Lua 103

Awesomeness never stops!

Contents

Logical operators

> “ Logic will get you from A to B. Imagination will take you everywhere. (Albert Einstein)

Exercise: and

Write a function `logical_and()` that, given too boolean parameters, returns true if both parameters are true (and false otherwise):

```
function logical_and(a,b)
    -- your code here
end

print(logical_and(false, false))  -- false
print(logical_and(false, true))   -- false
print(logical_and(true, false))   -- false
print(logical_and(true, true))    -- true
```

Solution: and

```
function logical_and(a,b)
    if a then
        if b then
            return true
        end
    end
    return false
end
```

or shorter:

```
function logical_and(a,b)
    if a then
        return b
    else
        return false
    end
end
```

Exercise: or

Write a function `logical_or()` that, given too boolean parameters, returns true if at least one parameter is true (and false otherwise):

```
function logical_or(a,b)
    -- your code here
end

print(logical_or(false, false))  -- false
print(logical_or(false, true))   -- true
print(logical_or(true, false))   -- true
print(logical_or(true, true))    -- true
```

Solution: or

```
function logical_or(a,b)
    if a then
        return true
    elseif b then
        return true
    end

    return false
end
```

Or shorter:

```
function logical_or(a,b)
    if a then
        return true
    else
        return b
    end
end
```

Exercise: not

Write a function `logical_not()` that, given true, returns false, and vice versa:

```
function logical_not(a)
    -- your code here
end

print(logical_not(true))   -- false
print(logical_not(false))  -- true
```

Solution: not

```
function logical_not(a)
    if a then
        return false
    else
        return true
    end
end
```

Or shorter:

```
function logical_not(a)
    return a ~= true
end
```

Logical operators: `and`, `or`, `not`

These three logical operations exist as the operators `and`, `or`, and `not`:

```
print(false and false)  -- false
print(false and true)   -- false
print(true and false)   -- false
print(true and true)    -- true

print(false or false)  -- false
print(false or true)   -- true
print(true or false)   -- true
print(true or true)    -- true

print(not false)       -- true
print(not true)        -- false
```

In (mathematical) logic:

- "A and B" means "both A and B are true"
- "A or B" means "either A or B (or both) are true"
- "not A" means "A is not true"

Control flow logic

These operators take booleans, and form boolean expressions. As such, they can be used as conditions in `if`, and `while`:

```
N = 50
if N % 2 == 0 and N > 100 then
    print(N .. " is odd, and bigger than 100!")
end
```

They can use variables:

```
N = 100
isOdd = N % 2 == 0
isBig = N > 50

if isOdd and isBig then
    print(N .. " is odd, and bigger than 50!")
end
```

Saving booleans

...and their results can be saved in variables:

```
isRaining = true
haveUmbrella = false

willGoForAWalk = (isRaining and haveUmbrella) or not isRaining
```

Like other operations, they can be chained as much as you want:

```
if a and b and c and d then
    print("a, b, c and d are all true!")
end

if a or b or c or d then
    print("At least one of a, b, c or d is true!")
end
```

Mini exercise: logical operators

Complete the following program so that it prints all elements of the array that:

- start or end with the letter "a"
- contain more than 5 letters and can be cut in three equal parts

```
words = {"avocado", "burrito", "enchilada",
    "guacamole", "tequila", "mojito", "ola"}

-- your code here
```

Tables

> A single conversation across the table with a wise man is better than ten years of study. (Chinese proverb)

Tables

Arrays are a special case of a more general type: the table. A table associates *keys* with *values*. Syntax to set or get elements of a table is the same as arrays (because arrays *are* tables): `name[key]`

```
age = {}       -- create empty table
age["Alice"] = 10
age["Bob"] = 5
age["Charlie"] = 22
```

Key	Value
Alice	10
Bob	5
Charlie	22

Types in tables

Can mix types as much as wanted:

```
t = {}
t["Hello"] = 10
t[2.5] = true
t[false] = "Whatever"
```

Key	Value
"Hello"	10
2.5	true
false	"Whatever"

Note: the only thing you cannot do is use nil as a key.

