

Use `print()` and the comparison operators to check if the following are true:

- If I have 23 boxes, and each box contains 45 cards, I have more than 1000 cards.
- A company had 180 employees. In the next 4 years, they hired 215 employees per year. Did they end up with less than 2000 employees?
- Is 215 divided by 5 equal to 43?

Recap #2

We've now seen all the basic *types* in Lua:

- numbers
- strings
- booleans
- nil

and how to write them in the source code (*literals*)

```
5    -6.5
"A string"   'Another string'
true     false
nil
```

Plus a bunch of operators that can can operate on them:

```
+ - * / == ~= < > <= >= .. #
```

Equality

`==` and `~=` work with all the types, not just numbers:

```
print(5 == 5)            -- true
print("Foo" == "Foo")    -- true
print("Foo" ~= "Bar")    -- true
print(true == true)      -- true
print(true == false)     -- false
```

Values of different types are never equal, though:

```
print(10 == "10")        -- false
```

Comparing to `nil` can be used to check if a variable is empty:

```
print(a == nil)      -- true
a = 100
print(a == nil)      -- false
```

Mini exercise: equality

Complete the following program to check if the username is Bob:

```
username = "Bob"

-- your code here
```

Control flow

The essence of programming

"Lua executes the instructions one by one, from the first to the last."

Well... not always. We can also choose what parts of the code to execute, depending on conditions. Typically:

- dealing with different inputs in different ways
- repeating instructions many times
- skipping unnecessary instructions in certain cases
- stopping the program early

Most *algorithms* (a plan to solve a specific problem) rely on this. Without control flow, a program would always execute exactly the same way, couldn't be interactive, etc.

`if` control structure

The `if then end` keywords let you choose whether a sequence of instructions (a *block* of code) will be executed, according to a condition:

```
number = 10

print("Let's test our number.")

if number < 100 then
    print("The number is less than 100!")
    print("Isn't that crazy?")
end

print("Well, that was fun.")
```

Whether the two middle `print()`s will be executed depends on the content of the variable `number`.

`if` syntax

```
if expression then
    -- block
end
```

It means: "If the expression is true, then execute the block of code, otherwise skip it (go directly to the next instruction after the `end` keyword)"

Booleans as conditions

The expression we give as condition should be of type boolean:

- typically, a comparison with one of the following operators: `==` `~=` `<` `>` `<=` `>=`
- that's the main use of booleans

```
condition = true

if condition then
    print("If we are here, it's because condition is true")
end
```

It's interesting mostly when the condition is related to a variable (testing an input, the current state of the program, etc.)

Optional instructions

One possible use: settings

```
printResult = true

-- some calculation
result = 100

-- print the result only if we want to
if printResult then
    print(result)
end
```

Mini exercice: if

Complete the following program so that it prints "a is smaller than b" when the number in `a` is smaller than the one in `b`:

```
a = 10
b = 3

-- your code here
```

Then, add to your program a second test, which will print "a is bigger than 20" when the number in `a` is bigger than 20.

Blocks of code

A block of code can be any number of instructions, including none:

```lua
if someCondition then
    -- Do nothing! That's really useless!
end
```

and other `if`s:

```lua
number = 10

if number < 100 then
    print("Number is less than 100")
    if number >= 0 then
        print("Number is also more than 0!")
    end
end
```

Coding style: indenting. When starting a new block, *always* add an indentation (using tab key). When finishing a block (with the `end` keyword), remove an indentation.

Mini exercise: nesting `if`s

Given a variable `n` that contains a number, write a program that outputs :

```
n is between 20 and 200
```

if `n` is indeed bigger than 20 and smaller than 200.

`else` block

Can add an `else` block, to execute when the condition is false:

```lua
age = 25

if age < 18 then
    -- reduction for youngsters
    price = 50
else
    -- full price for adults
    price = 100
end

print("When you are " .. age .. " years old, you pay " ..
    price .. "€")
```

`else` syntax

```
if condition then
    -- block to execute if condition is true
else
    -- block to execute if condition is false
end
```

It means: "If condition is true, execute the first block. Otherwise, execute the second block".

