Kurs języka Lua

03 – Tekst

Jakub Kowalski

Instytut Informatyki,
Uniwersytet Wrocławski

2017

Plan na dzisiaj

Opowiemy sobie dokładniej o

1. Napisach
   - UTF-8
2. Komunikacji ze światem zewnętrznym
3. Dacie i czasie

Drobna zagadka

```
a = {}
print (a)         --> table: 0054b2d0
a.a = a
print (a.a.a.a) -->
```

Drobna zagadka

```
a = {}
print (a)         --> table: 0054b2d0
a.a = a
print (a.a.a.a) --> table: 0054b2d0
a.a.a.a = 3
```

Drobna zagadka

```
a = {}
print (a)         --> table: 0054b2d0
a.a = a
print (a.a.a.a) --> table: 0054b2d0
a.a.a.a = 3
print (a.a.a.a) -->
```

Drobna zagadka

```
a = {}
print (a)         --> table: 0054b2d0
a.a = a
print (a.a.a.a) --> table: 0054b2d0
a.a.a.a = 3
print (a.a.a.a) --> ERROR: attempt to index
                --> a number value (field 'a')
```

Napisy

Escape characters

Lista specjalnych znaków

- \a – bell
- \b – backspace
- \f – form feed (page break)
- \n – newline
- \r – carriage return
- \t – horizontal tab
- \v – vertical tab
- \\ – backslash
- \" – double quote
- \' – quote
- *ddd*, \x*hhh* (where *ddd* is up to three decimal digits sequence and *hh* represents two hexadecimal digits) – any ASCII character given its code.
- \u{*h*...*h*} (in Lua 5.3) – any UTF-8 character given its code.
- \z – skips all subsequent spaces until first non-space character.

Multiline strings

Wielolinijkowe napisy

- Zaczynają się od `[[` i kończą `]]`.
- nie interpretują *escape characters*
- Podobnie jak przy komentarzach można stosować $[=^n[$, $]=^n]$.
- Jeśli pierwszym znakiem napisu jest znak nowej linii jest on automatycznie pomijany!

```
main = [[
<html>
<head>
 <tile>Example page</title>
</head>
<body>
 ...
</body>
</html>
]]
```

Usage

Długie napisy nie będące tekstem

- Wielolinijkowych napisów nie zaleca się do zapisywania danych nietekstowych
- Zamiast tego, do wygodnego zapisywania długich danych można używać zwykłych napisów w połączeniu z \z.

```
data = "\x00\x01'x02\x03\x04\z
        \x05\x06\x07\x08\x09"
```

Korzystanie z numerycznych wartości znaków

```
print "\u{3b1}\u{3b2}\u{3b3} \065\097 \x41\x61"
--> αβγ Aa Aa
```

Długość napisu

Operator

```
local s = 'Lua'
print (#s) --> 3
print (#'Moon') --> 4
print (#'Księżyc') -->
```

Długość napisu

Operator

```
local s = 'Lua'
print (#s) --> 3
print (#'Moon') --> 4
print (#'Księżyc') -->  9
```

- Operator długości mierzy długość napisu w bajtach!
- Co znaczy, że działa w sposób intuicyjny dla danych binarnych, ale nie dla napisów w UTF-8.

Funkcja/metoda `len` (równoważna #)

```
print(string.len(s)) -- funkcja biblioteki string
print(s:len()) -- metoda zmiennej 's' typu string
print(string.len('Lua')) --> OK
print('Lua':len()) --> parse error!
print(('Lua'):len()) --> OK
```

Biblioteka string

Funkcje biblioteki `string` których pierwszym argumentem jest napis którego dotyczą, można wywałoać alternatywnie jako metodę tego napisu:

```
string.metoda(napis, ...) <--> napis:metoda(...)
```

string.rep (s, n [, sep])