Retrieving from tables

To get an element, same syntax as arrays: `table[key]`:

```
t = {}
t["Hello"] = 10
t[2.5] = true
t[false] = "Whatever"
t["Foo"] = "Bar"

print(t["Hello"])
print(t[2.5])
print(t[false])
print(t["Foo"])
```

Keys

Just like indices in arrays, the key can be any expression (including using a variable or an operation):

```
t = {}
key = "Foo"
t[key] = 23
t[key .. key] = -2.4

print(t[key])       -- 23
print(t["FooFoo"])  -- -2.4
```

Just like in arrays, trying to access an element which is not in the table will return nil:

```
t = {}
print(t["Blah"])    -- nil
```

Erasing elements

...and erasing an element from the table can be done by assigning nil:

```
t = {}
t[3.5] = false
print(t[3.5])   -- false
t[3.5] = nil
print(t[3.5])   -- nil
```

And checking the presence of a key/value pair is checking that the value is non nil:

```
if t[key] ~= nil then
    print("There is a value for key " .. key)
end
```

Table recap

Not much is new! Works the same as arrays, except that the index (or "key") can be of any type. Arrays are simply tables were all keys are positive integers.

Reminder: when all keys are consecutive positive integers, we have "special" things that we can do with the table:

- use the length operator `#`
- use `table.insert()` and `table.remove()`
- use `table.concat()`

But really, we've been using tables all along!

Mini exercises: tables

- Using a table, write a program that greets a user according to their name. If it's Alice, say "Hello Alice", if it's Bob, say "Oh hi, Bob!", and if it's Charlie, say "Charlie, so nice to see you!". If it's someone else, just say "Welcome".

- Fill a table `age` with the following information, and write a function that prints results such as "You are Alice, you are 22 years old.", or an error message if the person is unknown:

Key	Value
Alice	22
Bob	5
Charlie	65
Dorothy	34
Emily	50

Table literal

Syntax to create a table directly with key/value pairs: `{[key] = value, [key] = value, ...}`

```
t =
{
    ["Alice"] = 3,
    [true] = 2.4,
    [-4] = 3
}

print(t[true])  -- 2.4
```

Here too, keys and values can be any expression (as long as keys don't evaluate to nil).

Reference type

Like the array before, tables are a *reference type*: variables do not contain the table, only a reference to it. Several variables can refer to the same table, etc.

```
a = { ["Foo"] = "Bar" }
b = a
print(a["Foo"], b["Foo"])   -- Bar  Bar
a["Foo"] = 245
print(b["Foo"])     -- 245
```

It's exactly the same as arrays, because arrays *are* tables.

Generic `for` and `pairs()`

To iterate over all the key/value pairs in a table: the generic `for` loop:

```
t = {}
t["Hello"] = 10
t[2.5] = true
t[false] = "Whatever"
t["Foo"] = "Bar"

for key,value in pairs(t) do
    print(key, value)
end
```

Output:

```
Hello   10
false   Whatever
2.5 true
Foo Bar
```

Generic `for` syntax

```lua
for key,value in pairs(table) do
    -- use key and value
end
```

- `key` and `value` are variable names (often `k,v`), `table` is an expression of type table.
- there is no order in a table. Using `for` and `pairs()` will not necessarily iterate in the same order that the values were added to the table.
- the variables `key` and `value` are local to the body of the loop. They will shadow any outer variable with the same names, and they stop existing after the loop exits.
- like in any loop, you can use `break` to exit immediately

Mini exercise: generic `for` loop

- using the same age table as before, write a program that outputs sentences such as "Hello Alice, you are 22 years old!" for all persons in the table
- write a function that takes a table and another parameter `k`, and returns true if `k` is in the table *as a key*
- write a function that takes a table and another parameter `v`, and returns true if `v` is in the table *as a value*
- write a function that takes a table and returns the number of key/value pairs in it

Arrays are tables!