In any case, only one of the two blocks is executed, and then the program continues after the `end`.

Mini exercise: `else`

Given a variable `country` containing a string, print

`Wine and cheese`

if `country` is France, and print

`Some other food`

otherwise.

`elseif` block

Can add `elseif` block(s), to check for more conditions, in case the first one(s) fail:

```
username = "Alice"

if username == "Alice" then
    print("Nice to see you, Alice!")
elseif username == "Bob" then
    print("Oh hello, Bob!")
elseif username == "Charlie" then
    print("Hey, it's you, Charlie!")
else
    print("I don't know who you are, " .. username)
end
```

Note: there can be any number of `elseif` blocks, plus an optional `else` block at the end. In any case, only one of the blocks will execute, and the program will then continue after the `end`.

`elseif` syntax

```
if condition1 then
    -- block 1
elseif condition2 then
    -- block 2
elseif condition3 then
    -- block 3
else
    -- else block
end
```

It means: "If condition1 is true, execute block 1. If it's false, but condition2 is true, exectute block 2. If that's also false, but condition 3 is true, execute block 3. If they were all false, execute the else block."

Reminder: `else` and `elseif` blocks are optional.

Mini exercise: `elseif`

Given a variable `number` containing a number, write different informative outputs depending on whether:

- `number` is negative (smaller than 0)
- `number` is between 0 and 100
- `number` is between 100 and 1000
- `number` is bigger than 1000

Numeric `for` loop

The `for do end` keywords let you repeat a block of code a certain number of times.

```lua
for i = 1,10 do
    print("Hello!")
end
```

Each time the block is executed, the variable `i` will take the next value, starting at 1 and going up to 10:

```lua
for i = 1,10 do
    print("Executing the block")
    print("This is iteration number " .. i)
end
```

`for` loop: syntax

```
for variable = start,finish do
    -- block
end
```

`variable` should be an unused variable (or one you don't need anymore), `start` and `finish` should be expressions of type number.

In real life:

- as a shortcut instead of writing the same instructions many times
- apply the same operation to a lot of different data

Mini exercices: `for`

- Write a program that prints "Programming is fun!" 20 times.
- Write a program that outputs all numbers from 34 to 88.
- Write a program that, given a variable `N` that contains a number, outputs "I'm in a loop!" `N` times.

Inside the loop

Like in `if`, the block in the `for` loop can be any number of instructions, including:

- `print()` instructions
- assignments (`a = 4`)
- `if` control structures
- other `for` loops!

```lua
for i = 1,10 do

    j = i * 2
    k = i * 3

    print("i is " .. i)
    print("Its double is " .. j)
    print("And its triple is " .. k)
end
```

Inside the loop 2

```lua
-- testing many numbers in a row:

for i = 10,50 do
    print("Let's test " .. i)
    if i < 25 then
        print(i .. " is smaller than 25")
    elseif i == 25 then
        print(i .. " is equal to 25")
    else
        print(i .. " is bigger than 25")
    end
end

-- adding numbers many times
-- (a slightly dumb way to compute 10 * 5)

sum = 0
for i = 1,10 do
    sum = sum + 5
end
print(sum)
```

Nesting loops

Printing all possible pairs of numbers between 1 and 5:

```lua
for i = 1,5 do
    for j = 1,5 do
        print(i .. "," .. j)
    end
end
```

Mini exercices: more loops

- For a given number `N`, write a program that computes the sum of all numbers up to `N`. For instance, with `N = 6`, it would compute 1 + 2 + 3 + 4 + 5 + 6, which is 21.
- Given a string `username` and a number `N`, write a program that ouptuts "Hello" followed by the user name `N` times, on one line. For instance with `username = "Bob"` and `N = 6`, the output would be `Hello Bob Bob Bob Bob Bob Bob!`
- Given a number `N`, write a program that outputs all numbers from `N` to 100, or from 100 to `N`, depending on which is bigger.
- For all numbers from 1 to 100, print whether they are smaller than 35, between 35 and 75, or bigger than 75.
- Write a program that finds all pairs of integers A and B, smaller than 1000, such that A × A + B × B = 40000

Fun with loops

Loops are what gives a language its computational power to solve *any* problem in existence!