Konkatenuje napis `s` `n` razy, opcjonalnie wstawiając pomiędzy kolejne wystąpienia separator.

```
string.rep('nan', 0) --> ''
string.rep('nan', 3) --> nannannan
string.rep('Lua', 5, ',') --> Lua,Lua,Lua,Lua,Lua
('Lua'):rep(5, ',')       --> Lua,Lua,Lua,Lua,Lua
local str = '<->'
str:rep(2) --> <-><->
```

Biblioteka string

Funkcje biblioteki string zakładają że każdy znak zajmuje jeden bajt. Wszystkie funkcje tworzą kopie i nie modyfikują swoich argumentów.

string.reverse (s)

```
('Long string!'):reverse()      --> !gnirts gnoL
('Księżyc'):reverse()      --> ********
```

string.lower (s)

```
string.lower('Long STRING!') --> long string!
```

string.upper (s)

```
string.upper('Long STRING!') --> LONG STRING!
```

Biblioteka string

string.sub (s, i [, j])

- Zwraca podłańcuch znajdujący się między zadanymi indeksami.
- Indeksy mogą przyjmować wartości negatywne.
- Domyślna wartość ostatniego argumentu to -1 (#s).

```
('1234567'):sub(2) --> 234567
('1234567'):sub(-3) --> 567
('1234567'):sub(4,4) --> 4
('1234567'):sub(2, 5) --> 2345
('1234567'):sub(-1,-1) --> 7
```

Biblioteka string

string.char (...)

Dla zadanej argumentem listy liczb będący kodami znaków zwraca napis składający się z tych znaków.

```
string.char(97)  --> a
string.char(97, 98, 99, 48, 43, 56) --> abc0+8
```

string.byte (s [, i [, j]])

Zwraca (jako wiele wartości) kody znaków podanego napisu od indeksów `i` (domyślnie 1) do `j` (domyślnie `i`).

```
string.byte('abc0+8')          --> 97
string.byte('abc0+8', -1)      --> 56
string.byte('abc0+8', 2, 4)    --> 98 99 48
{('abc0+8'):byte(1,-1)} --> {97,98,99,48,43,56}
```

Biblioteka string

string.format (formatstring, ···)

- Formatuje napis zgodnie z zadanymi dyrektywami konwersji zmiennych
- Dyrektywy formatowania są podobne do C-owego printf
- Format dyrektyw: %; padding; litera formatu, np.
  - d – liczby całkowite dziesiętnie
  - x – liczby całkowite szesnastkowo
  - f – liczby zmiennoprzecinkowe
  - s – napisy
  - q – napis reinterpretowalny przez Lua

```
local d = 14; m = 3; y = 2017
string.format("Today is %02d/%02d/%01d", d, m, y)
--> Today is 14/03/2017
string.format("Today is '%3d/%3d/%5d'", d, m, y)
--> Today is ' 14/  3/ 2017'
('%x or %X'):format(200, 200) --> c8 or C8
```

Biblioteka string – string.format

```
local pi = math.pi
string.format("pi=%f",pi)      --> pi=3.141593
string.format("pi=%.4f",pi)    --> pi=3.1416
string.format("pi=%8.4f",pi)   --> pi=  3.1416
string.format("pi=%08.4f",pi)  --> pi=003.1416
```

```
local str = 'with "quotes" and \n new line'
string.format("1) %s \n2) %q", str, str)
--> 1) with "quotes" and \n new line
--> 2) "with \"quotes\" and \\n new line"
```

Biblioteka string

Na razie omówimy proste użycia funkcji bazujących na wyszukiwaniu i dopasowniu wzorców.

string.find (s, pattern [, init [, plain]])

- `init` wyznacza startowy indeks przeszukiwania (domyślnie 1)
- Funkcja zwraca dwie wartości – indeks początkowy i końcowy poprawnego dopasowania...
- lub `nil` jeśli dopasowanie się nie powiodło

```
string.find('Hello World!', 'World') --> 7   11
string.find('Hello World!', 'world') --> nil
string.find('Hello World!', 'W..l.') --> 7   11
string.find('abc abc abc', 'abc', 2) --> 5   7
```

Biblioteka string

string.gsub (s, pattern, repl [, n])