Reminder: arrays *are* tables. We just restricted the keys to positive integers:

```
array = {"a", "b", "c", "d", "e"}

-- is exactly the same as
array =
{
    [1] = "a",
    [2] = "b",
    [3] = "c",
    [4] = "d",
    [5] = "e"
}

-- which is the same as
array = {}
array[1] = "a"
array[2] = "b"
array[3] = "c"
array[4] = "d"
array[5] = "e"
```

Arrays are tables!!!

Since arrays *are* tables, the generic `for` loop works on arrays too:

```
array = {"a", "b", "c", "d", "e"}

for k,v in pairs(array) do
    print(k,v)
end
```

Output:

```
1   a
2   b
3   c
4   d
5   e
```

Reminder: when using `for` and `pairs`, the order is never guaranteed. The only certain thing is that you will iterate over all the key/value pairs...

Arrays and `pairs()`

... on the other hand, it's convenient for sparse arrays (arrays with gaps):

```
a = {}
a[2] = "Two"
a[1000] = "One thousand"

for k,v in pairs(a) do
    print(k,v)
end
```

Arrays and `ipairs()`

For tables that use consecutive positive integers as keys (aka arrays), use the generic `for` loop with `ipairs()` (notice the i for "integer" or "index"):

```
array = {"a", "b", "c", "d", "e"}

for index,value in ipairs(array) do
    print(index,value)
end
```

- this one is guaranteed to go over the elements in order
- `index` and `value` are also local variables, often `i,v`
- doesn't play well with gaps (might stop at the first nil it finds)
- basically a shortcut for:

```
for index = 1,#array do
    local value = array[index]
    -- block
end
```

Mini exercise: `ipairs()`

Using the generic `for` loop and `ipairs()`:

- write a function that returns true if an element `e` is in an array
- write a function that returns the smallest element of an array of numbers
- write a function that returns the average of all elements of an array of numbers

Multiple assignments

> “ The love of beauty in its multiple forms is the noblest gift of the human cerebrum. (Alexis Carrel)

Multiple assignment

So far we've always done assignment with one variable on the left, and one expression on the right:

```
var = 10
```

In Lua, we can assign *multiple* values to *multiple* variables at the same time, by separating them with commas:

```
var1, var2, var3 = 10, "Hello", true
print(var1, var2, var3) -- 10  Hello   true
```

Evaluation order

All the expressions on the right are evaluated first, and then the values are assigned to the variables.

Convenient for swapping values:

```
a, b = "foo", "bar"
print(a,b)
a, b = b, a
print(a,b)
```

Scope

Writing `local` before the variables will declare all the variables as locals:

```
a,b,c = 1,2,3

if true then
    local a,b,c = 10,20,30
    print(a,b,c)    -- 10   20  30
end

print(a,b,c)    -- 1    2   3
```

Multiple return values

Functions can return multiple values too!

```
function doubleAndHalf(n)
    return n*2, n/2
end
```

To get all the return values, we put several variables on the left side of the assignment:

```
double, half = doubleAndHalf(10)
print(double, half)     -- 20   5
```

If you put N variables on the left, you get only the first N return values. If you put more variables than there are return values, the additional ones get nil:

```
double = doubleAndHalf(10)
print(double)       -- 20
a,b,c,d = doubleAndHalf(10)
print(a,b,c,d)      -- 20  5   nil nil
```

Standard library

Some built-in functions return multiple values:

```
integral, fractional = math.modf(21.56)
print(integral, fractional)     -- 21    0.56

first, last = string.find("This is cool", "cool")
print(first, last)      -- 9    12
```

Mini exercises: multiple return values

- modify your "smallest element of an array of numbers" function, so that it returns both the smallest number, and the index it was found at
- write a function that returns the first and last elements of an array

String manipulation

> “ There is geometry in the humming of the strings, there is music in the spacing of the spheres. (Pythagoras)

Fun with strings

So far, we know how to:

- get the length of a string: `#s`
- concatenate strings to build bigger strings: `a .. b`
- get the `i`-th character in a string: `string.sub(s, i, i)`

There is a lot more we can do!