- `if`: skip code, ie. jump forward
- `while`: repeat code, ie. jump backwards

They modify the *control flow* of the program. ("Control" means "what line of code are we executing now?")

> You are now a programmer!

Bonus stuff

Less important, but useful

More math operators

Modulo operation (or "remainder of Euclidean division") using the `%` operator. Example: I have 10 pieces of pie, to share evenly for 3 persons. Each person gets 3 pieces, and 1 piece is left. 10 modulo 3 is 1.

```
print(10 % 3)        -- 1
```

Uses:

- when `a % b == 0`, we say that "b divides a" or "a is a multiple of b".
- when `a % 2 == 0`, a is even, otherwise it is odd.

Note: has nothing to do with percents, despite the symbol used.

This is surprisingly useful. I swear!

Mini exercise: modulo

Use a program to answer the following questions:

- Is 2546 divisible by 9?
- Can we split 77 toys in 6 children evenly? If not, how many toys are left?
- I am facing North, and make 23 quarter turns to the right. What direction am I facing now?
- If today is Tuesday, what day will it be in 200 days?

Then, write a program that finds all the divisors of 360.

Then, given a number N, write a program that counts how many divisors does N have (for instance, 360 has 24 divisors).

Even more math operators

Exponentiation (repeated multiplication): "b power n" or "b to the n-th power", b^n:

$$b^n = \underbrace{b \times \cdots \times b}_{n}$$

Written in Lua with the ^ operator:

```
print(10^4)              -- 10000
print(10 * 10 * 10 * 10)     -- 10000
print(2^10)              -- 1024
```

Mini exercise: solving equations

Find all the integers from -100 to 100 that verify the following equation:

$$N^4 - 23 \times N^3 - 685 \times N^2 + 8999 \times N = -9660$$

`for` loop with a step

In a `for` loop, we can also specify a step (after the start and finish values):

```
-- even numbers
for i = 0,10,2 do
    print(i)
end
```

and go backwards, by using a negative step:

```
-- downwards
for i = 10,1,-1 do
    print(i)
end
```

Breaking out

The `break` instruction: exits a loop early:

```lua
for i = 1,10000 do
    -- some computation
    if foundSolution then
        break    -- this will bring us...
    end
end
-- ... here!
```

Note: `break` always breaks the innermost loop only:

```lua
for i = 1,1000 do
    for j = 2,50 do
        if someReason do
            break    -- this will bring us...
        end
    end
    -- ... here!
end
```

Deep break

To break from several loops at once, you can use a boolean variable that will instruct the outer loops that they should break as well:

```
shouldExit = false

for i = 1,1000 do
    for j = 1,1000 do
        if someReason then
            shouldExit = true
            break
        end
    end

    if shouldExit then
        break
    end
end
```

Mini exercise: `break`

Find the first solution of the following equation (knowing that it's less than 10000)

$144 \times N - N^2 = 5148$

Then, modify your earlier program (the one that finds all pairs of integers A and B, smaller than 1000, such that A × A + B × B = 40000), to stop at the first solution.

Scientific notation

Convenient for very big (or very small numbers):

Decimal	Scientific	Lua
300	3×10^2	`3e2`
7200000	7.2×10^6	`7.2e6`
-533000	-5.33×10^5	`-5.33e5`
0.04	4×10^{-2}	`4e-2`
0.0000034	3.4×10^{-6}	`3.4e-6`

```
print(3e6)          -- 3000000
print(3 * 10^6)     -- 3000000
print(2.5e-4)       -- 0.00025
print(2.5 * 10^-4)  -- 0.00025
print(2e6 + 2)      -- 2000002
```

Exercices

General programming advice

- work iteratively: don't try to solve the whole thing at once, but focus on solving each sub-problem independently, then putting them all together.
- don't try to program in your head: run your code as often as possible, even when you know it's not yet correct. It's free! You will:
  - catch your errors earlier
  - feel good about progressing towards the goal
- avoid "random programming": guessing almost never works, whether it is about syntax (the interpreter is very strict about that), or algorithms. There is always a reason why something works or doesn't. All the information you need is in the slides. Take your time, take a deep breath, you'll be just fine!
- never hesitate to debug your code using `print()`: checking the content of your variables often will show you exactly what is going on in your program, and you will understand better why it works or doesn't. You can remove all your debug `print()`s once the program is correct.