- gsub ≡ Global SUBstitution
- Zwraca kopię s w której każde (lub pierwsze n) dopasowanie wzorca zostaje odpowiednio zastąpione.
- Zastąpienie może być tekstem, ale także tablicą lub funkcją.
- Funkcja zwraca dwie wartości – zmodyfikowany tekst oraz liczbę dokonanych podstawień

```
string.gsub('Hello World!', 'l', '-')
--> He--o Wor-d!  3
string.gsub('Hello World!', 'l', '-', 2)
--> He--o World!  2
string.gsub('Hello World!', 'xx', '?')
--> Hello World!  0
string.gsub('Password', '.', '*')
--> ********  8
-- (pattern '.' dopasowuje się do każdego znaku)
```

Iterowanie po napisach

W Lua nie ma prostego iteratora `for c in str do` ani `str[i]`.

Podstawowa metoda

```
for i = 1, #str do
    local c = str:sub(i,i)
    -- do something with c
end
```

Korzystając z iteratora

```
for c in str:gmatch"." do
    -- do something with c
end
```

- Bardziej efektywnie.
- Więcej o `gmatch` i patternach powiemy później.

Unicode

- Z funkcji biblioteki `string` część funkcji może mieć problemy z napisami w unicode.
- Dlatego w Lua 5.3 dodana została specjalna biblioteka `utf8` do radzenia sobie w takich sytuacjach.

`utf8.len (s [, i [, j]])`

- Zwraca liczbę znaków w kodowaniu UTF-8 zaczynając od pozycji `i` (domyślnie 1) do pozycji `j` (domyślnie -1).
- Jeśli znajdzie nieprawidłową sekwencję, funkcja zwróci fałsz i pozycję pierwszego nieprawidłowego bajtu.

```
utf8.len('Księżyc')  --> 7
utf8.len('Ka\x93')   --> nil 3
utf8.len('Kα\x93')   -->
```

Unicode

- Z funkcji biblioteki `string` część funkcji może mieć problemy z napisami w unicode.
- Dlatego w Lua 5.3 dodana została specjalna biblioteka `utf8` do radzenia sobie w takich sytuacjach.

`utf8.len (s [, i [, j]])`

- Zwraca liczbę znaków w kodowaniu UTF-8 zaczynając od pozycji `i` (domyślnie 1) do pozycji `j` (domyślnie -1).
- Jeśli znajdzie nieprawidłową sekwencję, funkcja zwróci fałsz i pozycję pierwszego nieprawidłowego **bajtu**.

```
utf8.len('Księżyc')  --> 7
utf8.len('Ka\x93')   --> nil 3
utf8.len('Kα\x93')   --> nil 4
utf8.len('K\x93sieżyc', 4, 6) --> 2
```

Biblioteka utf8

utf8.char (...)

Odpowiednik string.char.

```
utf8.char(75, 115, 105, 281, 380, 121, 99)
--> Księżyc
```

utf8.codepoint (s [, i [, j]])

Odpowiednik string.byte
Również działa na bajtach(!)

```
utf8.codepoint('Księżyc') --> 75
utf8.codepoint('Księżyc', 1, -1)
--> 75  115  105  281  380  121  99
utf8.codepoint('Księżyc', 6, -1) --> 380 121 99
utf8.codepoint('Księżyc', 5, -1) -->
```

Biblioteka utf8

utf8.char (...)