Substring

To extract a part of a string (a "SUBstring"): `string.sub(s, i, j)`

> Returns the substring of `s` that starts at `i` and continues until `j`

```
s = "Hello, this is dog!"

print(string.sub(s, 1, 5))         -- Hello
print(string.sub(s, 8, 11))        -- this

-- the whole thing!
print(string.sub(s, 1, #s) == s)   -- true
```

Find

To look for a certain substring in a string, `string.find(s, pattern)`

> Looks for the first match of pattern in the string `s`. If it finds a match, then `find` returns the indices of `s` where this occurrence starts and ends; otherwise, it returns nil.

```
s = "This is a beautiful day"

first,last = string.find(s, "is")
print(first,last)       -- 3   4

first,last = string.find(s, "beautiful")
print(first,last)       -- 11  19

first,last = string.find(s, "dinosaur")
print(first,last)       -- nil nil
```

Note: cannot (yet) look for the characters `^$()%.[]*+-?`. We'll see later why.

Find: start index

`string.find()` can take an additional parameter to indicate where to start the search: `string.find(s, pattern, init)`

```
s = "This is a beautiful day"

first,last = string.find(s, "is")
print(first,last)       -- 3   4

first,last = string.find(s, "is", 5)
print(first,last)       -- 6   7
```

Replace

Search and replace: `string.gsub(s, pattern, repl)` (`gsub` stands for Global SUBstitution)

> Returns a copy of `s` in which all occurrences of `pattern` have been replaced by a replacement string `repl`. `gsub` also returns, as its second value, the total number of matches that occurred.

```
s = "We're having coffee, because coffee is nice!"

fixed, num = string.gsub(s, "coffee", "fun")
print(fixed)
print("Replaced " .. num .. " times")
```

Optional additional parameter `n`: replace only the first `n` occurences:

```
print(string.gsub("blah blah blah", "blah", "meh", 2))
```

Mini exercises: strings

- write a function that takes a string, and returns the array of all its letters: passing "Foo" would return `{"F", "o", "o"}`.
- write a function that takes a string, splits it into pieces of length 3, and returns them in an array. For instance, passing "What the hell?" would return `{"Wha", "t t", "he ", "hel", "l?"}`.
- write a function `find_all(s, sub)` that prints all the positions of a substring `sub` in a string `s`. For instance, `find_all("doo bi doo bi doo", "doo")` would print "1 3", "8 10" and "15 17".
- write a function that takes a string in which it replaces "dog" by "cat" and "awesome" by "evil", so that "All dogs are awesome" would become "All cats are evil".

Patterns

In `string.find()` and `string.gsub()` the patterns can include special characters called "character classes". The simplest one is the dot (`.`) which means "any character":

```
s = "I got a book, what did you get?"

result = string.gsub(s, "g.t", "BLAH")
print(result)       -- I BLAH a book, what did you BLAH?
```

The pattern `"g.t"` means: the letter "g" followed by any letter, followed by "t". This pattern *matches* both "get" and "got".

```
s = "Bambi is a deer"
print(string.find(s, "d..."))           -- 12   15
print(string.find("Adorable!", "d...")) -- 2    5
```

The pattern `"d..."` matches a "d" followed by any three characters.