And most importantly: have fun! Experiment. Make up your own problems! These exercises might seem boring or useless, but they'll ensure you understand the bases of programming, before we move on to more fun (and complex) stuff.

Factorial

The factorial of an integer N, denoted N!, is the product of all consecutive integers from 1 to N. For instance:

5! = 1 × 2 × 3 × 4 × 5 = 120

and:

16! = 20922789888000

Write a program that computes the factorial of whatever number is in the variable N, so that for instance:

```
N = 16

-- your code here
```

would output 20922789888000

Roadtrip song

```
99 bottles of beer on the wall, 99 bottles of beer,
Take one down and pass it around, 98 bottles of beer on the wall.
98 bottles of beer on the wall, 98 bottles of beer,
Take one down and pass it around, 97 bottles of beer on the wall.
...
1 bottle of beer on the wall, 1 bottle of beer.
Take one down and pass it around, no more bottles of beer on the wall.
No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.
```

Write a program that prints the lyrics to the song "99 bottles of beer".

Possible steps:

- write a program that writes numbers from 99 to 1
- modify it to write the first line of each verse
- add the second line of each verse
- find a way to handle the last 2 verses, which are slightly different than the rest

Fibonacci sequence

> 1, 1, 2, 3, 5, 8, 13, ...

A sequence of integers starting with two ones, and where each number after that is the sum of the two previous ones.

Write a program that computes the first 100 terms of the Fibonacci sequence.

Possible steps:

- program the first few steps without a loop, by using new variables every time
- see if you can rewrite this using less variables, by reusing the ones you don't need anymore
- "automatize" the process by using a loop

Drinking song

> " 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizzbuzz, 16, 17, Fizz, 19, Buzz, Fizz, 22, 23, ...

Write a program that ouputs the numbers from 1 to 100, replacing multiples of 3 with "Fizz", multiples of 5 with "Buzz", and multiples of both with "Fizzbuzz".

Reminder: a is a multiple of b when `a % b == 0`

Possible steps:

- write a program that outputs all numbers from 1 to 100
- modify it to also print "Fizz" when the number is a multiple of 3
- modify it to also print "Buzz" when the number is a multiple of 5
- modify it to *not* print the number when printing "Fizz" or "Buzz"
- complete the exercise

Hint: don't hesitate to use debug instructions to check what is going on. Typically, `print(i)` in a `for` loop, to know what iteration you are at.

Prime numbers

A prime number is an integer divisible only by 1 and itself. The first few prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23...

Write a program that outputs all the prime numbers smaller than 1000.

Possible steps:

- write a program that checks if a number `N` is a prime number
- put this into a loop

Reminder: a is divisible by b when `a % b == 0`

Hint: a number is *not* prime if it is a multiple of 2. Not either if it is a multiple of 3. Or 4. Or 5. And so on. It is not prime if it is a multiple of *anything* (except 1 and itself). And a yes/no question tends to translate as a boolean.

Bonus question: modify your program to print the first prime number greater than one million.

Bonus question 2: modify your program to print the 2000th prime number

Compound interests

To pay for your retirement, you put 500 euros every month in a savings account that grows by 8% per year, compounded every month. Write a program that simulates the content of your savings account during the next 50 years.

> ❛   ♫ I don't care too... much for money, for money can't buy me love! ♫

Hint: if the annual growth rate is `yearlyRate`, the monthly growth rate is `monthlyRate = (1 + yearlyRate)^(1/12) - 1`