Odpowiednik string.char.

```
utf8.char(75, 115, 105, 281, 380, 121, 99)
--> Księżyc
```

utf8.codepoint (s [, i [, j]])

Odpowiednik string.byte
Również działa na bajtach(!)

```
utf8.codepoint('Księżyc') --> 75
utf8.codepoint('Księżyc', 1, -1)
--> 75  115  105  281  380  121  99
utf8.codepoint('Księżyc', 6, -1) --> 380 121 99
utf8.codepoint('Księżyc', 5, -1) --> ERROR
```

Biblioteka utf8

utf8.offset (s, n [, i])

- Zwraca pozycję w bajtach dla n-tego znaku napisu, zaczynając od i (domyślnie 1 jeśli $n > 0$ i #s+1 wpp.).
- Jeśli $n < 0$ zwraca pozycję n-tego znaku od końca.
- Dla nieprawidłowego znaku lub pozycji zwraca nil.

```
utf8.offset('Księżyc', 5) --> 6
local s = 'Księżyc'
utf8.codepoint(s, utf8.offset(s, 5), -1)
--> 380 121 99
string.sub(s, utf8.offset(s, -4)) --> ężyc
utf8.offset('Księżyc', 11) --> nil
```

Biblioteka utf8

utf8.codes (s)

- Pozwala na iterowanie po napisach w UTF-8.
- Zwraca informacje o początkowym bajcie oraz kodzie każdego ze znaków.

```
for i, c in utf8.codes('Księżyc') do
  print (i, c)
end
--> 1   75
--> 2   115
--> 3   105
--> 4   281
--> 6   380
--> 8   121
--> 9   99
```

I/O

Prosty model I/O

- Prosty model polega na tym, że podmienia domyślne strumienie: wejściowy i wyjściowy
- Pozwala to pisać (czytać) tylko do (z) jednego pliku na raz

output

```
-- zwraca handler do aktualnego outputu
io.output()
-- otwiera plik i ustawia na niego handler
io.output ('tmp.txt')
-- podmienia output handler na zadany file handle
io.output (file)

-- pisze zadaną treść na output
io.write('test', 1, '\ntest2')

-- zamyka output
io.output():close()
```

Prosty model I/O

output – przykład

```
print (io.output()) --> file (75782920)
io.output ('tmp.txt')
print (io.output()) --> file (75782960)
io.write('test', 1, '\ntest2')
io.output():close()
print (io.output()) --> file (closed)
print (io.stdout)   --> file (75782920)
--> File 'tmp.txt':
--> test1
--> test2
```

io.write i print

`print` zawsze wypisuje na stdout i powinien być sotosowany tylko jako quick-and-dirty debug. `io.write` nie dodaje znaków nowych linii, tabulatorów, pozwala przekierowywać output.

Prosty model I/O

input

```
-- działa analogicznie do io.output
io.input( [file])
-- wczytuje dane z aktualnego inputu
io.read(...)
-- zamyka input
io.input():close()
```

przykład

```
io.input ('tmp.txt')
print (io.input()) --> file (75782960)
print (io.read() ) --> test1
io.input():close()
print (io.input()) --> file (closed)
```

Wczytywanie ze strumienia

`file:read (···)`

5.0/5.1	5.2	5.3	Wczytuje:
`'*n'`	`'*n'`	`'n'`	liczbę (int lub float) / `nil`
`'*a'`	`'*a'`	`'a'`	cały plik (pusty napis jeśli koniec pliku)
`'*l'`	`'*l'`	`'l'`	następną linię (bez znaku EOL) / `nil`
	`'*L'`	`'L'`	następną linię (ze znakiem EOL) / `nil`
number	*number*	*number*	zadaną liczbę bajtów / `nil`

- Metoda `read` zwraca tyle wartości ile dostaje argumentów
- Lua 5.3 jest kompatybilna wstecz
- `'n'` pomija początkowe białe znaki
- Domyślny format to `'l'`.
- Wszystkie opcje oprócz `'n'` zwracają napisy

```
print (io.read('l', 'l') ) --> 'test1'  'test2'
print (io.read('all') )    --> 'test1\ntest2'
print (io.read('n') )      --> nil
```

I/O

io.lines()

Iteruje po liniach pliku `io.input()`. Zamyka plik.

```
for line in io.lines() do
  print ('-->', line)
end
--> test1
--> test2
```

Wbudowane strumienie

`io.stdin`, `io.stdout`, `io.stderr`

ZeroBraneStudio

Przy operacjach dyskowych *Project Directory* ma znaczenie i musi być brane pod uwagę przy definiowaniu ścieżek do plików.