Patterns

```
s = "Some interesting string"

print(string.gsub(s, ".st.", "X"))
-- output: Some interXngXing    2

print(string.gsub(s, ".....", "X"))
-- output: XXXXing   4

print(string.gsub(s, "ing.", "X"))
-- output: Some interestXstring 1
```

More character classes

- `%a`: letters
- `%d`: digits
- `%l`: lower case letters
- `%p`: punctuation characters
- `%s`: space characters
- `%u`: upper case letters
- `%w`: alphanumeric characters

```
s = "Today is 12/09/2015, and it is 13:45; super!"

print(string.gsub(s, "%l", "x"))
--> Txxxx xx 12/09/2015, xxx xx xx 13:45; xxxxx!    18
print(string.gsub(s, "%p", "X"))
--> Today is 12X09X2015X and it is 13X45X superX    6
print(string.gsub(s, "%w", "X"))
--> XXXXX XX XX/XX/XXXX, XXX XX XX XX:XX; XXXXX!    31

print(string.gsub(s, "%as", "X"))
--> Today X 12/09/2015, and it X 13:45; super!  2
print(string.gsub(s, "%si", "XX"))
--> TodayXXs 12/09/2015, andXXtXXs 13:45; super!    3
```

More interesting patterns

```
s = "Today is 12/09/2015, and it is 13:45; super!"

first,last = string.find(s, "%d%d/%d%d/%d%d%d%d")
if first ~= nil then
    print("Found date: " .. string.sub(s, first, last))
end

first,last = string.find(s, "%d%d:%d%d")
if first ~= nil then
    print("Found time: " .. string.sub(s, first, last))
end
```

`string.match()`

When more interested in *what* matched rather than *where*, use `string.match(s, pattern)`. It will return the first substring that matches the pattern, or nil if nothing was found.

```
s = "Today is 12/09/2015, and it is 13:45; super!"

date = string.match(s, "%d%d/%d%d/%d%d%d%d")
if date ~= nil then
    print("Found date: " .. date)
end

time = string.match(s, "%d%d:%d%d")
if time ~= nil then
    print("Found time: " .. time)
end
```

Like `string.find()`, it can also take a third parameter `init` to specify at what index to start searching.

Anchors

When a pattern starts with the special character `^`, it will match only at the *start* of a string:

```
s = "I see 25 planes!"
print(string.match(s, "%d%d"))  -- 25
print(string.match(s, "^%d%d")) -- nil
print(string.match(s, "^...."))  -- I se
```

When a pattern ends with the special character `$`, it will match only at the *end* of a string:

```
s = "I see 25 planes!"
print(string.match(s, "s."))  -- se
print(string.match(s, "s.$")) -- s!
```

You can use both to make sure that the pattern matches the whole string:

```
print(string.match("foo5", "%a%d"  ))   -- o5
print(string.match("foo5", "^%a%d$"))   -- nil
print(string.match("f2",   "^%a%d$"))   -- f2
```

Pattern escapes

We have seen that in a pattern, some characters have a special meaning:

- `.`: matches any character
- `%`: starts a character class, like `%a` for letters
- `^`: matches only at the beginning of a string
- `$`: matches only at the end of a string

There are others (we'll see later what they mean): `()[]*+-?`

To match any of these characters in a pattern, put a `%` before. This means "I don't want the special meaning, just the character itself".

```
s = "Was the 34$ price raised by 20% this year? Harsh..."

print(string.match(s, "%d%d%%"))    -- 20%
print(string.match(s, "%d%d%$"))    -- 34$
print(string.match(s, "....%?"))    -- year?
print(string.match(s, "%.%.%."))    -- ...
```

Mini exercises: patterns

- write a function that checks if a string starts with a capital letter.
- write a function that checks if a string ends with two digits.
- Estonian license plates have the following format: ABC 123 (three capital letters, a space, and three numbers). Write a function that checks whether a string is a valid Estonian license plate.
- Check if a string contains a date in any of the following formats: dd/mm/yyyy, yyyy-mm-dd, dd.mm.yyyy

Optional character

In a pattern, a character followed by `?` is optional:

```
s = "One cat, many cats, yay!"
print(string.gsub(s, "cats?", "X"))

--> One X, many X, yay! 2
```

The pattern `"cats?"` means "the letters c, a, t, possibly followed by an s". It matches both "cat" and "cats".

```
s = "I tend to lose my loose t-shirt."
print(string.gsub(s, "loo?se", "X"))

--> I tend to X my X t-shirt.   2
```

In other words, the pattern `x?` matches 0 or 1 `x`.

Character repetition

In a pattern, a character followed by `+` matches 1 or more times this character:

```
s = "Here is a beautiful baby called Abby Obbbberty."
print(string.gsub(s, "b+", "X"))

--> Here is a Xeautiful XaXy called AXy OXerty. 5

s = "I said damn, daaaamn daaaaaaaamn!"
print(string.gsub(s, "da+mn", "X"))

--> I said X, X X!  3
```

Character repetition

In a pattern, a character followed by `*` matches 0 or more times this character:

```
s = "I lost the stupid soot."
print(string.gsub(s, "so*t", "X"))

--> I loX my Xupid X.   3

s = "Really, kids think lol and looooool are words."
print(string.gsub(s, "lo*l", "X"))

--> ReaXy, kids think X and X are words.    3
```

The `-` modifier is similar, but matches the *smallest* possible string:

```
s = "I lost the stupid soot."
print(string.gsub(s, "l.*t", "X"))
--> I X.    1
print(string.gsub(s, "l.-t", "X"))
--> I X the stupid soot.    1
```

Repetitions and classes

Becomes interesting when mixed with character classes:

```
s = "A bunch of words"
print(string.gsub(s, "%a+", "X"))

--> X X X X 4

s = "24 cats, 1 dog and 900 birds"
print(string.gsub(s, "%d+", "X"))

--> X cats, X dog and X birds   3

s = "A fractional number: 243.54 euros"
print(string.gsub(s, "%d+%.%d+", "X"))

--> A fractional number: X euros    1
```

Mini exercises: patterns

- write a function that finds all words of more than 5 letters in a string and returns them in an array.
- write a function that finds in a string all coordinates in the style of the Battleship game (A2, B12, F4, etc.)
- write a function that checks if a string looks like Finnish date: 1.2.2015, 21.8.1987, 15.11.2012, etc. (you don't need to check that the numbers make sense, so 70.24.3541 would be valid).

Patterns: more

There's a lot more you can do with patterns, but this is enough for us at the moment. If you're interested:

http://www.lua.org/manual/5.1/manual.html#5.4.1

Exercise: digital poetry

> “ I would define, in brief, the poetry of words as the rhythmical creation of Beauty. (Edgar Allan Poe)

Our first "big" program

A random text generator!

- read a corpus of text by famous writers
- analyze the words they use
- reproduce the patterns and sequences to generate our own

> She grew absolutely ashamed of herself, whether sinning or sinned against; but straightway upon the brilliant moonlight, but exceedingly monotonous and forbidding; not thou, carpenter; do we fail in latently engendering an element in him hide, too sick of the Tattoo Land?

Corpus

Free ebooks from Project Gutenberg:

- Dracula (Bram Stoker)
- Brother Grimm's tales
- Mody Dick (Herman Melville)
- Pride and Prejudice (Jane Austin)

Get them as raw text here. Unzip them in the same directory as your program.

Loading a text file

Without explanation (uses concepts we haven't learned yet):

```
function read_file(name)

    local file = io.open(name, "r")
    local text = file:read("*a")
    file:close()

    return text
end
```

Check that it works:

```
local text = read_file("dracula.txt")
print(text)
```

Find all words

Write a function `get_all_words(text)` that, given a string, returns an array of all the words in the string.

```
local words = get_all_words("Hello, this is dog!")

for i,word in ipairs(words) do
    print(word)
end
```

```
Hello
this
is
dog
```

Randomness

Knowing that the standard function `math.random(max)` returns a random integer from 1 to `max`, write a function `random_array_element(array)` that, given an array, returns a random element from that array.

```
local words = {"Hello", "this", "is", "dog"}
local word = random_array_element(words)
print(word)
```

might output, for instance: "this".