Pełny model I/O

- Explicite posługujemy się uchwytami plików
- Funkcje biblioteki `io` zastępujemy metodami plików

`io.open (filename [, mode])`

- Otwiera plik w podanym trybie.
- Dostępne tryby: `r`, `w`, `a`, `r+`, `a+`, `w+`, plus sufix `b`
- W przypadku powodzenia zwraca uchwyt do pliku
- W przypadku błędu: `nil`, treść błędu, kod błędu

```
print (io.open('no-file.txt', 'r'))
--> nil no-file.txt: No such file or directory 2
print (io.open('no-file.txt', 'w'))
--> file (75782960)
```

Podstawowa obsługa błędów

assert (v [, message])

- Jeśli pierwszy argument to `nil` lub `false` wywołuje `error`
- Opcjonalnie przekazuje `message` jako komunikat błędu (domyślnie *assertion failed!*)
- Jeśli nie ma błędu – zwraca wszystkie swoje argumenty.

```
assert (io.open('no-file.txt', 'r'))
--> I:\ZeroBraneStudio...\tests.lua:123:
    no-file.txt: No such file or directory
stack traceback:
        [C]: in function 'assert'
        ...\tests.lua:123: in main chunk
        [C]: in ?

print (assert(io.open('no-file.txt', 'w')))
--> file (75782960)
```

Uchwyty plików

`file:close ()`

Zamyka plik. Pliki bez wskaźników są automatycznie zamykane przez garbage collector ("eventually").

`file:read (···)`

Działa jak `io.read` (które jest skrótem dla `io.input():read(···)`).

`file:write (···)`

Działa jak `io.write` (które jest skrótem dla `io.output():write(···)`).

`file:flush ()`

Zapisuje do pliku wszystkie przekazane dane.

Uchwyty plików

`file:setvbuf (mode [, size])`

Ustawia tryb buforowania (`size` ustala wielkośc bufora w dwóch ostatnich przypadkach):

- `'no'` – wszystkie rezultaty natychmiast zapisywane
- `'full'` – zapis tylko gdy bufor jest pełny lub explicite użyta metoda `flush`
- `'line'` – buforowane do napotkania nowej linii, lub inputu ze strony plików specjalnych (np. terminalu).

`file:seek ([whence [, offset]])`

Odczytuje i modyfikuje aktualną pozycję w pliku (jako bajty od początku pliku). Dodaje `offset` do `whence` które może być:

- `'set'` – bazą jest początek pliku
- `'cur'` – bazą jest aktualna pozycja w pliku
- `'end'` – bazą jest koniec pliku

Domyślne wartości (wywołanie `file:seek()` jest równoważne `file:seek('cur', 0)` i zwraca aktualną pozycję w pliku.

Uchwyty plików

file:lines (...)

- Pozwala na iterowanie po pliku korzystając z formatowania jak w funkcji `read`
- W przeciwieństwie do `io.lines` plik nie zostaje zamknięty.

```
-- x.txt:
-- 1 2 3
-- 2 3 4
-- 3 4 5
local f = io.open('x.txt')
for a, b, c in f:lines('n', 'n', 'n') do
  io.write(table.concat{'max (', a, ',', b, ',',
           c, ') = ', math.max(a, b, c), '\n'} )
end
--> max (1,2,3) = 3
--> max (2,3,4) = 4
--> max (3,4,5) = 5
```

Wczytywanie blokami

```
-- Kopiuje input na output blokami wielkości 8 KB
for block in io.input():lines(2^13) do
  io.write(block)
end
```

Obliczanie wielkości pliku

```
--- Gets the file size without changing position
-- @param file Handler to opened file
-- @return Size of the file
function fsize (file)
  local current = file:seek() -- save position
  local size = file:seek('end') -- get size
  file:seek('set', current) -- restore position
  return size
end
```

Dziękuję za uwagę

Za tydzień:
data i czas,
pattern-matching,
moduły,
programowanie funkcyjne
...?