First attempt: random words!

Combining all the previous functions you wrote, write a function `generate_text(words, n)` which prints a text made of `n` random words from array `words`.

Try it out.

> useful all It ears we for the most not saw I made you plates were cracked went Bilder never far the this a on starting lay in as big he lighthouse us and meant order people parting should he the and coming write and her is of to forehead either Now good a no indeed inquest Omnipotent call ship An spell was windows can and if find Lucy the the soon some have and moonshine s sunshine more instantly his he they When them train man a and will short his least be not from when praising is renting my

First attempt: sad panda

Problems:

- no punctuation
- makes no sense at all

Punctuation

To simplify we will restrict ourselves to: .,;:!?-'

Write a function `is_allowed_punctuation(s)` that, given a string made of a single character, returns true if it is one of the allowed characters:

```
print(is_allowed_punctuation(";"))      --> true
print(is_allowed_punctuation("%"))      --> false
```

Getting the next word

Write a function `get_next_word(text, position)` that, given a string and an integer:

- checks if the character at the given position in the string is an allowed punctuation
  - if so, it returns it
- if it wasn't a punctuation, it checks if there is a word at the given position
  - if so, it returns it
- it there was neither, return `nil`

```lua
local s = "Hey, isn't this cool?"

print(get_next_word(s, 1))   --> Hey
print(get_next_word(s, 4))   --> ,
print(get_next_word(s, 12))  --> this
print(get_next_word(s, 21))  --> ?
```

Getting all words

Rewrite your `get_all_words(text)` function to return all words and punctuation from `text` in an array.

```
local s = "Hey, isn't this cool?"
local words = get_all_words(s)

for i,v in ipairs(words) do
    print(v)
end
```

```
Hey
,
isn
'
t
this
cool
?
```

Second try: with punctuation

Run `generate_text(words, n)` again now that you have punctuation.

> very be and deceive believe those , shut tip both and you . , now along it a his drinkin he be out river to am ground her him all would find I beginning , he swore that during - it it are every should often . has now the the of behind - times open don would know screams and , now , for against sent dear they at or patience you pall a I world kept the As the from memory Dr not the you my himself first and from go already ' to learn the wishes and

Still makes no sense.

Analyzing sequences of words

We will analyze our corpus to check what sequences of words appear. Specifically, we will record all sequences of three words:

> “ But consider your daughters. Only think what an establishment it would be for one of them.

First	Second	Third
But	consider	your
consider	your	daughters
your	daughters	.
daughters	.	Only
.	Only	think
...	...	...

Generating

- start with a random word that starts with a capital letter
- check in the database what word can come after it

Then, repeat:

- consider the last two words written
- check in the database what possible words can come after them
- pick one randomly

Data structure: array of arrays

```lua
local sequences =
{
    {"But", "consider", "your"},
    {"consider", "your", "daughters"},
    {"your", "daughters", "."},
    {"daughters", ".", "Only"}
    -- ...
}
```

Possible, but slow.

- our corpus has 650313 words, so 650311 sequences of three words
- each time we generate the next word, we need to check the whole array of sequences to find one that matches our two previous words. That would be equivalent to going through the entire corpus every time!
- on average ~650000 checks per word: to generate a text of 100000 words, that makes about 65 billions checks!

NOPE! We can do better.

Using tables

To simplify, we will start by dealing with sequences of 2 words: we will create a table where:

- the keys are all the words we found in the books
- the values are arrays of all the words we found following the key

```
local sequences =
{
    ["But"] = {"consider"},
    ["consider"] = {"your"},
    ["your"] = {"daughters"},
    ["daughters"] = {"."}

    -- ...
}
```

As we fill this table up, we will build a database of all the words in the books, and all the possible words that can follow them.

Building the database

Write a function `analyze(words, sequences)`, that given an array `words`, and a table `sequences`, will:

- go through all words in order
- for each word `w`:
  - check if `w` exists as a key in the `sequences` table
  - if not, add it, with an empty array as value
  - add the word after `w` to the array associated to `w`

Building the database

```lua
local sequences = {}
analyze(get_all_words(read_file("dracula.txt")), sequences)
analyze(get_all_words(read_file("pap.txt")), sequences)
analyze(get_all_words(read_file("moby.txt")), sequences)

for k,v in pairs(sequences) do
    print(k .. ": " .. table.concat(v, "|"))
end
```

```
isle: of|!|fort|were|of|,|;|.
preserve: my|silence|my|the|her|,|the|him|much|the|all
hugging: me|a
hunt: him|the|me|out|'|from|,|together|,|whales|,|us|Moby|him|him|of|?|,|;|
writhed: ;|;
emptied: it|line|into|of
perfect: torrent|agony|nervous|desolation|good|symmetry|indifference|indiff
shutters: ,|had|he|were|in|of|;
```

Generating, third attempt!

Now that you have a database of words and the words that can follow them, rewrite the function `generate_text(sequences, n)`:

- Start with a random word that starts with a capital
- For each new word, check from the `sequences` table what words are allowed to follow, and pick one randomly

Repeat until you get `n` words.

> Society might plunge into oneness , art she disguised in Bible language as concerned that went hot fire which calmly rose as getting dark it my trowsers buttons , exultingly - hazard line palms . Gaining the imperturbable godly - Ram ; yours or had served his would old sailor friend Seward keeps feeding he move from eating maniac ; nevertheless obeys ; with emphasis , walk with happiness and heartening them , borrowed the unconscious understandings , solitary hours together !

Not bad!

Longer sequences

To make the result more natural, we will now consider sequences of 3 three words.

- Modify your `analyze(words, sequences)` function to be one level deeper: if `one` and `two` are two consecutive words in the corpus, `sequences[one][two]` should be an array of all possible words that came after `one` and `two`.
- Adapt your `generate(n)` function to use the new `sequences` table.

> “ Sons and Candy to Lord Godalming and Quincey on the water like a strip of alpine land lying along lengthwise in the bottom of my veins to think over the Danube mouth . Upon my word . But that contradiction in the jungle overlays her own room to look about you . Yes , and horizontally suspended the sail - cloth ; and his desperadoes were too much reason to believe . To my grandmother ' s Alley , with here and there may be deemed , under the present report ; and they revolve .

Pretty good! ☺

Polishing

- After the `n` words were generated, continue until you generate a dot, to finish a sentence.
- When printing the words, prevent printing a space before a punctuation, or after ' and -.
  - You can use `io.write()`, which works like `print()`, but doesn't add a new line at the end
- Call `math.randomseed(os.time())` before generating to avoid getting the "same randomness" everytime.

Some cool results I got:

> And every morning she went again into the curve of the bodies and glittering god uprising from the shoal which we now slide into the Oriental seas to witness the capture of a frigate's thought.

> “ Well, then, to fidget about in the midst of the tempest seemed to have prevailed the most animated language of the deck, and together moved out into the eye with the black clouds, high over Kettleness.

and:

> “ It was in such a time when you may perhaps have heard of him from their flushed faces and trembling hands, radiating without end from God?

Notes

- In the sequence arrays (`sequences[first][second]`), words can be duplicated if this particular sequence of three words appeared multiple times in the corpus
  - pros: it conserves the relative frequency of the sequences (a word that appeared often after two others has more chances to be picked)
  - cons: it wastes memory. Keeping the frequency might not be that useful for our purpose.
- We could try with longer sequences (4, 5, ...)
  - pros: would make the text closer to real English
  - cons: there is a risk of rewriting exactly the original books, since longer sequences are more likely to be unique and not have many "follower words"

The mathematical model is called a Markov chain. Actually used in science to predict/simulate future behavior based on past recordings (weather, word suggestion in auto-correct, ...)